



HAL
open science

GJK++: Leveraging Acceleration Methods for Faster Collision Detection

Louis Montaut, Quentin Le Lidec, Vladimír Petřík, Josef Sivic, Justin Carpentier

► **To cite this version:**

Louis Montaut, Quentin Le Lidec, Vladimír Petřík, Josef Sivic, Justin Carpentier. GJK++: Leveraging Acceleration Methods for Faster Collision Detection. IEEE Transactions on Robotics, In press. hal-04070039v2

HAL Id: hal-04070039

<https://hal.science/hal-04070039v2>

Submitted on 29 Nov 2023 (v2), last revised 3 Apr 2024 (v3)

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.

GJK++: Leveraging Acceleration Methods for Faster Collision Detection

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

1 **Abstract**—Collision detection is a fundamental problem in
 2 various domains, such as robotics, computational physics, and
 3 computer graphics. In general, collision detection is tackled as
 4 a computational geometry problem, with the so-called Gilbert,
 5 Johnson, and Keerthi (GJK) algorithm being the most adopted
 6 solution nowadays. While introduced in 1988, GJK remains the
 7 most effective solution to compute the distance or the collision
 8 between two 3D convex geometries. Over the years, it was
 9 shown to be efficient, scalable, and generic, operating on a
 10 broad class of convex shapes, ranging from simple primitives
 11 (sphere, ellipsoid, box, cone, capsule, etc.) to complex meshes
 12 involving thousands of vertices. In this article, we introduce
 13 several contributions to accelerate collision detection and distance
 14 computation between convex geometries by leveraging the fact
 15 that these two problems are fundamentally optimization problems.
 16 Notably, we establish that the GJK algorithm is a specific
 17 sub-case of the well-established Frank-Wolfe (FW) algorithm in
 18 convex optimization. By adapting recent works linking Polyak
 19 and Nesterov accelerations to Frank-Wolfe methods, we also
 20 propose two accelerated extensions of the classic GJK algorithm.
 21 Through an extensive benchmark over millions of collision pairs
 22 involving objects of daily life, we show that these two accelerated
 23 GJK extensions significantly reduce the overall computational
 24 burden of collision detection, leading to computation times that
 25 are up to two times faster. Finally, we hope this work will
 26 significantly reduce the computational cost of modern robotic
 27 simulators, allowing the speed-up of modern robotic applications
 28 that heavily rely on simulation, such as reinforcement learning
 29 or trajectory optimization.

30 **Index Terms**—Convex Optimization, Collision Detection, Com-
 31 putational Geometry, Computer Graphics, Simulation, Trajec-
 32 tory Optimization, Motion Planning

I. INTRODUCTION

34 **P**HYSICS engines designed to simulate rigid bodies are an
 35 essential tool used in a wide variety of applications, nota-
 36 bly in robotics, video games, and computer graphics [1]–[3].
 37 Collision detection, a crucial feature of any physics engine or
 38 robot motion planner [4]–[6], consists of finding which objects
 39 are colliding or not, *i.e.* are sharing at least one common point
 40 or if there exists a separating hyper-plane between both. As
 41 simulation often needs to deal with multiple objects and run in
 42 real-time (*i.e.*, in video games) or at very high frequencies (*i.e.*,

Louis Montaut is with Inria, Département d’Informatique de l’École Nor-
 male Supérieure, PSL Research University in Paris, France and also with the
 Czech Institute of Informatics, Robotics and Cybernetics in Prague, Czech
 Republic.

Vladimir Petrik and Josef Sivic are with the Czech Institute of Informatics,
 Robotics and Cybernetics, Czech Technical University in Prague.

Quentin Le Lidec and Justin Carpentier are with Inria and Département
 d’Informatique de l’École Normale Supérieure, PSL Research University in
 Paris, France.

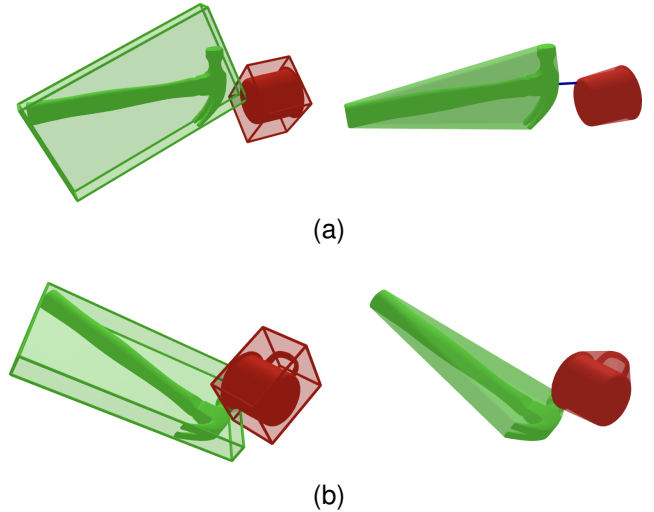


Fig. 1. Two distinct collision problems using shapes from the YCB dataset: in (a) the shapes \mathcal{A}_1 (in green) and \mathcal{A}_2 (in red) are not in collision ($\text{dist}(\mathcal{A}_1, \mathcal{A}_2) > 0$) whereas in (b) the shapes are in collision ($\text{dist}(\mathcal{A}_1, \mathcal{A}_2) = 0$). In the left column, the oriented bounding boxes (OBB) of the objects are represented in light colors. In the right column, the light colors represent the convex hull of each object. In both collision problems, (a) and (b), the broad phase finds a collision between the object’s OBBs; the narrow phase must thus be called to confirm or infirm the collision. The right column corresponds to the narrow phase in which the GJK algorithm is called on the objects’ convex hulls. In this paper, we propose the Polyak-accelerated GJK and Nesterov-accelerated GJK algorithms in order to accelerate collision detection.

in robotics), collision detection must be carried out as fast as
 possible. To reduce computational times, collision detection is
 usually decomposed into two phases thoroughly covered in [7].
 The first phase is the so-called *broad phase* which consists
 in identifying which pair of simulated objects are potentially
 colliding. The broad phase relies on the simulated objects’
 bounding volumes, as shown in Fig. 1, allowing to quickly
 assess if the objects are *not* in collision. The second phase is
 the so-called *narrow phase* in which each pair identified in
 the broad phase is tested to check whether a collision is truly
 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 them into a collection of convex sub-
 shapes [8]. 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

1 minimization problem of the form:

$$d_{1,2} = \min_{\mathbf{x}_1 \in \mathcal{A}_1, \mathbf{x}_2 \in \mathcal{A}_2} \|\mathbf{x}_1 - \mathbf{x}_2\|^2 \quad (1)$$

and $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) = \sqrt{d_{1,2}}$,

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

8 Problem (1) allows us to consider both the *distance com-*
 9 *putation* problem and the computationally cheaper *Boolean*
 10 *collision check* as one single convex optimization problem. In
 11 the distance computation problem, we aim at computing the
 12 separation distance between \mathcal{A}_1 and \mathcal{A}_2 , denoted $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$,
 13 *i.e.* the distance between their closest points. This distance
 14 is helpful in some applications such as collision-free path
 15 planning [9], [10], especially for pairs of objects entering the
 16 narrow phase. If the broad phase has not selected a pair of
 17 objects, a cheap estimate of $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ is usually enough [7].
 18 In the Boolean collision check, we only aim at determin-
 19 ing if \mathcal{A}_1 and \mathcal{A}_2 intersect, and computing $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ is
 20 unnecessary. However, we will later see that the Boolean
 21 collision check is a sub-problem of the distance computation
 22 problem: solving (1) can be early-stopped once a separating
 23 plane between \mathcal{A}_1 and \mathcal{A}_2 has been found. In the rest of this
 24 paper, we will use the generic term “collision detection” to
 25 refer to distance computation and Boolean collision checking
 26 altogether. We will specify when the distinction is needed.

27 **Contributions and paper outline.** Our work builds on the
 28 seminal works by [11] and [12] as well as on the work of [13],
 29 [14] to globally accelerate distance computation and collision
 30 checking algorithms between convex shapes. We make the
 31 following contributions:

- 32 \hookrightarrow In Sec. II, we provide an in-depth tutorial on the state-of-
 33 the-art Gilbert-Johnson-Keerthi (GJK) algorithm used to
 34 solve collision detection problems. Although it is often
 35 presented and discussed as a computational geometry
 36 algorithm, we show that GJK is, in fact, a sub-case of the
 37 much older fully-corrective Frank-Wolfe (FW) algorithm;
- 38 \hookrightarrow In Sec. III, we adapt recent works on Polyak and
 39 Nesterov-accelerated FW to accelerate both the distance
 40 computation and the Boolean collision check problems;
- 41 \hookrightarrow In Sec. IV, we empirically analyze the convergence of
 42 our proposed approach on two large shape benchmarks.
 43 Results show a faster convergence of our approach lead-
 44 ing to a computational time up to two times faster than
 45 the state-of-the-art GJK algorithm on both distance com-
 46 putation and Boolean collision checking. We also provide
 47 a benchmark against other collision detection algorithms
 48 from the libCCD [15], FCL [16] and BulletCollision [17]
 49 collision detection libraries which are used in physics
 50 simulators like Drake [18], MuJoCo [3], ODE [19] and
 51 Bullet [1].
- 52 \hookrightarrow We empirically show that GJK-like algorithms, which our
 53 proposed methods belong to, are superior by orders of

magnitude to generic quadratic programming solvers on
 collision detection problems;

- \hookrightarrow Finally, we show that our methods can be used in any
 physics simulator by benchmarking them on trajectories
 generated by the Bullet simulator. Like GJK, our methods
 can benefit from being warm-started using the previous
 simulation time steps, enabling temporal coherence for
 our proposed accelerated collision detection algorithms.

This article is an extended version of a previously published
 paper [20] which presented the Nesterov-accelerated GJK
 algorithm. To expand on our previous work, we introduce the
 Polyak-accelerated GJK algorithm, provide additional bench-
 marks, notably against existing collision detection libraries,
 and show that our proposed methods can be used in the context
 of physics simulation by benefiting from being warm-started
 using previous simulation steps.

In the rest of this section, we first formulate the problem of
 collision detection and then provide an overview of collision
 detection related works.

Related work. The so-called Gilbert-Johnson-Keerthi algo-
 rithm (GJK) [12] is the most well-known algorithm for colli-
 sion detection between two convex shapes. It can handle the
 distance computation and the Boolean collision check [21].
 The expanding polytope algorithm (EPA) [22], an extension
 to GJK, can compute the penetration depth *i.e.* the norm of the
 separation vector, when shapes are in collision. The separation
 vector is the vector of smallest norm needed to translate one of
 the two shapes such that the two shapes do not intersect. The
 EPA solves a non-convex and more complex problem than (1),
 which is not the focus of this paper.

Most alternatives to GJK in the literature focus on com-
 puting collisions between convex polyhedra, such as the Lin-
 Canny algorithm [23] or the V-Clip [24] algorithm. Although
 GJK is equivalent in performance to these algorithms [25],
 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 dif-
 ference are used to cleverly compute support vectors on
 the Minkowski difference (these notions are introduced and
 detailed in Sec. II). GJK can thus handle collision detection,
 and distance computation for many different shapes such as
 convex polyhedra and basic primitives (*i.e.*, spheres, ellipsoids,
 cylinders, capsules etc.) [7], [21], [26]. The Minkowski Portal
 Refinement (MPR) [27] is a variant of the GJK algorithm
 which is slightly simpler to implement; however, MPR can
 only perform boolean collision checks as it cannot be used
 to perform distance computation between non-overlapping
 shapes. Overall, the generality of GJK, efficiency, good preci-
 sion, and ease of implementation make it the state-of-the-art
 algorithm for collision detection between two convex shapes.

Collision detection has been casted into a convex opti-
 mization problem for many years [12], [28], [29]. However,
 collision detection is traditionally presented and considered
 mainly as a computational geometry problem [7], [21], [24],
 [25], [30], [31]. Over the years, this computational geometric
 perspective allowed enhancing the computational efficiency of
 GJK, thanks to improvements to its internal sub-routines [21],

[30]. However, we argue that this view has also limited collision detection improvement. Instead, we propose to tackle collision from the perspective of convex optimization. This correlates with some observations raised in the original GJK papers. Indeed, as briefly mentioned already in their 1988 paper [12] and brought up again by [29], [32], the ideas developed by Gilbert, Johnson, and Keerthi are rooted in convex optimization, notably in the works of [33] and [34] for solving Minimum-Norm Point (MNP) problems. This article proposes exploiting the Frank-Wolfe convex optimization setting to tackle collision detection. In particular, by leveraging recent progresses in acceleration methods in convex optimization [35], we show 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.

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* 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 [33], [34], [36]–[40]. In [38] and [40], widely used corrections of the FW algorithm are analyzed, and 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 [40], applied to solving a MNP problem. Finally, recent works have also tried accelerating the FW algorithm by applying the so-called Nesterov acceleration [41], a classic acceleration technique in unconstrained optimization. Nesterov momentum has been successfully added by [14] to accelerate FW.

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 (FW) setting. The first part of this section is a tutorial on collision detection, starting with FW and gradually working towards GJK. We then show that the GJK algorithm can be seen as a variant of the FW algorithm that leverages properties of convex 3D shapes to lower the computational complexity drastically.

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 case of $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) > 0$, finding a strictly positive lower bound on $d_{1,2}$ to solve the collision problem is sufficient. In the context of convex shapes, this is often simpler than computing the distance between the two shapes [10] 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 more straightforward Boolean collision check covered later in this section. Results for the particular Boolean collision checking case are analyzed in the experimental section IV.

Collision detection from the perspective of quadratic programming. From the perspective of numerical optimization, the first idea is to look at problem (1) through the lens of quadratic programming. In the case of meshes, which are shapes represented by soups of 3D points and which faces represented as triangles, we can use the implicit description of a convex mesh as a linear inequality of the form $Ax \leq b$. The collision detection problem between two meshes can thus be cast as a quadratic programming (QP) problem:

$$\begin{aligned} d_{1,2} = \min_{\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^3} & \quad \|\mathbf{x}_1 - \mathbf{x}_2\|^2 \\ \text{s.t.} & \quad A_1 \mathbf{x}_1 \leq \mathbf{b}_1 \\ & \quad A_2 \mathbf{x}_2 \leq \mathbf{b}_2. \end{aligned} \quad (2)$$

While many off-the-shelf solvers exist to solve QP problems, their performances scale poorly with respect to the number of constraints [42]. This is especially true in the presence of complex meshes composed of hundreds or thousands of vertices, for which QP solvers can take a few milliseconds to assess a collision, as we experimentally highlight in Sec. IV-C. Having established that solving collision detection requires dedicated methods, we turn our attention to solutions such as GJK, which has been shown to operate on a large class of shapes, ranging from simple primitives to very complex meshes.

