Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

OpenGJK for C, C# and Matlab: Reliable solutions to distance queries between convex bodies in three-dimensional space

Mattia Montanari [*], Nik Petrinic

Department of Engineering Science, University of Oxford, Parks Road, OX1 3PJ, UK

A R T I C L E   I N F O

Article history:
Received 18 April 2018
Received in revised form 3 September 2018
Accepted 17 October 2018

Keywords:
Minimum distance
Collision detection
GJK algorithm

A B S T R A C T

Implementing a reliable algorithm for computing the distance between convex bodies is an involved and time-consuming task. Common applications of these algorithms include robot path planning, image rendering and collision detection. However, each application has strict requirements in terms of either accuracy or speed, making the implementation less portable and more difficult. This paper introduces openGJK, a library for solving distance queries between convex bodies that are simply described by lists of points. OpenGJK features the fastest and most accurate version of the Gilbert–Johnson–Keerthi (GJK) algorithm published to date, is written in C, cross-platform and comes with interfaces for C# and Matlab.

Code metadata

Current code version	1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2018_38
Legal Code License	GNU General Public License (GPL) v3
Code versioning system used	git
Software code languages, tools, and services used	C, Matlab, mex
Compilation requirements, operating environments & dependencies	CMake, gcc, clang, Microsoft Visual Studio
If available Link to developer documentation/manual	http://iel.eng.ox.ac.uk/?page_id=504
Support email for questions	mattia.montanari@eng.ox.ac.uk

1. Introduction

Computing the minimum distance between convex bodies is a common problem in computer graphics and scientific computing. In practise this means evaluating the minimum Euclidean distance, $d(\Omega_P, \Omega_Q)$, between two bodies Ω_P and Ω_Q:

$$d(\Omega_P, \Omega_Q) = \min\{\|p - q\| : p \in P,\ q \in Q\} \tag{1}$$

where each body is defined by a finite set of points, respectively P and Q.

One of the fastest and most versatile procedures published in the literature for solving Eq. (1) is the Gilbert–Johnson–Keerthi (GJK) algorithm [1]. This has been successfully applied in robotics [2], computer games [3], physics [4] and engineering [5]. When compared to similar procedures, the GJK algorithm shows good performance [6] but, unlike other methods, its applicability is not limited to polytopes [7] and it can measure overlap [8].

To date, researchers have to spend significant effort to take advantage of the GJK algorithm because open-source implementations have limited portability, lack of documentation and/or robustness. For instance, an early implementation by Cameron [9] has been reported as inaccurate [8]. Whereas Solid [10] and Bullet [11], two renowned libraries for collision detection, target computer games and are not suitable for accurate engineering applications. The scientific community is essentially left with no choice but to re-implement the GJK algorithm, which is an involved and time-consuming task.

With the aim of providing researchers from all communities with an easy-to-use solution, openGJK is now released. It features the fastest and more accurate version of GJK algorithm [12], is cross-platform and available for C/C++ with interfaces for C# and Matlab. Moreover, to cover all relevant applications, it comes in two flavours: *fast* and *accurate*. These differ only by the

[*] Corresponding author.
E-mail address: mattia.montanari@eng.ox.ac.uk (M. Montanari).

https://doi.org/10.1016/j.softx.2018.10.002
2352-7110/© 2019 Published by Elsevier B.V.

floating-point arithmetic routines used, and the *fast* version suits nearly all applications.

This paper presents examples for importing openGJK and demonstrates its applicability. Section 2 describes the library. Section 3 introduces two examples that illustrate how to solve Eq. (1). Finally, Section 4 shows applications in computational mechanics, to detect collision between objects, and material science, to solve a large number of distance queries in an optimisation problem.

2. Software

OpenGJK is written in C, licensed under the GPL conditions and is compiled across multiple platforms with CMake. Doxygen is used to generate the documentation of the library.

2.1. Functionality

OpenGJK provides a good compromise between capabilities and portability for solving the problem in Eq. (1). It is designed to solve distance queries out of the box without any user-defined parameter.