Recasting the distance computation problem onto the Minkowski difference. The first important idea of 1988's paper by Gilbert, Johnson, and Keerthi [12] is to recast the distance computation problem onto the Minkowski difference \mathcal{D} of the shapes and defined as follows:

$$\mathcal{D} = \mathcal{A}_1 - \mathcal{A}_2 = \{\mathbf{x} = \mathbf{x}_1 - \mathbf{x}_2 \mid \mathbf{x}_1 \in \mathcal{A}_1, \mathbf{x}_2 \in \mathcal{A}_2\} \subset \mathcal{C}, \quad (3)$$

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 isomorphic to \mathbb{R}^n , we distinguish between the two to highlight the change in perspective. In Fig. 2, we illustrate the link between a pair of two convex shapes and their corresponding Minkowski difference. 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, illustrated in Fig. 2, 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 $\mathbf{0}_C$, lies inside the Minkowski difference \mathcal{D} , *i.e.* $\mathbf{0}_C = \mathbf{x}_1 - \mathbf{x}_2$ for some $\mathbf{x}_1 \in \mathcal{A}_1$ and $\mathbf{x}_2 \in \mathcal{A}_2$.
- 3) If \mathcal{A}_1 and \mathcal{A}_2 are not intersecting, the projection of $\mathbf{0}_C$ onto \mathcal{D} , $\mathbf{x}^* = \text{proj}_{\mathcal{D}}(\mathbf{0}_C)$, corresponds to two vectors

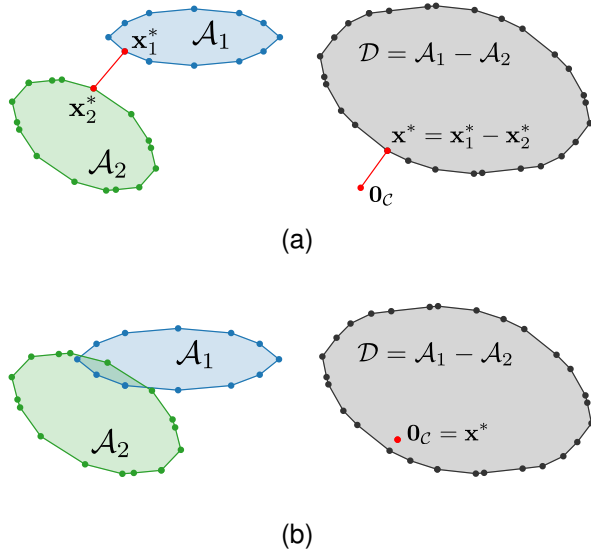


Fig. 2. (a) Distant vs. (b) overlapping pairs of shapes and their respective Minkowski difference. Left column: two convex shapes in 2D. Right column: the Minkowski difference \mathcal{D} of \mathcal{A}_1 and \mathcal{A}_2 . Since \mathcal{A}_1 and \mathcal{A}_2 are convex, \mathcal{D} is also convex.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

1 $\mathbf{x}_1^* \in \mathcal{A}_1$ and $\mathbf{x}_2^* \in \mathcal{A}_2$, also called witness vectors in the
 2 computational geometry literature [7]. Contrary to \mathbf{x}^* ,
 3 these vectors \mathbf{x}_1^* and \mathbf{x}_2^* are not necessarily unique, as
 4 is the case for non-strictly convex shapes such as two
 5 parallel boxes.

6 4) Finally, we always have $\|\mathbf{x}^*\| = \text{dist}(\mathcal{A}_1, \mathcal{A}_2)$.

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

$$d_{1,2} = \min_{\mathbf{x} \in \mathcal{D}} \|\mathbf{x} - \mathbf{0}_C\|^2 = \min_{\mathbf{x} \in \mathcal{D}} \|\mathbf{x}\|^2. \quad (4)$$

9 The convex optimization problem (4) is equivalent to (1)
 10 and is known as a Minimum-Norm Point problem in the
 11 optimization literature [34], [40], [43]. In our case, $\mathbf{0}_C \in \mathcal{C} = \mathbb{R}^n$
 12 is the null vector *i.e.* the origin of the collision
 13 space. We thus aim at finding the point in \mathcal{D} with the lowest
 14 norm. This vector \mathbf{x}^* is the *optimal solution* to (4), given
 15 by $d_{1,2} = \|\mathbf{x}^*\|^2 = \text{dist}(\mathcal{A}_1, \mathcal{A}_2)^2$.

16 Directly computing the Minkowski difference \mathcal{D} is neither
 17 analytically tractable nor computationally efficient. Most of
 18 the first and second-order methods for constrained convex
 19 optimization problems, such as projected gradient descent or
 20 interior point methods [44], are thus sub-optimal choices.
 21 However, computing support vectors of the Minkowski differ-
 22 ence \mathcal{D} , a notion defined hereinafter in this section, is relatively
 23 simple and largely demonstrated by [12]. As we discuss next,
 24 solving convex optimization problems by computing support
 25 vectors is the strength of the Frank-Wolfe algorithm and its
 26 variants [38].

27 **Distance computation using the Frank-Wolfe algorithm.**
 28 The Frank-Wolfe algorithm (FW) [11] is one of the oldest

Algorithm 1 Frank-Wolfe algorithm with linesearch [38]

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

For $k=0, 1, \dots$ do

- 1: $\mathbf{d}_k = \nabla f(\mathbf{x}_k)$ ▷ Direction of support
 - 2: $\mathbf{s}_k \in \arg \min_{\mathbf{s} \in \mathcal{D}} \langle \mathbf{d}_k, \mathbf{s} \rangle (= S_{\mathcal{D}}(\mathbf{d}_k))$ ▷ Support (8)
 - 3: **If** $g_{FW}(\mathbf{x}_k) \leq \epsilon$, **return** $f(\mathbf{x}_k)$ ▷ Duality gap (16)
 - 4: $\gamma_k = \arg \min_{\gamma \in [0,1]} f(\gamma \mathbf{x}_k + (1-\gamma)\mathbf{s}_k)$ ▷ Linesearch
 - 5: $\mathbf{x}_{k+1} = \gamma_k \mathbf{x}_k + (1-\gamma_k)\mathbf{s}_k$ ▷ Update iterate
*In the case of the distance computation problem (4),
 where $f(\mathbf{x}) = \|\mathbf{x}\|^2$, line 4-5 correspond to projecting $\mathbf{0}_C$
 on the segment $[\mathbf{x}_k, \mathbf{s}_k]$:*
 - 6: $\mathbf{x}_{k+1} = \text{proj}_{[\mathbf{x}_k, \mathbf{s}_k]}(\mathbf{0}_C)$ ▷ Project $\mathbf{0}_C$ on $[\mathbf{x}_k, \mathbf{s}_k]$
-

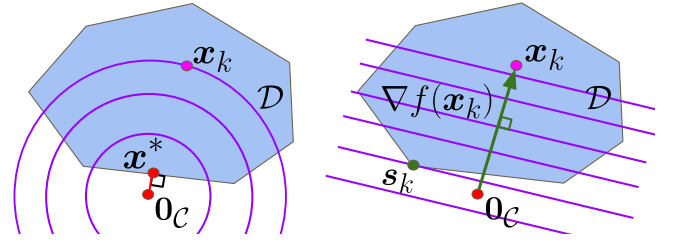


Fig. 3. Computing a support vector \mathbf{s}_k in direction $\nabla f(\mathbf{x}_k)$ on convex set \mathcal{D} . We illustrate with the example of distance computation. On the left, we draw the Minkowski difference \mathcal{D} of which point of minimum norm (MNP) is \mathbf{x}^* *i.e.* \mathbf{x}^* is the projection of $\mathbf{0}_C$ onto \mathcal{D} , $\mathbf{x}^* = \text{proj}_{\mathcal{D}}(\mathbf{0}_C)$. The iterate at iteration k of the FW algorithm is \mathbf{x}_k . In purple we draw the level sets of the function $f(\mathbf{x}) = \|\mathbf{x}\|^2$. On the right, we draw in purple the level sets of the linearization of f at iterate \mathbf{x}_k , h_k . The first step of the FW algorithm is to compute support vector \mathbf{s}_k in the direction of $\nabla f(\mathbf{x}_k)$ (green arrow), $\mathbf{s}_k \in S_{\mathcal{D}}(\nabla f(\mathbf{x}_k))$. In the second step of the FW algorithm, we compute \mathbf{x}_{k+1} as a convex combination of \mathbf{x}_k and \mathbf{s}_k *i.e.* \mathbf{x}_{k+1} is a point on the segment $[\mathbf{x}_k, \mathbf{s}_k]$.

convex optimization methods and solves the following con-
 strained optimization problem:

$$f(\mathbf{x}^*) = \min_{\mathbf{x} \in \mathcal{D}} f(\mathbf{x}), \quad (5)$$

where $f: \mathbb{R}^n \rightarrow \mathbb{R}$ is a convex and differentiable function and
 \mathcal{D} is a compact convex set. For our distance computation
 problem (4), we use $f(\mathbf{x}) = \|\mathbf{x}\|^2$ and the Minkowski
 difference \mathcal{D} as convex constraint set. As a side note, the
 following discussed algorithms all require an initial starting
 point $\mathbf{x}_0 \in \mathcal{D}$. Shapes used in physics engines are usually
 attached to a frame to keep track of their position and
 orientation in space. We denote $\mathbf{c}^1 \in \mathcal{A}_1$ and $\mathbf{c}^2 \in \mathcal{A}_2$ the
 origins of the frames attached to \mathcal{A}_1 and \mathcal{A}_2 , respectively. In
 the rest of this paper, we take $\mathbf{x}_0 = \mathbf{c}^1 - \mathbf{c}^2$.

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 \mathbf{x}^* of (5). If
 we denote by \mathbf{x}_k the estimate of \mathbf{x}^* at iteration k , these two
 steps correspond to:

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

1 In the following, we detail these steps in the context of
 2 distance computation. At iteration k , the current iterate \mathbf{x}_k
 3 is the estimate of the optimal solution \mathbf{x}^* and $f(\mathbf{x}_k)$ is the
 4 estimate of the optimal value of (5), $f(\mathbf{x}^*)$, at iteration k . We
 5 write the linearization of the function f at \mathbf{x}_k and denote it
 6 as h_k :

$$h_k(\mathbf{s}) = f(\mathbf{x}_k) + \langle \nabla f(\mathbf{x}_k), \mathbf{s} - \mathbf{x}_k \rangle \quad (6)$$

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

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

$$\mathbf{s}_k \in \arg \min_{\mathbf{s} \in \mathcal{D}} h_k(\mathbf{s}) = \arg \min_{\mathbf{s} \in \mathcal{D}} \langle \nabla f(\mathbf{x}_k), \mathbf{s} \rangle. \quad (7)$$

13 Fig. 3 gives a graphical understanding of support \mathbf{s}_k . The
 14 vector \mathbf{s}_k belongs to \mathcal{D} and is in the *most* opposite direction
 15 w.r.t. $\nabla f(\mathbf{x}_k)$. In order to highlight the importance of the
 16 direction in which a support \mathbf{s}_k is computed, we now introduce
 17 the notion of *support direction* and *support function*. Given a
 18 support direction $\mathbf{d} \in \mathbb{R}^n$, the support function $S_{\mathcal{D}}$ returns a
 19 set of \mathcal{D} and is defined as:

$$S_{\mathcal{D}}(\mathbf{d}) = \arg \min_{\mathbf{s} \in \mathcal{D}} \langle \mathbf{d}, \mathbf{s} \rangle \subset \mathcal{D}. \quad (8)$$

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

$$\forall \mathbf{d} \in \mathbb{R}^n, \forall \alpha > 0, S_{\mathcal{D}}(\alpha \mathbf{d}) = S_{\mathcal{D}}(\mathbf{d}). \quad (9)$$

25 A support $\mathbf{s}_k \in \mathcal{D}$ at iteration k is thus computed in the
 26 direction $\mathbf{d}_k = \nabla f(\mathbf{x}_k)$ and belongs to $S_{\mathcal{D}}(\nabla f(\mathbf{x}_k))$, $\mathbf{s}_k \in$
 27 $S_{\mathcal{D}}(\nabla f(\mathbf{x}_k))$.

28 We now explain how to compute the support vector \mathbf{s}_k in
 29 the case of the distance computation problem (4) where we
 30 minimize $f(\mathbf{x}) = \|\mathbf{x}\|^2$ on the Minkowski difference \mathcal{D} of \mathcal{A}_1
 31 and \mathcal{A}_2 . First, we have $\nabla f(\mathbf{x}) = 2\mathbf{x}$. Therefore, in the case
 32 of problem (4), it follows that:

$$\mathbf{s}_k \in S_{\mathcal{D}}(\mathbf{x}_k) = \arg \min_{\mathbf{s} \in \mathcal{D}} \langle \mathbf{x}_k, \mathbf{s} \rangle. \quad (10)$$

33 As demonstrated by [12], any vector $\mathbf{s} \in S_{\mathcal{D}}(\mathbf{d})$ related to the
 34 Minkowski difference can be decomposed as the difference be-
 35 tween two support vectors $\mathbf{s}_{\mathcal{A}_1} \in S_{\mathcal{A}_1}(\mathbf{d})$ and $\mathbf{s}_{\mathcal{A}_2} \in S_{\mathcal{A}_2}(-\mathbf{d})$
 36 over the two individual shapes, leading to the following
 37 relation:

$$\mathbf{s} = \mathbf{s}_{\mathcal{A}_1} - \mathbf{s}_{\mathcal{A}_2} \in S_{\mathcal{D}}(\mathbf{d}). \quad (11)$$

38 Equation (11) shows that we can construct a support of the
 39 Minkowski difference from the supports of the original shapes.
 40 This property highlights the powerful change of perspective
 41 of working on the Minkowski difference. Indeed, there exists
 42 a large number of shapes for which computing supports
 43 is simple: spheres, ellipsoids, cylinders, capsules, polytopes
 44 etc. [7], [21], [26]. Fig. 4 illustrates the construction of a
 45 support of the Minkowski difference \mathcal{D} using the supports of
 46 the original shapes \mathcal{A}_1 and \mathcal{A}_2 .

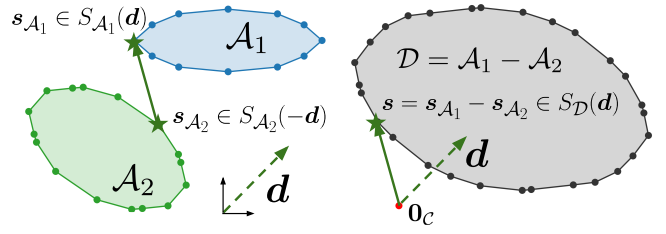


Fig. 4. Computing a support vector on the Minkowski difference using
 support vectors (represented by star shapes in the drawing) on the individual
 shapes. The vector $\mathbf{s}_{\mathcal{A}_1}$ is a support vector of shape \mathcal{A}_1 in direction \mathbf{d} . The
 vector $\mathbf{s}_{\mathcal{A}_2}$ is a support vector of shape \mathcal{A}_2 in direction $-\mathbf{d}$. The constructed
 vector $\mathbf{s} = \mathbf{s}_{\mathcal{A}_1} - \mathbf{s}_{\mathcal{A}_2}$ is a support vector of the Minkowski difference \mathcal{D}
 in the direction \mathbf{d} .

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

$$\mathbf{x}_{k+1} = \gamma_k \mathbf{x}_k + (1 - \gamma_k) \mathbf{s}_k, \quad (12)$$

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

$$\begin{aligned} \gamma_k &= \arg \min_{\gamma \in [0, 1]} f(\gamma \mathbf{x}_k + (1 - \gamma) \mathbf{s}_k) \\ \mathbf{x}_{k+1} &= \gamma_k \mathbf{x}_k + (1 - \gamma_k) \mathbf{s}_k. \end{aligned} \quad (13)$$

In the distance computation case where $f(\mathbf{x}) = \|\mathbf{x}\|^2$, this
 linesearch (13) is equivalent to projecting $\mathbf{0}_C$ onto the seg-
 ment $[\mathbf{x}_k, \mathbf{s}_k]$, $\mathbf{x}_k = \text{proj}_{[\mathbf{x}_k, \mathbf{s}_k]}(\mathbf{0}_C)$ (line 4 in Alg. 1). Since \mathcal{D}
 is convex, both (12) and (13) updates are guaranteed to remain
 in \mathcal{D} .

Stopping criteria. As Frank-Wolfe deals with convex prob-
 lems, the *duality gap* associated with problem (5) can be used
 as a stopping criterion. Due to its convexity, the function f is
 always above its linearization. Otherwise said, for any $\mathbf{x} \in \mathbb{R}^n$
 and any $\mathbf{s} \in \mathbb{R}^n$:

$$f(\mathbf{s}) \geq f(\mathbf{x}) + \langle \nabla f(\mathbf{x}), \mathbf{s} - \mathbf{x} \rangle. \quad (14)$$

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

$$f(\mathbf{x}) - f(\mathbf{x}^*) \leq - \min_{\mathbf{s} \in \mathcal{D}} \langle \nabla f(\mathbf{x}), \mathbf{s} - \mathbf{x} \rangle = g_{\text{FW}}(\mathbf{x}). \quad (15)$$

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

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq g_{\text{FW}}(\mathbf{x}_k) = \langle \nabla f(\mathbf{x}_k), \mathbf{x}_k - \mathbf{s}_k \rangle, \quad (16)$$

where $\mathbf{s}_k \in S_{\mathcal{D}}(\nabla f(\mathbf{x}_k))$ is the support vector computed at it-
 eration k in the direction of $\nabla f(\mathbf{x}_k)$. The duality-gap $g_{\text{FW}}(\mathbf{x}_k)$
 serves as a convergence criterion for the Frank-Wolfe method
 and is cheap to compute. Applied to the distance computation
 problem (4), the duality gap at iteration k , $g_{\text{FW}}(\mathbf{x}_k)$, guarantees
 that:

$$\|\mathbf{x}_k\|^2 - \|\mathbf{x}^*\|^2 \leq g_{\text{FW}}(\mathbf{x}_k) = 2 \langle \mathbf{x}_k, \mathbf{x}_k - \mathbf{s}_k \rangle. \quad (17)$$

Algorithm 2 Boolean collision checking: separating plane condition

Insert after line 2 in Alg. 1:

1: **If** $\max(0, \langle \mathbf{d}_k / \|\mathbf{d}_k\|, \mathbf{s}_k \rangle) > \epsilon_{\text{col}}$, **return False**
If after termination $d_{1,2} \leq \epsilon_{\text{col}}$, **return True, otherwise return False.**

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

$$\|\mathbf{x}_k - \mathbf{x}^*\|^2 \leq \|\mathbf{x}_k\|^2 - \|\mathbf{x}^*\|^2 \leq g_{\text{FW}}(\mathbf{x}_k). \quad (18)$$

3 Inequality (18) is useful in practice as it allows the fine
4 control of the desired tolerance on the distance to the optimal
5 solution \mathbf{x}^* (line 3 in Alg. 1). Indeed, if ones wants to compute
6 an estimate \mathbf{x} of the optimal solution \mathbf{x}^* at precision ϵ ,
7 meaning that $\|\mathbf{x} - \mathbf{x}^*\| \leq \sqrt{\epsilon}$, it is sufficient to check that
8 $g_{\text{FW}}(\mathbf{x}) \leq \epsilon$.

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

$$\langle \mathbf{d}, \mathbf{s} \rangle > 0, \mathbf{s} \in S_{\mathcal{D}}(\mathbf{d}), \quad (19)$$

15 then the plane supported by the vector \mathbf{d} separates \mathcal{D}
16 from $\mathbf{0}_C$ [21]. This also means that, in the case where the
17 two shapes intersect, collision checking has the same com-
18 putational complexity as distance computation. In general, at
19 iteration k , the value $\max(0, \langle \mathbf{d}_k / \|\mathbf{d}_k\|, \mathbf{s}_k \rangle)$ is a lower bound
20 on the distance between the shapes, $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$, and it can
21 be used as a stopping criterion in the boolean collision check.
22 As soon as the lower bound on $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ is guaranteed
23 to be positive, the algorithm can be stopped. Otherwise, the
24 algorithm continues until the stopping criterion defined by (18)
25 is met. As shown in Alg. 2, we add this separating plane
26 condition before line 2 in Alg. 1. The condition on the lower
27 bound of $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ is met relative to a threshold ϵ_{col} . This
28 threshold sets at what distance the shapes are considered to
29 be in collision and depends on the application. In practice,
30 since we use double-precision floats in our benchmarks (see
31 Sec. IV), the test on the lower bound of $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ can be
32 computed down to machine precision. This means the distance
33 computation threshold ϵ can take values down to 10^{-12} in (18).
34 This corresponds to a precision on the distance between shapes
35 on the order of micro-meters ($\sqrt{\epsilon} = 10^{-6}\text{m}$). In robotics
36 applications where the entire pipeline is no more precise than
37 millimeters, it is often sufficient to consider that below a
38 threshold of 10^{-4}m (i.e $\epsilon = 10^{-8}$) the shapes are in collision;
39 above that threshold the shapes are not in collision. In our
40 applications, we use $\epsilon_{\text{col}} = \sqrt{\epsilon}$.

42 **Computing support vector on meshes.** While the support
43 vector of basic primitives (sphere, ellipsoid, box, etc.) presents
44 closed-form solutions, this is not the case for meshes. In the
45 case of convex meshes, an efficient approach for computing the

Algorithm 3 Frank-Wolfe algorithm with line-search (see Alg. 1) rewritten with active-sets and applied to the distance computation problem (4)

Let $\mathbf{x}_0 \in \mathcal{D}$, $W_0 = \{\mathbf{x}_0\}$, $\epsilon > 0$

For $k=0, 1, \dots$ **do**

- | | |
|--|--|
| 1: $\mathbf{d}_k = \mathbf{x}_k$ | ▷ Direction of support |
| 2: $\mathbf{s}_k \in S_{\mathcal{D}}(\mathbf{d}_k)$ | ▷ Support (8) |
| 3: If $g_{\text{FW}}(\mathbf{x}_k) \leq \epsilon$, return $f(\mathbf{x}_k)$ | ▷ Duality gap (16) |
| 4: $\widetilde{W}_{k+1} = W_k \cup \{\mathbf{s}_k\}$ | ▷ Augment active-set |
| 5: $\mathbf{x}_{k+1} = \text{proj}_{\text{conv}(\widetilde{W}_{k+1})}(\mathbf{0}_C)$ | ▷ Project $\mathbf{0}_C$ on $\text{conv}(\widetilde{W}_{k+1})$ |
| 6: $W_{k+1} = \{\mathbf{x}_{k+1}\}$ | ▷ Update active-set |
-

support direction of meshes is the *hill-climbing* algorithm [26],
which allows retrieving the supporting vertex or face of the
meshes thanks to a simple neighbor-descent procedure. Yet,
this procedure is sensitive to the initial-guess solution. By
leveraging Nesterov and Polyak acceleration schemes intro-
duced in Sec. III, which both tend to reduce the oscillations
hindered by gradient-descent type algorithms, we show in
Sec. IV that this helps the hill-climbing algorithm to perform
less iterations in practice, leading to faster computation times.

The Frank-Wolfe active-set. As with many gradient-descent
algorithms, the FW method tends to zig-zag towards the
optimal solution [40], slowing down the convergence to the
optimum. This behavior is undesired and amplified if the
optimal solution \mathbf{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 [40], also called *active-set* in the
optimization literature [44]. In the FW setting, the active set
at iteration k , denoted $W_k = \{\mathbf{s}^0, \dots, \mathbf{s}^r\} \subset \mathcal{D}$, is the set
of vectors in \mathcal{D} used by the algorithm to maintain a convex
combination of the iterate \mathbf{x}_k :

$$\mathbf{x}_k = \sum_{i=0}^r \lambda^i \mathbf{s}^i, \sum_{i=0}^r \lambda^i = 1 \text{ with } \mathbf{s}^i \in W_k \subset \mathcal{D} \text{ and } \lambda^i > 0. \quad (20)$$

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

- At iteration k , the active-set is only composed of \mathbf{x}_k , $W_k = \{\mathbf{x}_k\}$.
- The active-set W_k is then augmented by computing a support \mathbf{s}_k (line 2 in Alg. 3) to obtain $\widetilde{W}_{k+1} = \{\mathbf{x}_k, \mathbf{s}_k\}$ (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 $[\mathbf{x}_k, \mathbf{s}_k]$. For the distance computation problem (4), this linesearch operation is equivalent to projecting $\mathbf{0}_C$ onto the segment $[\mathbf{x}_k, \mathbf{s}_k]$ (line 5 in Alg. 3).
- Finally, the active-set is updated $W_{k+1} = \{\mathbf{x}_{k+1}\}$ (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 [40]. In the optimization literature, a rich and wide variety of variants

1 of the FW algorithm have been introduced to efficiently cope
 2 with the active set in order to improve the convergence rate
 3 of the FW method [34], [36], [37], [45], [46]. However,
 4 these variants remain too generic and are not suited for the
 5 specific problem of collision detection. In the following, we
 6 propose instead incorporating the active-set strategy used in
 7 GJK within the Frank-Wolfe setting.

8 **Connection between GJK and Frank-Wolfe.** In the case
 9 of collision detection, [12] developed an efficient strategy to
 10 handle the active set at a minimal cost. To represent the current
 11 estimate \mathbf{x}_k and the optimal solution \mathbf{x}^* , GJK exploits the
 12 concept of *simplexes* in \mathbb{R}^3 . A simplex in \mathbb{R}^n corresponds to
 13 a set containing *at most* $n + 1$ vectors of \mathbb{R}^n , and the *rank* r
 14 of a simplex is the number of vectors it contains ($0 < r \leq$
 15 $n + 1$). For 3-dimensional spaces, a simplex corresponds either
 16 to a point ($r = 1$), a segment ($r = 2$), a triangle ($r = 3$),
 17 or a tetrahedron ($r = 4$). Similarly to the simplex methods
 18 for Linear Programming [47], the Carathéodory theorem [48]
 19 motivates the use of simplexes. Let \mathcal{Y} be a set of $N \geq n$
 20 vectors in \mathbb{R}^n , $\mathcal{Y} = \{\mathbf{y}^i \in \mathbb{R}^n\}_{0 \leq i \leq N}$. The Carathéodory
 21 theorem states that any vector $\mathbf{x} \in \text{conv}(\mathcal{Y})$ can be expressed
 22 as the convex combination of at most $n + 1$ vectors of \mathcal{Y} :

$$\mathbf{x} = \sum_{j=0}^r \lambda^j \mathbf{y}^j, \text{ with } \mathbf{y}^j \in \mathcal{Y}, \lambda^j > 0, \sum_{i=0}^r \lambda^i = 1. \quad (21)$$

23 Hence, any vector in \mathcal{D} , and particularly the optimal solu-
 24 tion $\mathbf{x}^* \in \mathcal{D} = \text{conv}(\mathcal{D})$ of the distance computation
 25 problem (4), can be identified as a convex combination of
 26 the vectors composing a simplex W . Relying on simplexes
 27 is attractive as there is no need to run any algorithm to
 28 compute the convex hull of a simplex as they are convex
 29 by construction. Frank-Wolfe algorithms may operate on more
 30 complex active sets, which might become hard to tackle from
 31 a computational point of view [38], [40]. In other words, the
 32 problem of finding the optimal solution \mathbf{x}^* can be reformulated
 33 as the problem of identifying the optimal simplex W^* on
 34 which \mathbf{x}^* can be decomposed into a convex combination. This
 35 is precisely the approach followed by GJK that we now detail
 36 as well as illustrate in Fig. 5.