To minimise the number of dependencies and to keep the inputs as simple as possible, two (secondary) features of the GJK algorithms are not supported. Firstly, even though the GJK algorithm can handle quadrics and splines [7,8], openGJK is limited to spheres, clouds of points and polytopes. Secondly, openGJK does not provide an incremental version of the GJK algorithm, such as [9]. Despite these limitations, openGJK handles most applications in computer graphics, computational mechanics, optimisation and robotics — even in real-time.

The interfaces with C# and Matlab facilitate modern computing practices, where coding often involves scripting languages for prototyping first, and low-level programming afterwards. This allows users to run the same code throughout the development of a scientific program.

Finally, openGJK is multi-thread safe, C/C++ compatible and it comes in two flavours: *fast* (default) and *accurate*. The mode is selected with CMake. The default mode will suit most applications in computer graphics and engineering, including for collision detection purposes. However, some applications prioritise accuracy to speed, for example mesh generation and the optimisation problem illustrated in Section 4. Only for these applications the accurate mode is recommended.

2.2. Algorithms

The literature presents several versions of the GJK algorithm; the one discussed in [12] addresses a well-known stability issue and it is therefore implemented in openGJK. This section presents a brief explanation.

The GJK algorithm is iterative and may be represented as a simple conditional loop [9]. The loop solves Eq. (1) by updating a vector $\mathbf{v}_k$ at each iteration, so that $\mathbf{v}_{k+1}$ converges to the solution; namely:

```
while   Not close_enough ( $\mathbf{v}_k$ )
        $\mathbf{v}_{k+1}$ ← subalgorithm ( $\mathbf{v}_k$ )          (2)
end
```

Upon convergence the solution $\|\mathbf{v}_k\| = d(\Omega_P, \Omega_Q)$ is verified [1]. Eq. (2) suggests that the optimal convergence rate and the maximum accuracy for $d(\Omega_P, \Omega_Q)$ are achieved only if $\mathbf{v}_{k+1}$ is computed accurately to machine precision.

The role of `subalgorithm` is often overlooked in the literature, with consequences that include reduced speed and limited accuracy [12]. Scientific applications involving fine meshes, such as finite element analyses of large-deformations, are penalised by the lack of accuracy of the sub-algorithm originally used in [1]. The following paragraph briefly describes how the method presented in [12] addresses this issue.

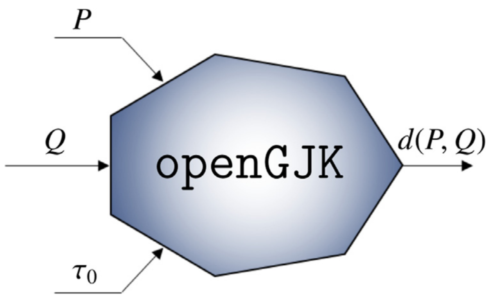

Fig. 1. The inputs required by openGJK are two sets of points, P and Q, which describe two bodies. Optionally, the user may decide to initialise the simplex τ_0. The output is the minimum distance between the two bodies.

The vector $\mathbf{v}_k$ is computed by solving a system of equations $A\lambda = b$. This has a peculiar feature: given two vectors $\mathbf{p} = p_i - p_j$ with $p_i,\ p_j \in P$ and $\mathbf{q} = q_i - q_j$ with $q_i,\ q_j \in Q$, A embeds an orthogonal condition between $\mathbf{v}_k$ and a vector $\mathbf{s} = \mathbf{p} + \mathbf{w}$. This imposes that $d(\Omega_P, \Omega_Q)$ effectively is the minimum distance between the bodies. However, the difference operator defining $\mathbf{p}$ and $\mathbf{q}$ is subject to cancellation error, which in turn affects the coefficients of A, and hence the solution of $A\lambda = b$. The new sub-algorithm introduced in [12] transfers the orthogonal condition from A to the right-hand side of the system of equations. This yields to a new system $M\lambda = p$, whose solution does not suffer from cancellation error and is computed accurately to machine precision.