37 At iteration k of GJK, the current iterate \mathbf{x}_k is a convex
 38 combination of the vectors composing the simplex W_k of
 39 rank $r_k \leq n$. This corresponds to Fig. 5a. To update \mathbf{x}_k
 40 and W_k , the following procedure is applied:

- 41 • After computing support vector \mathbf{s}_k (line 2 in Alg. 3,
 42 illustrated in Fig. 5b), we add \mathbf{s}_k to W_k to ob-
 43 tain $\widetilde{W}_{k+1} = W_k \cup \{\mathbf{s}_k\}$ (line 4 in Alg. 3). The set \widetilde{W}_{k+1}
 44 is now a simplex of rank $\widetilde{r}_{k+1} \leq n + 1$, as shown in
 45 Fig. 5c.
- 46 • We then minimize function $f(\mathbf{x}) = \|\mathbf{x}\|^2$ on \widetilde{W}_{k+1}
 47 to obtain \mathbf{x}_{k+1} , corresponding to projecting $\mathbf{0}_C$
 48 onto \widetilde{W}_{k+1} : $\mathbf{x}_{k+1} = \text{proj}_{\text{conv}(\widetilde{W}_{k+1})}(\mathbf{0}_C)$ ¹ (line 5 in
 49 Alg. 3). This projection is illustrated in figures 5c and 5d.
- 50 • We then have two cases, summarized in Alg 4:

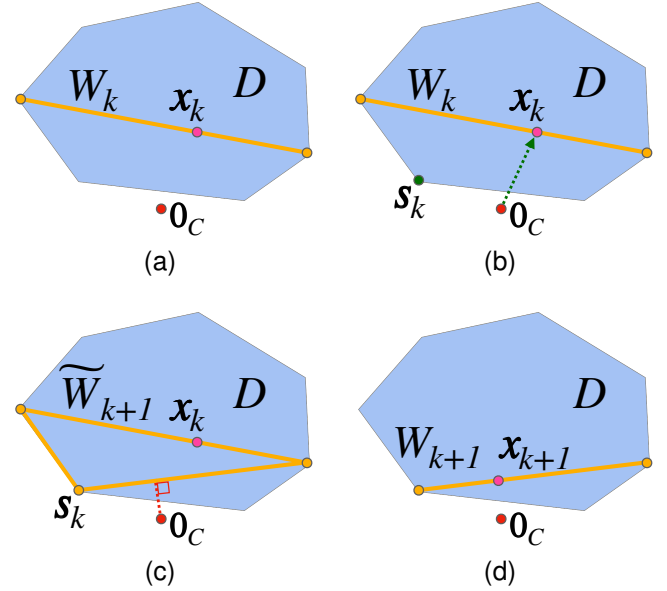


Fig. 5. Illustration of the GJK simplex strategy in 2D: (a) beginning of the k^{th} iteration, (b) support point computation, (c) simplex augmentation, (d) simplex update.

- 1) If $\mathbf{x}_{k+1} = \mathbf{0}_C$, the algorithm is stopped. Thus, we have $\mathbf{x}^* = \mathbf{0}_C$ and $d_{1,2} = 0$ in (4) (line 1 in Alg. 4).
- 2) 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 \mathbf{x}_{k+1} as a convex combination (line 2 in Alg. 4). Indeed, as $\mathbf{0}_C \notin \widetilde{W}_{k+1}$, the projection \mathbf{x}_{k+1} of $\mathbf{0}_C$ 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 $r_{k+1} < \widetilde{r}_{k+1} \leq n + 1$. For example, Fig. 5d shows the result of the simplex update obtained in Fig. 5c.

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 [40]. 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 Polyak and Nesterov acceleration paradigm [41].

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 [44], such as Newton methods, have faster convergence rates than GD at the price of requiring the computation and the inversion of Hessian quantities. Momentum methods have thus been introduced in the optimization literature to provide gradient-based methods with improved convergence rates without requiring

¹The efficient projection onto simplexes in \mathbb{R}^3 , named the *distance sub-algorithm* by [12], is thoroughly covered in [7], [21] and its robustness is improved in [30].

Algorithm 4 Fully-corrective FW using simplexes, applied to the distance computation problem (4). This algorithm is identical to GJK [12]

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

1: **If** $\mathbf{x}_{k+1} = \mathbf{0}_c$, **return** 0

If the algorithm has not terminated, update \widetilde{W}_{k+1} to retain only the smallest number of vectors needed to express \mathbf{x}_{k+1} :

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

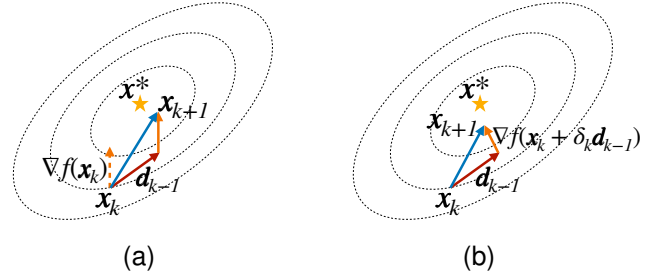


Fig. 6. (a) Polyak and (b) Nesterov acceleration schemes for unconstrained gradient descent. The gradient descent algorithm aims at finding the optimum \mathbf{x}^* by following the slope given by the gradient of function f , ∇f . The vector \mathbf{d}_{k-1} is the momentum accumulated over the optimization trajectory. The two schemes differ in where the gradient is computed at iteration k ; the Nesterov scheme introduces an intermediary point $\mathbf{y}_k = \mathbf{x}_k + \delta_k \mathbf{d}_{k-1}$ to compute the gradient.

1 costly Hessian evaluation. In this section, we use recent
2 work linking the *Polyak* and *Nesterov* accelerations of GD
3 to the FW algorithm [13], [14] to globally accelerate collision
4 detection. These global accelerations of collision detection are
5 experimentally evaluated in Sec. IV on several benchmarks.

6 *A. Background on acceleration methods in convex optimization*

8 **Polyak acceleration for unconstrained optimization.** We
9 initially consider the following *unconstrained* minimization
10 problem:

$$f(\mathbf{x}^*) = \min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}), \quad (22)$$

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

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \nabla f(\mathbf{x}_k), \quad (23)$$

16 where $\mathbf{x}_k \in \mathbb{R}^n$ is the current iterate and $\alpha_k \in \mathbb{R}$ is
17 the gradient step. This standard setting leads to a simple
18 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 [35], by relying on relatively cheap gradient evaluations. Among these gradient-descent acceleration techniques, the Polyak (or Heavy-Ball) [49] and Nesterov acceleration [41] are two of the better-studied and most popular in practice [35]. These techniques are based on accumulating previously computed gradients in a *momentum* term \mathbf{d}_k and using this momentum \mathbf{d}_k to update the current iterate \mathbf{x}_k . The Polyak update scheme for unconstrained gradient descent is illustrated in Fig. 6a and goes as follows:

$$\mathbf{d}_k = \delta_k \mathbf{d}_{k-1} + \alpha_k \nabla f(\mathbf{x}_k) \quad (24a)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k, \quad (24b)$$

19 where schemes $\delta_k \in \mathbb{R}$ is the momentum parameter. The
20 role of momentum \mathbf{d}_k is to smooth the trajectory of iterates
21 converging towards the optimum by geometrically averaging
22 previously computed gradients. The δ_k momentum parameter
23 is selected to prevent damping or overshooting of the iterate
24 trajectory when going towards the optimal solution \mathbf{x}^* .

Nesterov acceleration for unconstrained optimization. The Nesterov update scheme is the second most well-known method for accelerating unconstrained gradient descent and it is only a slight modification on top of the Polyak scheme. Contrary to the Polyak case, in the Nesterov acceleration scheme the current iterate \mathbf{x}_k is extrapolated using the momentum term \mathbf{d}_k to compute the intermediate vector $\mathbf{y}_k = \mathbf{x}_k + \delta_k \mathbf{d}_k$. The gradient is then computed at the vector \mathbf{y}_k . The Nesterov update scheme for unconstrained gradient descent is illustrated in Fig. 6b and goes as:

$$\mathbf{y}_k = \mathbf{x}_k + \delta_k \mathbf{d}_{k-1} \quad (25a)$$

$$\mathbf{d}_k = \delta_k \mathbf{d}_{k-1} + \alpha_k \nabla f(\mathbf{y}_k) \quad (25b)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{d}_k \quad (25c)$$

where δ_k is the momentum parameter as in the Polyak scheme, and $\mathbf{y}_k \in \mathbb{R}^n$ is an intermediate quantity. Computing the term \mathbf{y}_k leads to an anticipatory behavior in similar spirit to extra-gradient methods [35].

Accelerating the Frank-Wolfe algorithm with Polyak and Nesterov. Recent works of [13], [14] have proposed to adapt the Polyak and Nesterov accelerations 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 , \mathbf{s}_k , is computed in the direction of the gradient $\nabla f(\mathbf{x}_k)$ (line 1 in Alg. 1). In the Polyak acceleration of FW proposed by [13], the direction of support for computing \mathbf{s}_k is instead defined by:

$$\mathbf{d}_k = \delta_k \mathbf{d}_{k-1} + (1 - \delta_k) \nabla f(\mathbf{x}_k) \quad (26a)$$

$$\mathbf{s}_k = S_{\mathcal{D}}(\mathbf{d}_k), \quad (26b)$$

where $\delta_k = \frac{k+1}{k+3} \in [0, 1]$ is the momentum parameter and $S_{\mathcal{D}}$ is the support function as defined in (8). In the Nesterov acceleration of FW proposed by [14], the direction of support for computing \mathbf{s}_k is slightly different from the Polyak scheme (26) as it introduces \mathbf{y}_k , an intermediary vector as in the GD Nesterov scheme (25) in order to evaluate the

Algorithm 5 Polyak-accelerated and Nesterov-accelerated Frank-Wolfe [13], [14]

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

$$\begin{aligned} 1: \mathbf{y}_k &= \begin{cases} \mathbf{x}_k & \text{Polyak} \\ \delta_k \mathbf{x}_k + (1 - \delta_k) \mathbf{s}_{k-1} & \text{Nesterov} \end{cases} \\ 2: \mathbf{d}_k &= \delta_k \mathbf{d}_{k-1} + (1 - \delta_k) \nabla f(\mathbf{y}_k) \end{aligned}$$

gradient $\nabla f(\mathbf{y}_k)$:

$$\mathbf{y}_k = \delta_k \mathbf{x}_k + (1 - \delta_k) \mathbf{s}_{k-1} \quad (27a)$$

$$\mathbf{d}_k = \delta_k \mathbf{d}_{k-1} + (1 - \delta_k) \nabla f(\mathbf{y}_k) \quad (27b)$$

$$\mathbf{s}_k = S_{\mathcal{D}}(\mathbf{d}_k), \quad (27c)$$

1 where \mathbf{s}_{k-1} is the support vector computed at the previous
2 iteration. To ensure \mathbf{y}_k stays in \mathcal{D} , it is a convex combination
3 of \mathbf{x}_k and \mathbf{s}_{k-1} , both vectors of \mathcal{D} . The direction of support is
4 then obtained by taking a convex combination of the previous
5 support direction \mathbf{d}_{k-1} and the gradient $\nabla f(\mathbf{y}_k)$. Both the
6 Polyak and Nesterov accelerations of Frank-Wolfe are summed
7 up in Alg. 5.

8 The works [13], [14] have experimentally shown that these
9 accelerations strategies lead to a better convergence rate of the
10 FW algorithm when compared to the original FW algorithm.
11 In the following, we explain how to adapt the Polyak and
12 Nesterov accelerations of FW to collision detection.

13 *B. Acceleration of collision detection and distance computa-*
14 *tion*

Adapting Nesterov and Polyak 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 (26) and (27) accordingly as:

$$\mathbf{y}_k = \begin{cases} \mathbf{x}_k & \text{if Polyak} \\ \delta_k \mathbf{x}_k + (1 - \delta_k) \mathbf{s}_{k-1} & \text{if Nesterov} \end{cases} \quad (28a)$$

$$\mathbf{d}_k = \delta_k \mathbf{d}_{k-1} + (1 - \delta_k) \nabla f(\mathbf{y}_k) \quad (28b)$$

$$\mathbf{s}_k = S_{\mathcal{D}}(\mathbf{d}_k), \quad (28c)$$

$$\widetilde{W}_{k+1} = W_k \cup \{\mathbf{s}_k\}, \quad (28d)$$

$$\mathbf{x}_{k+1} = \text{proj}_{\text{conv}(\widetilde{W}_{k+1})}(\mathbf{0}_{\mathcal{C}}). \quad (28e)$$

These steps are also summarized in Alg. 6. 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. 16) can no longer be used as a convergence criterion. Indeed, the following inequality:

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

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

Algorithm 6 Polyak and Nesterov-accelerated GJK

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

For $k=0, 1, \dots$ do

$$1: \delta_k = \frac{k+1}{k+3} \quad \triangleright \text{Momentum parameter value}$$

$$2: \mathbf{y}_k = \begin{cases} \mathbf{x}_k & \text{Polyak} \\ \delta_k \mathbf{x}_k + (1 - \delta_k) \mathbf{s}_{k-1} & \text{Nesterov} \end{cases} \quad \triangleright \text{Intermediary point (28a)}$$

$$3: \mathbf{d}_k = \delta_k \mathbf{d}_{k-1} + (1 - \delta_k) \nabla f(\mathbf{y}_k) \quad \triangleright \text{Support dir. (28b)}$$

$$4: \mathbf{s}_k \in S_{\mathcal{D}}(\mathbf{d}_k) \quad \triangleright \text{Support (8)}$$

$$5: \text{if } g(\mathbf{x}_k) \leq \epsilon \text{ then} \quad \triangleright \text{Fixed-point condition (32)}$$

$$6: \quad \text{If } \mathbf{d}_k = \mathbf{x}_k, \text{ return } f(\mathbf{x}_k) \quad \triangleright \text{Algorithm terminates}$$

$$7: \quad \mathbf{s}_k \in S_{\mathcal{D}}(\nabla f(\mathbf{x}_k)) \quad \triangleright \text{Compute } \mathbf{s}_k \text{ in dir. } \nabla f(\mathbf{x}_k)$$

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

$$8: \widetilde{W}_{k+1} = W_k \cup \{\mathbf{s}_k\} \quad \triangleright \text{Augment active-set}$$

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

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

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

Stopping criterion. As the number of iteration k increases, $\delta_k \rightarrow 1$ in (28). Therefore, \mathbf{d}_k tends to be equal to \mathbf{d}_{k-1} (28b) and thus $\mathbf{s}_k = \mathbf{s}_{k-1}$ (28c). As a consequence, augmenting W_k with \mathbf{s}_k to construct \widetilde{W}_{k+1} (see (28d)) and then projecting $\mathbf{0}_{\mathcal{C}}$ onto \widetilde{W}_{k+1} (28e) will not result in any progress. Therefore, $\mathbf{x}_{k+1} = \mathbf{x}_k$: the algorithm reaches a fixed point and is stuck on constant support direction \mathbf{d} .

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

$$\forall \mathbf{s}^i \in W_k, \langle \mathbf{x}_k, \mathbf{x}_k - \mathbf{s}^i \rangle = 0. \quad (29)$$

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

$$\langle \mathbf{x}_k, \mathbf{x}_k - \mathbf{s}_k \rangle \neq 0, \quad (30)$$

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

$$\langle \mathbf{x}_k, \mathbf{x}_k - \mathbf{s}_k \rangle = 0, \quad (31)$$

then \mathbf{s}_k is a linear combination of vectors in W_k . Adding \mathbf{s}_k to W_k will thus not result in any progress towards the optimum. As a consequence, Eq. (31) encompasses two cases:

- If the support direction \mathbf{d}_k is aligned with $\nabla f(\mathbf{x}_k)$, Eq. (31), corresponding to $g_{\text{FW}}(\mathbf{x}_k) = 0$, matches the termination criterion of the distance computation problem and therefore we have reached the optimum i.e $\mathbf{x}_k = \mathbf{x}^*$.
- Otherwise, if \mathbf{d}_k is not aligned with $\nabla f(\mathbf{x}_k)$, the algorithm cannot stop as a null duality gap is not met. The algorithm thus enters a cycle where it iterates until Eq. (31) does not hold. To cope with this undesired behavior we simply stop the Polyak or the Nesterov acceleration as soon as Eq. (31) is met and switch back to the non-accelerated version Alg. 4.

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

$$g(\mathbf{x}_k) = 2\langle \mathbf{x}_k, \mathbf{x}_k - s_k \rangle, \quad (32)$$

g is used in Alg. 6 as an optimality criterion ($g \leq \epsilon$) either for stopping the Polyak and Nesterov accelerations in order to continue with the vanilla GJK, or as stopping criteria qualifying an optimal solution, in which case $g = g_{\text{FW}}$ and (18) holds. The entire algorithm is summarized in Alg. 6.

Nesterov acceleration for non-strictly convex shapes. Let us explain the effect of the Nesterov acceleration on the support direction update (28b) and distinguish between strictly convex and non-strictly convex \mathcal{D} :

- If \mathcal{D} is strictly convex, any vector s belonging to the surface of \mathcal{D} has a unique corresponding direction \mathbf{d} such that $s = S_{\mathcal{D}}(\mathbf{d})$. Here, we stress the fact that the support function $S_{\mathcal{D}}$ returns only *one* vector. Consequently, we have $\mathbf{d}_k \neq \mathbf{d}_{k-1}$ and therefore $s_k \neq s_{k-1}$. The fixed point condition (31) is thus not met unless $\delta_k = 1$ and Nesterov acceleration continues to be applied in Alg. 6. In practice, the algorithm runs until δ_k gets close to 1 or \mathbf{x}_k gets close to $\mathbf{0}_c$. The condition (31) 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 $\{\mathbf{d}^1, \dots, \mathbf{d}^m, \dots\}$ can yield the same support vector $s \in S_{\mathcal{D}}(\mathbf{d}^1) = \dots = S_{\mathcal{D}}(\mathbf{d}^m) = \dots$ etc. Consequently, it is possible to have $\mathbf{d}_{k-1} \neq \mathbf{d}_k$ and $s_k = s_{k-1}$. Therefore, even though δ_k is not close to 1, the fixed point condition (31) 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 commonly encountered in simulation or motion planning with contacts. In (28b), this is due to the norm of $\nabla f(\mathbf{y}_k)$ being predominant over the norm of \mathbf{d}_{k-1} as k increases, $\|\mathbf{d}_{k-1}\| \ll \|\nabla f(\mathbf{y}_k)\|$. As a consequence, the Nesterov acceleration enters a cycle: the support direction \mathbf{d}_k does not change enough compared to \mathbf{d}_{k-1} , hence the support point s_k is identical to s_{k-1} and therefore the intermediary point \mathbf{y}_k does not change and the cycle repeats. As a consequence, the criterion (31) is met and the Nesterov acceleration is stopped to escape the cycle, possibly prematurely. To prevent this phenomenon observed on non-strictly convex \mathcal{D} , we propose to replace (28b) by a simple heuristic which normalizes the gradient and momentum directions as follows:

$$\mathbf{d}_k = \delta_k \frac{\mathbf{d}_{k-1}}{\|\mathbf{d}_{k-1}\|} + (1 - \delta_k) \frac{\nabla f(\mathbf{y}_k)}{\|\nabla f(\mathbf{y}_k)\|}, \quad (33)$$

summarized in Alg. 7. In Sec. IV, we experimentally prove this heuristic to significantly reduce the number of steps for distance computations for non-strictly convex shapes. We also show that this heuristic does not need to be applied to the Polyak acceleration, as, contrary to the Nesterov acceleration, the Polyak acceleration does not compute an intermediary point \mathbf{y}_k .

Algorithm 7 Normalize direction for non-strictly convex shapes in Nesterov-accelerated GJK

Replace line 3 in Alg. 6 by:

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

IV. EXPERIMENTS

In this section, we study the performance of both Polyak and Nesterov-accelerated GJK (Alg. 6) against the vanilla GJK (Alg. 4) algorithm.

In sections IV-A and IV-B, we benchmark our proposed Polyak-accelerated and Nesterov-accelerated GJK algorithms against the vanilla GJK algorithm on these two distinct benchmarks. The benchmark made of strictly-convex shapes represents a worst-case scenario regarding the number of iterations for all variants of GJK. The benchmark of non-strictly convex shapes represents shapes typically used in robotic or computer graphics applications. In Sec. IV-C, we benchmark GJK and our proposed accelerated gradients against the state-of-the-art quadratic programming solver ProxQP [50]. We show that GJK and our proposed accelerated variants vastly outperform generic quadratic programming (QP) solvers, making these QP solvers prohibitive for collision detection. Then, in Sec. IV-D, we compare our methods and our implementation of vanilla GJK against different collision detection solvers of various collision detection libraries. Finally, in Sec. IV-E, we benchmark vanilla, Polyak-accelerated, and Nesterov-accelerated GJK on a dataset of trajectories obtained using a physics simulator. We show that, similarly to vanilla GJK, our accelerated GJK algorithms can benefit from being warm-started with previous simulation time steps, outperforming the vanilla GJK in physics simulation scenarios.

Implementation. We leverage the HPP-FCL C++ library [28], [51], an extension of the original FCL library [28]. Unlike FCL, HPP-FCL provides its own implementation of GJK, which we have extended by implementing the Polyak and Nesterov-accelerated GJK algorithms (Alg. 6). The open-source code of the HPP-FCL library is publicly available at <https://github.com/humanoid-path-planner/hpp-fcl> under the BSD-3 license. The benchmark code is publicly available at <https://github.com/lmontaut/colbench> under the GNU AGP License.

Shapes datasets. 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 YCB dataset [52].

Ellipsoids. 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 shapes, using a uniform distribution for the relative rotation between the shapes. Regarding the translation part of the random poses, the directions are selected at random, but the norms are chosen so that we control the distance $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$

1 between the objects. This enables us to measure the influ-
 2 ence of the separation distance on the performance of the
 3 studied algorithms. The values used for $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ range
 4 from -0.1 m to 1 m. Negative values correspond to scenarios
 5 where the shapes intersect, with $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ corresponding
 6 to the separating vector’s norm. The separating vector is the
 7 vector of the smallest norm needed to translate one of the two
 8 shapes such that the two shapes do not intersect. Therefore, for
 9 each pair of ellipsoids, 100 random relative poses are sampled,
 10 so the shapes do not intersect. We translate the shapes along
 11 the axis given by their closest points for each relative pose to
 12 study the impact of $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$. We then set $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ to
 13 fixed values between -0.1 m to 1 m.

14 **YCB meshes.** On the other hand, the YCB mesh dataset
 15 contains about 60 shapes commonly used for robotic ma-
 16 nipulation tasks (kitchen appliances, tools, toys, etc.). Each
 17 object has three different resolution levels corresponding to
 18 the number of points representing the mesh. For each object,
 19 we take the lowest resolution, i.e., the *google-16k* versions of
 20 the meshes, as it is resolute enough for any robotic task. As
 21 GJK-like algorithms work on convex shapes, we pre-compute
 22 the convex hulls of each object in the YCB dataset. This
 23 procedure needs only to be done once; if more precision is
 24 required for a certain robotic task, it is common to decompose
 25 a non-convex object into a set of convex sub-objects. For
 26 the sake of simplicity, we will not decompose YCB objects
 27 into sub-objects, as the results presented in this section would
 28 essentially be the same. In the rest of this section, when we
 29 mention a shape, we refer to its convex hull unless explicitly
 30 stated otherwise. The resulting meshes extracted from the YCB
 31 dataset contain between 100 and 8000 vertices. About 50%
 32 of meshes contain between 100 and 1000 vertices. As in the
 33 ellipsoids benchmark, 100 random relative poses are sampled
 34 for each pair such that the shapes do not intersect and then
 35 set $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ to fixed values between -0.1 m and 1 m.

36 In both benchmarks (ellipsoids and YCB meshes), the
 37 characteristic sizes of the shapes range from a few centimeters
 38 up to a meter. Finally, for the distance computation problem,
 39 we select a convergence tolerance of $\epsilon = 10^{-8}$.

40 **Initialization strategy.** Apart from Sec. IV-E, the GJK al-
 41 gorithm and our proposed accelerated GJK algorithms are
 42 initialized with the centers of the shapes’ bounding boxes. The
 43 bounding box of a shape fully encapsulates it, as is shown in
 44 Fig. 1. Hence, if we denote c_1 and c_2 the geometric centers of
 45 the bounding boxes of \mathcal{A}_1 and \mathcal{A}_2 , then we initialize vanilla
 46 GJK, Polyak-accelerated GJK and Nesterov-accelerated GJK
 47 to $x_0 = c_1 - c_2$.

48 **Metrics.** To measure the performances of Polyak-accelerated
 49 GJK, Nesterov-accelerated GJK, and the vanilla GJK algo-
 50 rithms, we measure the number of iterations N^k to solve a
 51 given collision problem. For the mesh benchmark, we also
 52 measure the execution time T^μ of both methods. We solve
 53 each generated collision problem 100 times to cope with CPU
 54 throttling. We then report the average of the 90% lowest
 55 computation times. All the benchmarks in this paper were run
 56 on an Apple M1 Max CPU.

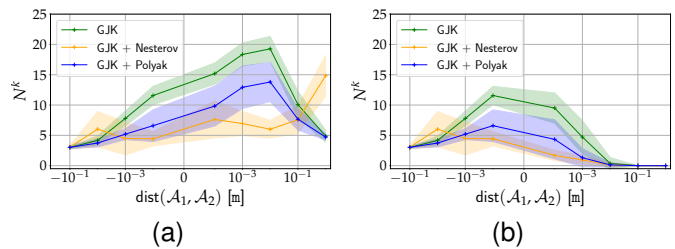


Fig. 7. Comparison of Polyak-accelerated GJK, Nesterov-accelerated GJK, and vanilla GJK on the ellipsoid benchmark for (a) distance computation and (b) Boolean collision checking. The graphs show the number of iterations (y-axis) vs. the signed distance between the two shapes (x-axis). The curve shows 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. The Polyak-accelerated GJK algorithm is more robust when shapes are strongly overlapping or distant.

A. Worst case scenario: strictly convex shapes - ellipsoids

1 We first focus on the ellipsoid benchmark to get a statistical
 2 understanding of the performance of Polyak and Nesterov-
 3 accelerated GJK against vanilla GJK. In the following, we
 4 explain why these shapes are interesting to study experimen-
 5 tally, as they represent the worst-case scenario that GJK-
 6 like algorithms can be confronted with. First, as previously
 7 explained, GJK-like algorithms look for the optimal active set
 8 of the solution x^* . Otherwise said, GJK-like methods find
 9 a set of support points $W^* = \{s^1, s^2, \dots\}$ such that the
 10 optimal solution x^* is a convex combination of the points
 11 of W^* , where s^i are support points computed while running
 12 GJK or our proposed accelerations. Then, contrary to non
 13 strictly-convex shapes, strictly-convex shapes have an infinite
 14 amount of support points. As explained at the end of Sec. III,
 15 each normalized support direction d corresponds to a unique
 16 support point $S_D(d)$. Therefore, it is fundamentally harder
 17 to identify the optimal active set when considering strictly-
 18 convex shapes, as there is an infinite amount of potential
 19 support points to consider. In contrast, there is a finite amount
 20 of support points to consider when using non strictly-convex
 21 shapes.
 22

23 In Fig. 7, we show the performance of the vanilla, the
 24 Polyak-accelerated, and the Nesterov-accelerated GJK algo-
 25 rithms on the ellipsoids benchmark. Fig. 7a and Fig. 7b
 26 show the mean and standard deviation of the number of
 27 iterations N^k of each method for the distance computation and
 28 the Boolean collision checking problems, respectively. When
 29 the shapes are shallowly intersecting, Polyak and Nesterov-
 30 accelerated GJK converge with the same or even fewer number
 31 of iterations than vanilla GJK. However, the shallower the
 32 penetration, the more Polyak and Nesterov accelerate over
 33 vanilla GJK, with Nesterov providing the most acceleration.
 34 The irregularity in standard deviation at -0.01 m is a crit-
 35 ical zone for the Nesterov momentum where the variance
 36 increases. When shapes are in close proximity, the Nesterov
 37 acceleration of GJK significantly reduces the number of
 38 iterations compared to vanilla GJK and Polyak-accelerated
 39 GJK. Finally, when shapes are distant, $1 \text{ m} \leq \text{dist}(\mathcal{A}_1, \mathcal{A}_2)$,
 40 the Nesterov acceleration is detrimental to convergence on
 41 the distance computation problem while Polyak-accelerated

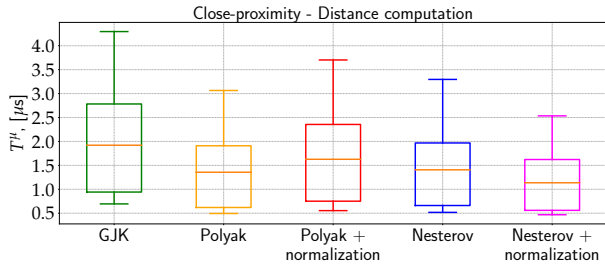


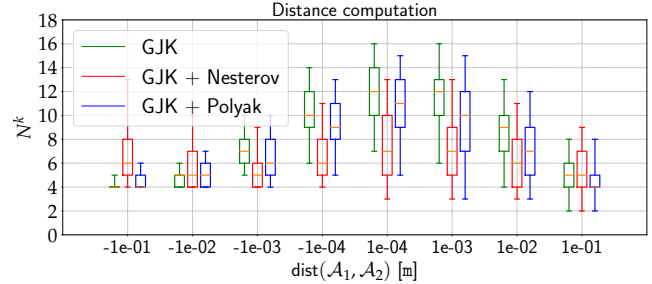
Fig. 8. Impact of support direction normalization in Polyak and Nesterov-accelerated GJK on the YCB benchmark. The graph shows the computation time T^μ (lower is better) for vanilla GJK, Polyak-accelerated GJK, and Nesterov-accelerated GJK with and without support direction normalization. Here, the two shapes are in close-proximity: $0 \text{ m} < \text{dist}(\mathcal{A}_1, \mathcal{A}_2) \leq 0.1 \text{ m}$. Normalizing the support direction benefits Nesterov-accelerated GJK, reducing the overall number of iterations compared to GJK and non-normalized Nesterov-accelerated GJK.

1 GJK remains competitive against vanilla GJK. This indicates
 2 that the Polyak acceleration is generally more robust than
 3 the Nesterov acceleration. However, it offers less acceleration
 4 over vanilla GJK when the shapes are in close-proximity
 5 or shallowly overlapping. A similar pattern of speed-ups of
 6 Polyak and Nesterov-accelerated GJK over vanilla GJK is
 7 shown for the collision detection problem in Fig. 7b.

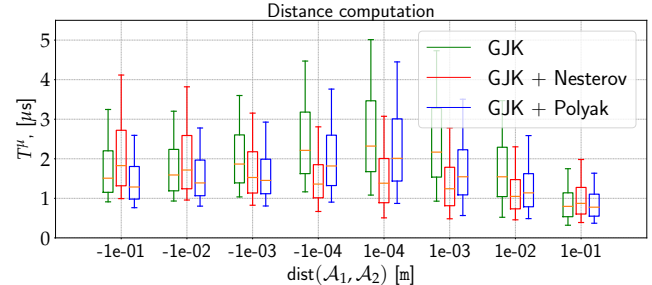
8 B. Non-strictly convex shapes: meshes

9 **Effect of support direction normalization.** For meshes, the
 10 importance of normalizing the support direction (see Eq. (33))
 11 in the Nesterov-accelerated GJK is highlighted in Fig. 8. For
 12 both the distance computation and Boolean collision checking
 13 problems, the normalization heuristic prevents the Nesterov
 14 acceleration from reaching a fixed point too early, and con-
 15 sequently, it reduces the overall amount of iterations needed
 16 to converge. This is, however, not the case for the Polyak-
 17 accelerated GJK algorithm, which does not benefit from sup-
 18 port normalization. As explained at the end of Sec. III, the
 19 Polyak acceleration does not compute an intermediary point,
 20 unlike the Nesterov acceleration scheme. In the following, we
 21 thus focus only on Polyak-accelerated GJK *without* support
 22 normalization and Nesterov-accelerated GJK *with* support
 23 normalization. We compare the performances of these two
 24 algorithms against the vanilla GJK algorithm.

25 **Statistical validation over the YCB dataset.** In Fig. 9 and
 26 Fig. 10, we report the number of iterations N^k and execution
 27 time T^μ for Polyak-accelerated GJK, Nesterov-accelerated
 28 GJK and vanilla GJK. In Fig. 11, we report relative accel-
 29 erations $T_{\text{GJK}}^\mu / T_{\text{polyak}}^\mu$ and $T_{\text{GJK}}^\mu / T_{\text{Nesterov}}^\mu$
 30 of Polyak-accelerated compared to GJK. These relative accel-
 31 erations are computed on a given collision problem, and Fig. 11
 32 reports their statistical distributions. These relative measures
 33 allow analyzing the effects of the studied algorithms on the
 34 same collision problems, which are not captured when using
 35 absolute values. Overall, Polyak and Nesterov-accelerated
 36 GJK significantly reduce the execution time when compared
 37 to GJK in cases where shapes are shallowly intersecting or
 38 in close-proximity. It is worth recalling, at this stage,
 39 that when two shapes are relatively far from each other, any
 broadphase algorithm will

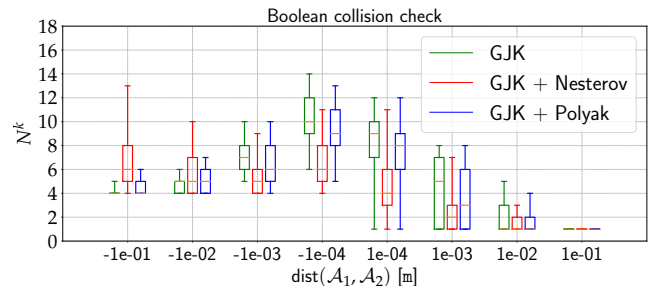


(a)

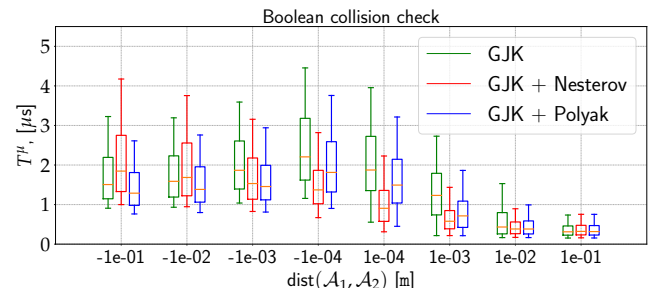


(b)

Fig. 9. Distance computation on the YCB benchmark. The graphs show the number of iterations N^k (a) and the execution time T^μ (b) for Polyak-accelerated GJK, Nesterov-accelerated GJK (with normalization) and vanilla GJK for a range of distances (x-axis) between the shapes. For both metrics, lower is better.






(a)



(b)

Fig. 10. Boolean collision check on the YCB benchmark. The graphs show the number of iterations N^k (a) and the execution time T^μ (b) for the Polyak-accelerated GJK, Nesterov-accelerated GJK (with normalization), and vanilla GJK algorithms for a range of distances (x-axis) between the shapes. For both metrics, lower is better.

TABLE I
COMPUTATION TIMES (μs) FOR DISTANCE COMPUTATION (T_D^μ) AND BOOLEAN COLLISION CHECKING (T_C^μ) ON THE YCB BENCHMARK FOR CLOSE-PROXIMITY OR SHALLOWLY INTERSECTING SHAPES. N DENOTES THE NUMBER OF VERTICES FOR EACH MESH.

		$N = 240$			$N = 1811$			$N = 3585$		
		GJK	Polyak	Nesterov	GJK	Polyak	Nesterov	GJK	Polyak	Nesterov
	T_D^μ	1.1 ± 0.3	0.9 ± 0.3	0.9 ± 0.3	1.9 ± 0.5	1.5 ± 0.5	1.4 ± 0.5	3.2 ± 0.8	2.3 ± 0.7	2.4 ± 0.7
	T_C^μ	0.9 ± 0.4	0.7 ± 0.4	0.8 ± 0.4	1.5 ± 0.6	1.2 ± 0.6	1.1 ± 0.6	2.5 ± 0.9	1.7 ± 0.8	1.9 ± 1.0
	T_D^μ				2.7 ± 0.8	2.1 ± 0.7	1.9 ± 0.6	4.0 ± 1.0	2.9 ± 0.9	2.9 ± 1.0
	T_C^μ				2.3 ± 0.8	1.6 ± 0.8	1.5 ± 0.8	3.1 ± 1.2	2.1 ± 1.0	2.2 ± 1.3
	T_D^μ							4.4 ± 1.3	2.9 ± 1.0	3.0 ± 0.9
	T_C^μ							3.0 ± 1.9	2.1 ± 1.4	2.1 ± 1.4

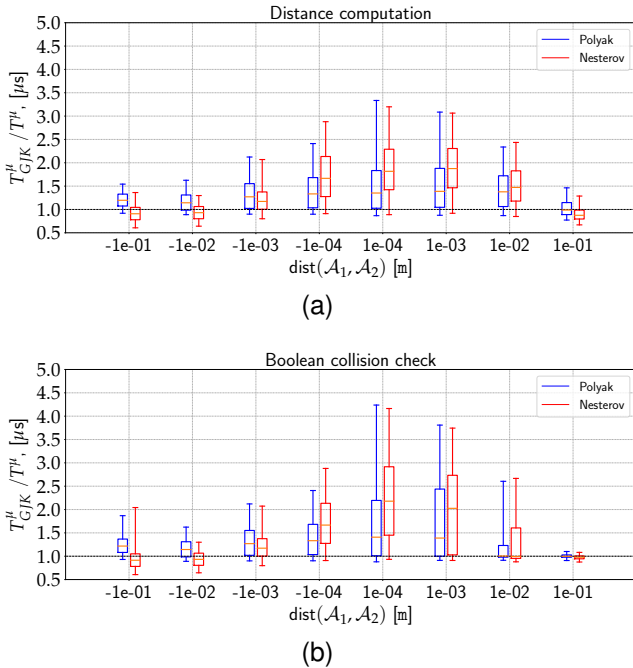


Fig. 11. Speed-ups of Polyak and Nesterov-accelerated GJK over vanilla GJK on the YCB benchmark. The plots show ratios of the number of execution times for (a) distance computation and (b) boolean collision checking of Polyak-accelerated GJK and Nesterov-accelerated GJK (with normalization) against vanilla GJK. Ratios over 1.0 show speed-ups of accelerated GJK over vanilla GJK. For both metrics, higher is better.

more robust than Nesterov-accelerated GJK as it is almost always better than vanilla GJK, even when the shapes are distant or overlap.

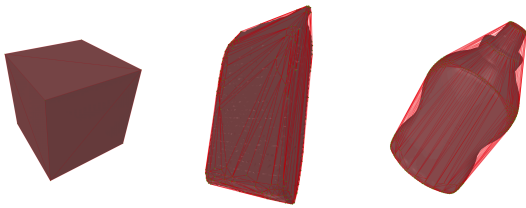
In Table. I, we select three meshes with an increasing number of vertices to highlight the benefits of the Polyak and Nesterov accelerations. For each pair, we report the mean and the standard deviation of the execution time for distance computation and Boolean collision checking. We consider the challenging set-up of close-by or shallowly intersecting shapes in the range of separation distances $-0.01 \text{ m} \leq \text{dist}(\mathcal{A}_1, \mathcal{A}_2) \leq 0.01 \text{ m}$. The lower mean and standard deviation show that Polyak and Nesterov-accelerated GJK are faster than the vanilla GJK and reduce the spread of computation times across the different collision problems in this setting.

From this benchmark involving shapes from the YCB dataset, we can distinguish two use cases in which one would prefer using Polyak-accelerated GJK compared to Nesterov-accelerated and vice-versa. In tasks where the exact distance between the shapes needs to be computed and where this distance separating the shapes can take any value, due to its robustness, the Polyak-accelerated GJK algorithm is better suited than its Nesterov counterpart. However, in a situation involving shapes interacting at close proximity, like in a contact physics simulation, it is preferable to choose the Nesterov-accelerated GJK. Before studying the performance of GJK and our proposed accelerations for physics simulation, we first show the benefits of using GJK-based algorithms for collision detection instead of standard off-the-shelf optimization solvers and provide a comparison of the implementations of our methods against baselines from other collision detection libraries.

1 automatically discard such a pair. Only in a small percentage
 2 of cases, Polyak-accelerated GJK and Nesterov-accelerated
 3 GJK are slower than GJK. When measuring the absolute
 4 performance of the two proposed methods, Polyak-accelerated
 5 GJK provides less acceleration than Nesterov-accelerated GJK
 6 in critical cases with close proximity and shallowly overlap-
 7 ping collision problems. However, Polyak-accelerated GJK is

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32

TABLE II
COMPUTATION TIME IN MICRO-SECONDS OF GJK-LIKE SOLVERS VS.
SOTA QUADRATIC PROGRAMMING PROXQP SOLVER.



	$N_v = 8$ $N_f = 6$	$N_v = 250$ $N_f = 496$	$N_v = 940$ $N_f = 1876$
ProxQP	$5.3 \pm 2.7 \mu\text{s}$	$(2 \pm 0.6) \cdot 10^3 \mu\text{s}$	$(20 \pm 14) \cdot 10^3 \mu\text{s}$
GJK	$0.2 \pm 0.03 \mu\text{s}$	$0.8 \pm 0.3 \mu\text{s}$	$2.1 \pm 0.5 \mu\text{s}$
Nesterov	$0.2 \pm 0.05 \mu\text{s}$	$0.7 \pm 0.2 \mu\text{s}$	$1.4 \pm 0.3 \mu\text{s}$
Polyak	$0.2 \pm 0.05 \mu\text{s}$	$0.6 \pm 0.2 \mu\text{s}$	$1.4 \pm 0.4 \mu\text{s}$

1 C. GJK-like algorithms vs. generic quadratic programming 2 solvers

3 As explained in Sec. II, in the case of two convex meshes,
4 the collision problem can be formulated as a Quadratic Pro-
5 gram (2) (QP), which can be solved using any generic QP
6 solver [50], [53]–[56]. In Table II, we compare the perfor-
7 mance of GJK and our proposed accelerations against the
8 state-of-the-art ProxQP solver [50]. We report the computation
9 timings in micro-seconds for pairs of identical shapes with
10 an increasing number of vertices (N_v) and faces (N_f). The
11 results are staggering: for very simple convex meshes like
12 a cube, GJK, and its accelerated variants are already more
13 than 10 times faster than the QP solver. When the complexity
14 of the meshes increases, GJK and its variants are thousands to
15 tens of thousands of times faster than the QP solver, making
16 generic QP solvers prohibitive for collision detection in real-
17 time applications like robotics or computer graphics. Although
18 these results are not surprising, they clearly showcase why
19 dedicated solvers such as GJK-like methods are crucial for
20 collision detection.

21 D. Comparison against other collision detection libraries

22 In this sub-section, we compare our implementations of
23 vanilla, Nesterov-accelerated and Polyak-accelerated GJK
24 against the following baselines from other collision detection
25 libraries: CCD’s MPR and GJK implementations [15], FCL’s
26 GJK implementation [16] and Bullet’s GJK implementation.
27 These collision detection algorithms are used in physics simu-
28 lators like Drake [18], MuJoCo [3], ODE [19] and Bullet [1].
29 Since GJK is the state-of-the-art algorithm for narrow phase
30 collision detection, it is no surprise that most collision de-
31 tection libraries implement only this algorithm. In addition
32 to being a general, robust and computationally fast method,
33 GJK-like algorithms also have the advantage of producing a
34 simplex surrounding the origin when shapes are in collision
35 (see Sec. II). This simplex is then fed to the Expanding
36 Polytope Algorithm (EPA) in order to estimate the penetration
37 depth and separation vector [22]; these contact informations
38 are used in physics simulation to resolve contact constraints.
39 Just like GJK, the core of EPA consist in computing support
40 points in order to expand a polytope inside the Minkowski

1 difference. Therefore, since GJK and EPA are made to work
2 in succession with one another, it is almost always the case
3 that physics simulators use GJK to first detect collisions and
4 then EPA to compute contact features if a collision is detected.
5 All the libraries listed before are written either in C or in C++.

6 In Fig. 12, we report the performance of the methods
7 listed before on the YCB benchmark used in Sec. IV-B.
8 We measure the execution time of the different collision
9 detection solvers on the boolean collision check task and
10 divide the results into three distance categories: when shapes
11 are distant ($\text{dist}(\mathcal{A}_1, \mathcal{A}_2) \geq 1\text{m}$), when shapes are in close-
12 proximity ($0\text{m} \leq \text{dist}(\mathcal{A}_1, \mathcal{A}_2) \leq 0.1\text{m}$) and when shapes
13 are overlapping ($\text{dist}(\mathcal{A}_1, \mathcal{A}_2) \leq 0\text{m}$). The collision problems
14 corresponding to the *distant* case would typically be filtered
15 by the broadphase. We report them to give a clear picture of
16 the performance of the different implementations. In practice,
17 the *close-proximity* and *overlapping* cases correspond to situa-
18 tions when the different collision detection solvers are actually
19 called; the broad phase cannot filter such collision problems
20 and the narrow phase is then called.

21 The results show our implementation of GJK and our
22 proposed methods outperform the solvers of the other collision
23 detection libraries. We find similar results for the distance
24 computation task. We now turn our attention to the context of
25 physics simulation and show that our proposed methods, just
26 like GJK, can be warm-started by using previous simulation
27 steps.

28 E. Collision detection for physics simulation

29 In the previous benchmarks, we have experimentally shown
30 the improvement of our methods, Polyak-accelerated GJK and
31 Nesterov-accelerated GJK, over the vanilla GJK algorithm for
32 collision problems which are important in practice, *i.e.* when
33 the broadphase has not filtered collision pairs and are thus
34 overlapping or in close proximity. So far, the benchmarks
35 have been constructed by randomly selecting poses for our
36 shapes. However, in robotics applications such as trajectory
37 optimization, motion planning, or computer graphics, the
38 successive poses between objects are usually correlated by
39 time. In this sub-section, we study how vanilla GJK, Polyak-
40 accelerated GJK, and Nesterov-accelerated GJK can be warm-
41 started using the previous time instant, as occurring inside
42 physics simulators.

43 To do so, we create a dataset of trajectories using pairs of
44 objects from the YCB dataset used in Sec. IV-B. We randomly
45 select 1000 pairs of YCB objects and drop them in a funnel
46 as shown in Fig. 14. At the beginning of the simulation, each
47 object is given a random pose and random translational and
48 rotational velocities. The simulation is then run at 120Hz for 1
49 second. When a collision occurs, the GJK and EPA (expanding
50 polytope algorithm) algorithms are called to determine the
51 position of the contact points and the corresponding normal
52 for the considered pair of objects. The collision is then
53 resolved using a contact solver based on the Projected Gauss-
54 Seidel [57] algorithm to account for a second-order cone
55 representing friction, following the implementation proposed
56 in [58]. In total, 120k collision problems are generated. For

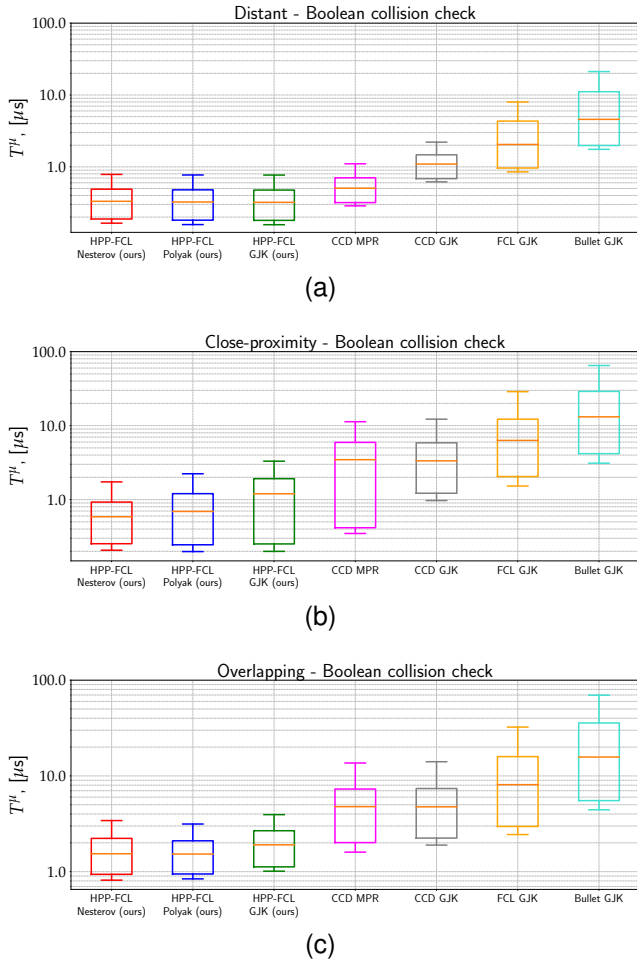


Fig. 12. Computation time in micro-seconds of implementations of different collision detection solvers from various C/C++ libraries on the YCB benchmark for boolean collision checking. The y-axis is a log-scale. We compare our implementation of vanilla GJK, Nesterov-accelerated GJK and Polyak-accelerated GJK against FCL’s GJK implementation, CCD’s GJK implementation and CCD’s MPR implementation. The results are split into three different categories: (a) when shapes are distant ($\text{dist}(\mathcal{A}_1, \mathcal{A}_2) \geq 1\text{m}$), (b) in close-proximity ($0 \leq \text{dist}(\mathcal{A}_1, \mathcal{A}_2) \leq 0.1\text{m}$) and (c) overlapping ($\text{dist}(\mathcal{A}_1, \mathcal{A}_2) \leq 0\text{m}$). Lower is better.

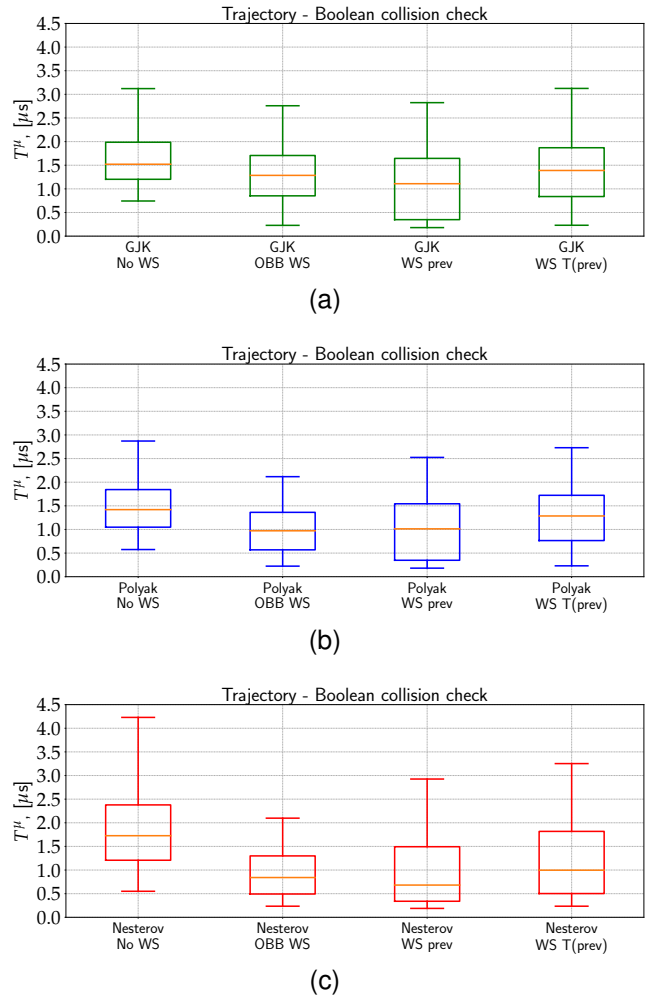


Fig. 13. Boolean collision checking of YCB objects’ trajectories (see Fig. 14) for different warm-start strategies for (a) vanilla GJK, (b) Polyak-accelerated GJK, and (c) Nesterov-accelerated GJK (with normalization). In the three figures, *WS* is an abbreviation of *warm-start*. The *No WS* strategy signifies the algorithm is initialized with $\mathbf{x}_0 = (1, 0, 0)^T$. The *OBB WS* strategy uses the objects’ current OBBs centers to compute \mathbf{x}_0 . In both *WS prev* and *WS T(prev)*, \mathbf{x}_0 is computed using GJK or EPA’s previous solution, when this solution is available (i.e., when the previous collision problem was not discarded by the broadphase). Contrary to *WS prev*, *WS T(prev)* corrects the previous solution using the relative displacement of the shapes between the two considered time steps.

1 each collision problem, we extract the YCB shapes and their
2 poses.

3 This dataset allows us to evaluate the vanilla, Polyak-
4 accelerated, and Nesterov-accelerated GJK algorithms on the
5 same collision problems generated by a physics simulation.
6 Interestingly, this dataset allows us to study only the collision
7 problems not filtered by the broadphase of the physics simula-
8 tor, as explained in Sec. I. During the broad phase, the oriented
9 bounding boxes of the objects (OBBs, as shown in Fig. 1) are
10 used to assess if objects are not in collision. Therefore, if the
11 broad phase does not filter a collision, the GJK algorithm and
12 our proposed accelerations are called and solve the boolean
13 collision check problem. Finally, this dataset allows us to test
14 different strategies to warm-start (WS) the GJK algorithm and
15 our proposed accelerations. We denote by \mathbf{x}_0^t the initial guess
16 given to vanilla, Polyak-accelerated and Nesterov-accelerated
17 GJK at time step t of the simulation. We also denote by \mathbf{x}^{t-1}
18 the separation vector found by GJK (accelerated or not) or
19 EPA at time-step $t - 1$ of the simulation. We consider four

different warm-start strategies for the vanilla GJK algorithm
and our proposed accelerations:

- 1) the first strategy is the *No WS* strategy, where the vanilla, Polyak, and Nesterov GJK algorithms are initialized using $\mathbf{x}_0^t = (1, 0, 0)^T$. This strategy serves as a baseline for the other warm-start strategies.
- 2) The second strategy is the *OBB WS* strategy, where $\mathbf{x}_0^t = \mathbf{c}_1^t - \mathbf{c}_2^t$ with \mathbf{c}_1^t and \mathbf{c}_2^t being the centers of the considered objects’ oriented bounding boxes. This warm-start is used in all the previous benchmarks, as explained at the beginning of this section.
- 3) The third strategy is the *WS prev* strategy, where $\mathbf{x}_0 = \mathbf{x}^{t-1}$ is initialized using the solution found by GJK or EPA in the previous simulation time step.
- 4) The fourth and last strategy is the *WS T(prev)* strategy. The difference with the *WS prev* strategy is that we use

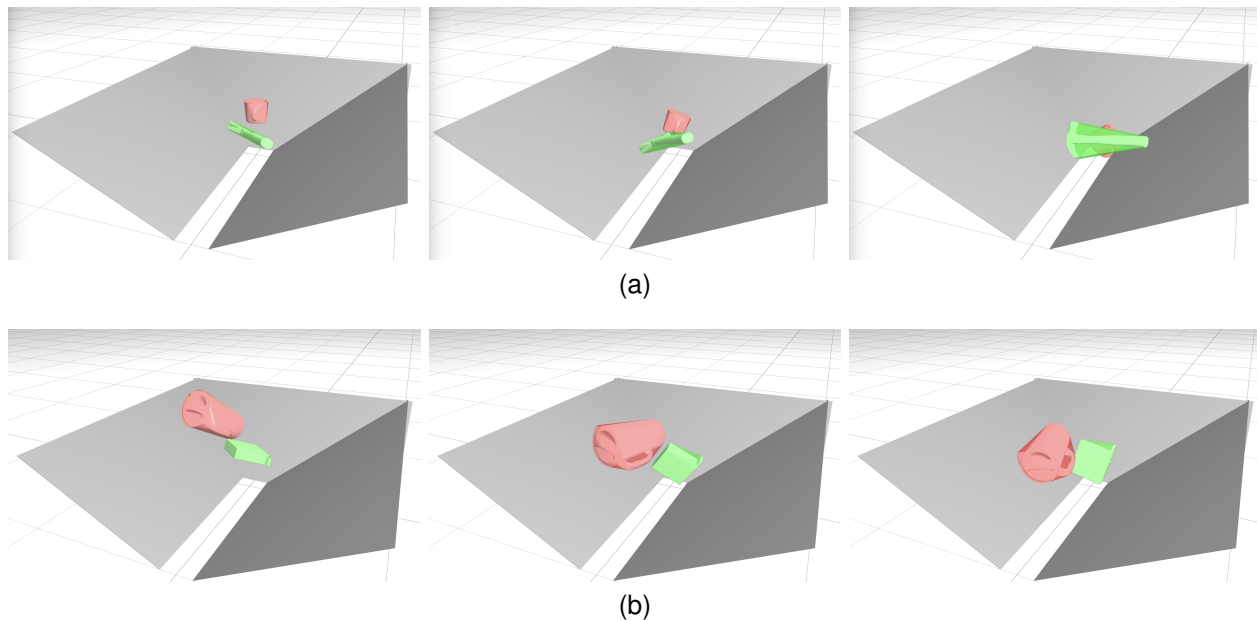


Fig. 14. Two different trajectories (a) and (b) with two different pairs of objects from the YCB dataset. The objects are dropped with a random initial velocity for each trajectory in a funnel (the grey walls). At each time step, if the broadphase cannot discriminate if the shapes are in collision or not, we use the vanilla GJK algorithm or our proposed Polyak and Nesterov-accelerated variants of GJK to determine if a collision occurs between the convex-hulls of the collision pair.

the relative transformation of the shapes between time steps t and $t - 1$ to anticipate how \mathbf{x}^{t-1} might move between these two time steps.

The last two warm-starting strategies might not always be actionable. Indeed, if at time step $t - 1$ the broad phase finds no collision between the two considered shapes, the GJK and EPA algorithms are not called, and therefore, \mathbf{x}^{t-1} does not exist. Consequently, if GJK needs to be called at time step t , it cannot use \mathbf{x}^{t-1} . In such a case, these two strategies fall back to the second strategy, which exploits the objects' OBBs.

We run vanilla, Polyak-accelerated and Nesterov-accelerated GJK on the dataset of trajectories described previously; the results of this benchmark are summed up in Fig. 13. In this figure, we report the computation time of the boolean collision check for GJK and our proposed accelerations. Importantly, this figure only considers the collision problems which were not filtered by the broad phase, as GJK or its accelerations would not be called otherwise. In doing so, we aim to provide the clearest possible picture of the computation time dedicated to GJK in a physics computation. Due to the filtering of the broad phase, the typical distance separating the shapes is less than a few centimeters; this corresponds to the overlapping and close-proximity cases described in the previous benchmarks. First, the results show that for the three studied methods, the *No WS* and *WS T(prev)* warm-start strategies provided a worse initial guess than the two other warm-start strategies. It appears that the *WS T(prev)* strategy is often the worse strategy; this observation means that the separation vector computed by GJK and/or EPA moves in a non-trivial manner between time steps $t - 1$ and t of the simulation. For vanilla GJK, the best warm-starting strategy is the *WS prev* strategy, which re-uses the separation vector computed by GJK and EPA at

time step $t - 1$ of the simulation. For Polyak-accelerated GJK, both the *OBB WS* and *WS prev* strategies perform better than vanilla GJK's best warm-starting strategy. However, contrary to GJK, the *OBB WS* strategy is arguably better than the *WS prev* strategy as it greatly reduces the variance of the computation timings distribution. For Nesterov-accelerated GJK, the results are even more significant: both the *OBB WS* and *WS prev* strategy significantly outperform GJK with its best warm-starting strategy. When using the *OBB WS* and *WS prev* strategies, the Nesterov acceleration allows the median of computation times to reach close to $0.5\mu\text{s}$, compared to a median above $1\mu\text{s}$ in the case of GJK's best warm-starting strategy. Like the Polyak acceleration, the Nesterov-accelerated GJK algorithm significantly reduces the spread of the distribution of computation times compared to GJK. This is especially visible when using the *OBB WS* strategy together with the Nesterov acceleration. Finally, this benchmark shows that physics simulation strongly benefits from using Nesterov-accelerated GJK warm-started using the *OBB WS* strategy.

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 adapting recent contributions on applying Polyak and 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 intersect or are close, accelerating collision detection by up to a factor of two. 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 accelerations correspond to improvements of GJK's execution time on the order of a few microseconds, modern robotics applications may solve millions to billions of collision problems when, for instance, learning a policy with RL [59].

The Polyak and Nesterov accelerations for GJK are already included in the HPP-FCL library [51], notably used by the HPP framework [4] for motion planning, the Pinocchio framework [60] dedicated to simulation and modeling, the Crocodyl [61] and the OSC-2 [62] software dedicated to trajectory optimization, to name a few. In future work, we plan to leverage these accelerated collision detection algorithms in the scope of differentiable collision detection [63], differentiable simulation [64], [65] and constrained optimal control involving contact interactions [61], [66], [67].

Finally, 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

We warmly thank Francis Bach, Adrien Escande, Joseph Mirabel, and Mehdi Benaïme for fruitful discussions on the various topics covered by this article. We also warmly thank the cohort of developers who contribute to developing open-source, useful, reproducible, and extensible software, which primarily benefits this project and, more widely, the robotics ecosystem.

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 the management of Agence Nationale de la Recherche as part of the "Investissements d'avenir" program, reference ANR-19-P3IA-0001 (PRAIRIE 3IA Institute), by the AGIMUS project, funded by the European Union under GA no.101070165 - views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission, neither the European Union nor the European Commission can be held responsible for them - and by the Louis Vuitton ENS Chair on Artificial Intelligence.

REFERENCES

- [1] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," 2016.
- [2] Nvidia, "Persistent contact manifold," 2008.
- [3] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ international conference on intelligent robots and systems*. IEEE, 2012, pp. 5026–5033.
- [4] J. Mirabel, S. Tonneau, P. Fernbach, A.-K. Seppälä, M. Campana, N. Mansard, and F. Lamiraux, "Hpp: A new software for constrained motion planning," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 383–389.
- [5] M. A. Toussaint, K. R. Allen, K. A. Smith, and J. B. Tenenbaum, "Differentiable physics and stable modes for tool-use and manipulation planning," *Robotics: Science and Systems*, 2018.
- [6] J. Schulman, Y. Duan, J. Ho, A. Lee, I. Awwal, H. Bradlow, J. Pan, S. Patil, K. Goldberg, and P. Abbeel, "Motion planning with sequential convex optimization and convex collision checking," *The International Journal of Robotics Research*, vol. 33, no. 9, pp. 1251–1270, 2014.
- [7] C. Ericson, *Real-time collision detection*. Crc Press, 2004.
- [8] K. Mamou and F. Ghorbel, "A simple and efficient approach for 3d mesh approximate convex decomposition," in *2009 16th IEEE international conference on image processing (ICIP)*. IEEE, 2009, pp. 3501–3504.
- [9] 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*, vol. 1, no. 1, pp. 21–30, 1985.
- [10] O. Stasse, A. Escande, N. Mansard, S. Miossec, P. Evrard, and A. Kheddar, "Real-time (self)-collision avoidance task on a hrp-2 humanoid robot," in *2008 IEEE international conference on robotics and automation*. IEEE, 2008, pp. 3200–3205.
- [11] M. Frank and P. Wolfe, "An algorithm for quadratic programming," *Naval research logistics quarterly*, vol. 3, no. 1-2, pp. 95–110, 1956.
- [12] 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*, vol. 4, no. 2, pp. 193–203, 1988.
- [13] B. Li, A. Sadeghi, and G. Giannakis, "Heavy ball momentum for conditional gradient," *Advances in Neural Information Processing Systems*, vol. 34, pp. 21 244–21 255, 2021.
- [14] B. Li, M. Coutino, G. B. Giannakis, and G. Leus, "A momentum-guided frank-wolfe algorithm," *IEEE Transactions on Signal Processing*, vol. 69, pp. 3597–3611, 2021.
- [15] D. Fiser, "Github: libccd," <https://github.com/danfis/libccd>, 2010.
- [16] J. Pan, S. Chitta, and D. Manocha, "Github: Fcl – the flexible collision library," 2012.
- [17] E. Coumans, "Github: Bullet physics sdk," <https://github.com/bulletphysics/bullet3>, 2015.
- [18] R. Tedrake and the Drake Development Team, "Drake: Model-based design and verification for robotics," 2019. [Online]. Available: <https://drake.mit.edu>
- [19] R. L. Smith, "Github: The open dynamics engine (ODE)," <https://github.com/thomasmarsh/ODE>, 2001–2007.
- [20] L. Montaut, Q. Lidec, V. Petrík, J. Sivic, and J. Carpentier, "Collision Detection Accelerated: An Optimization Perspective," in *Proceedings of Robotics: Science and Systems*, New York City, NY, USA, June 2022.
- [21] G. v. d. Bergen, "A fast and robust gjk implementation for collision detection of convex objects," *Journal of graphics tools*, vol. 4, no. 2, pp. 7–25, 1999.
- [22] G. Van den Bergen, "Proximity Queries and Penetration Depth Computation on 3D Game Objects," in *Game Developers Conference*, 2001.
- [23] M. C. Lin and J. F. Canny, "A fast algorithm for incremental distance calculation," in *ICRA*, vol. 91, 1991, pp. 9–12.
- [24] B. Mirtich, "V-clip: Fast and robust polyhedral collision detection," *ACM Transactions On Graphics (TOG)*, vol. 17, no. 3, pp. 177–208, 1998.
- [25] S. Cameron, "A comparison of two fast algorithms for computing the distance between convex polyhedra," *IEEE transactions on Robotics and Automation*, vol. 13, no. 6, pp. 915–920, 1997.
- [26] G. Van Den Bergen, *Collision detection in interactive 3D environments*. CRC Press, 2003.
- [27] G. Snethen, "Complex collision made simple," pp. 165–178, 2008.
- [28] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," in *2012 IEEE International Conference on Robotics and Automation*. IEEE, 2012, pp. 3859–3866.
- [29] M. Macklin, "Simulation for learning and robotics: Numerical methods for contact, deformation, and identification," Ph.D. dissertation, University of Copenhagen, 2020.
- [30] M. Montanari, N. Petrinic, and E. Barbieri, "Improving the gjk algorithm for faster and more reliable distance queries between convex objects," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 3, pp. 1–17, 2017.
- [31] J. Canny, "Collision detection for moving polyhedra," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 2, pp. 200–209, 1986.
- [32] X. Qin and N. T. An, "Smoothing algorithms for computing the projection onto a minkowski sum of convex sets," *Computational Optimization and Applications*, vol. 74, no. 3, pp. 821–850, 2019.
- [33] E. G. Gilbert, "An iterative procedure for computing the minimum of a quadratic form on a convex set," *SIAM Journal on Control*, vol. 4, no. 1, pp. 61–80, 1966.
- [34] P. Wolfe, "Finding the nearest point in a polytope," *Mathematical Programming*, vol. 11, pp. 128–149, 1976.

- [35] A. d'Aspremont, D. Scieur, A. Taylor *et al.*, "Acceleration methods," *Foundations and Trends® in Optimization*, vol. 5, no. 1-2, pp. 1–245, 2021.
- [36] D. Garber and E. Hazan, "Playing non-linear games with linear oracles," in *2013 IEEE 54th annual symposium on foundations of computer science*. IEEE, 2013, pp. 420–428.
- [37] J. Guélat and P. Marcotte, "Some comments on wolfe's 'away step,'" *Mathematical Programming*, vol. 35, no. 1, pp. 110–119, 1986.
- [38] M. Jaggi, "Revisiting frank-wolfe: Projection-free sparse convex optimization," in *International conference on machine learning*. PMLR, 2013, pp. 427–435.
- [39] T. Kerdreux, A. d'Aspremont, and S. Pokutta, "Projection-free optimization on uniformly convex sets," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2021, pp. 19–27.
- [40] S. Lacoste-Julien and M. Jaggi, "On the global linear convergence of frank-wolfe optimization variants," *Advances in neural information processing systems*, vol. 28, 2015.
- [41] Y. E. Nesterov, "A method of solving a convex programming problem with convergence rate $\mathcal{O}(k^{-2})$," in *Doklady Akademii Nauk*, vol. 269, no. 3. Russian Academy of Sciences, 1983, pp. 543–547.
- [42] J. Nocedal and S. J. Wright, *Numerical optimization*. Springer, 1999.
- [43] F. Bach *et al.*, "Learning with submodular functions: A convex optimization perspective," *Foundations and Trends® in Machine Learning*, vol. 6, no. 2-3, pp. 145–373, 2013.
- [44] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.
- [45] C. A. Holloway, "An extension of the frank and wolfe method of feasible directions," *Mathematical Programming*, vol. 6, pp. 14–27, 1974.
- [46] T. Kerdreux, A. d'Aspremont, and S. Pokutta, "Restarting frank-wolfe," in *The 22nd international conference on artificial intelligence and statistics*. PMLR, 2019, pp. 1275–1283.
- [47] G. Dantzig, *Linear programming and extensions*. Princeton university press, 2016.
- [48] C. Carathéodory, "Über den variabilitätsbereich der koeffizienten von potenzreihen, die gegebene werte nicht annehmen," *Mathematische Annalen*, vol. 64, no. 1, pp. 95–115, 1907.
- [49] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *Ussr computational mathematics and mathematical physics*, vol. 4, no. 5, pp. 1–17, 1964.
- [50] A. Bambade, S. El-Kazdadi, A. Taylor, and J. Carpentier, "Prox-qp: Yet another quadratic programming solver for robotics and beyond," in *RSS 2022-Robotics: Science and Systems*, 2022.
- [51] J. Pan, S. Chitta, D. Manocha, F. Lamiroux, J. Mirabel, J. Carpentier *et al.*, "Hpp-fcl: an extension of the flexible collision library," <https://github.com/humanoid-path-planner/hpp-fcl>, 2015–2022.
- [52] B. Calli, A. Singh, A. Walsman, S. Srinivasa, P. Abbeel, and A. M. Dollar, "The ycb object and model set: Towards common benchmarks for manipulation research," in *2015 international conference on advanced robotics (ICAR)*. IEEE, 2015, pp. 510–517.
- [53] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "Osqp: An operator splitting solver for quadratic programs," *Mathematical Programming Computation*, vol. 12, no. 4, pp. 637–672, 2020.
- [54] D. Goldfarb and A. Idrani, "A numerically stable dual method for solving strictly convex quadratic programs," *Mathematical programming*, vol. 27, no. 1, pp. 1–33, 1983.
- [55] A. Wächter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical programming*, vol. 106, pp. 25–57, 2006.
- [56] K. Tracy, T. A. Howell, and Z. Manchester, "Differentiable collision detection for a set of convex primitives," 2022.
- [57] B. Brogliato, A. Ten Dam, L. Paoli, F. Génot, and M. Abadie, "Numerical simulation of finite dimensional multibody nonsmooth mechanical systems," *Appl. Mech. Rev.*, vol. 55, no. 2, pp. 107–150, 2002.
- [58] Q. L. Lidec, W. Jallet, L. Montaut, I. Laptev, C. Schmid, and J. Carpentier, "Contact models in robotics: a comparative analysis."
- [59] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas *et al.*, "Solving rubik's cube with a robot hand," *arXiv preprint arXiv:1910.07113*, 2019.
- [60] J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiroux, O. Stasse, and N. Mansard, "The pinocchio c++ library: A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives," in *2019 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 2019, pp. 614–619.
- [61] C. Mastalli, R. Budhiraja, W. Merkt, G. Saurel, B. Hammoud, M. Naveau, J. Carpentier, L. Righetti, S. Vijayakumar, and N. Mansard, "Crocodyl: An efficient and versatile framework for multi-contact optimal control," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 2536–2542.
- [62] F. Farshidian *et al.*, "Optimal control for switched systems," <https://github.com/leggedrobotics/ocs2>, 2018.
- [63] L. Montaut, Q. L. Lidec, A. Bambade, V. Petrik, J. Sivic, and J. Carpentier, "Differentiable collision detection: a randomized smoothing approach," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2022.
- [64] K. Werling, D. Omens, J. Lee, I. Exarchos, and C. K. Liu, "Fast and feature-complete differentiable physics engine for articulated rigid bodies with contact constraints," in *Robotics: Science and Systems*, 2021.
- [65] Q. Le Lidec, I. Laptev, C. Schmid, and J. Carpentier, "Differentiable rendering with perturbed optimizers," *Advances in Neural Information Processing Systems*, vol. 34, pp. 20 398–20 409, 2021.
- [66] R. Budhiraja, J. Carpentier, C. Mastalli, and N. Mansard, "Differential dynamic programming for multi-phase rigid contact dynamics," in *2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids)*. IEEE, 2018, pp. 1–9.
- [67] W. Jallet, A. Bambade, N. Mansard, and J. Carpentier, "Constrained differential dynamic programming: A primal-dual augmented lagrangian approach," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 13 371–13 378.