2.3. Inputs and cost

The required inputs are simply two sets of points whose coordinates define the bodies. These may be polytopes, point clouds or spheres. Even if available, the information about the connectivity of these points is not needed. Fig. 1 presents inputs and outputs to openGJK. The user may choose to initialise the simplex τ_0 for Eq. (2), but this is not required.

Finally, two remarks about the inputs and the relationship that these have with the computational cost:

1. Should the bodies be non-convex, because of the nature of the GJK algorithm [1], openGJK will return the minimum distance between their convex hulls. Notice, however, that convex hulls are never computed explicitly.
2. The cost of a distance query increases linearly with the number of points in input. Should the reader experience slow performance of openGJK, it is advisable to improve the inputs, e.g. by removing points not on the frontier of the bodies.

2.4. Implementation

The principal functions invoked are shown in the diagram in Fig. 2. The sub-algorithm invokes three functions recursively to compute the minimum distance between the origin and a simplex. More details can be found in [12].

The library is written to be readable rather than highly optimised. The interfaces with C# and Matlab are provided to facilitate the use of openGJK. The latter uses a *mex* file, a dynamically linked subroutine, that enables to call openGJK within Matlab.

By default, openGJK does not have external dependencies. Only if compiled in the *accurate* mode CMake will link to `ext/predicates.c` to use the arbitrary precision floating-point arithmetic routines published by Shewchuk [13].

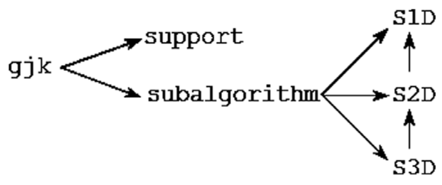

Fig. 2. Functions call stack of the openGJK library.

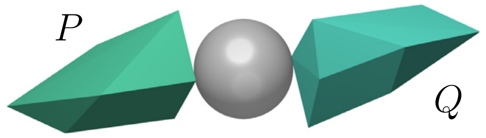

Fig. 3. Two polytopes P and Q at a distance, corresponding to the diameter of the sphere, computed with the mex function `openGJK`.

3. Examples

This section presents two examples where openGJK is: (i) linked to a standalone C application (ii) invoked within Matlab.

3.1. Standalone demo

OpenGJK may be imported in a C program in three simple steps:

1. link the library and include header files,
2. generate a suitable input data structure,
3. invoke the GJK algorithm.

The source code for this example is included in the `example1_c` folder and the reader can find in `main.c` comments that reflect the three steps above. Step (2) requires particular care; two variables for each body have to be specified: the number of vertices and the pointers to their coordinates. These are stored into a structure `bd` and used by the GJK algorithm. The structure `simplex` also needs to be initialised.

The program `demo` can be built with the provided CMake script to compute the minimum distance between two bodies whose coordinates are read from two input files: `userP.dat` and `userQ.dat`. The reader may edit these files to define different bodies.

3.2. Interface with Matlab

OpenGJK may be invoked as any Matlab built-in function. This section provides instructions for the compiler and an explanatory example.

The script `runme.m`, in the `example2_mex` folder, automatically compiles a mex function[1] and computes the minimum distance between the polytopes depicted in Fig. 3. The figure also includes a sphere which represents the distance between the two bodies, thus the outcome of the GJK algorithm.

The reader can invoke openGJK in any existing Matlab program simply by calling the `openGJK` function, as shown in `main.m`.

4. Impact

OpenGJK is relevant to a large range of applications since distance queries are ubiquitous in computer graphics, robotics and scientific computing. By adopting openGJK, researchers will reduce implementation effort and computing time spent to solve distance queries. The library is released to meet the demand of those who prototype using Matlab or C# and those who develop software in C/C++.

The applicability is herein demonstrated with two examples. The first one uses openGJK to reduce the computing time spent in collision detection for the study of granular materials. Recently, computational tools were employed to analyse the microscopic interactions between sand grains [14]. An example is illustrated in Fig. 4, where 100 grains fall under gravitational load and are compressed by a weight. OpenGJK is used in an in-house software for modelling grains as polytopes, rather than spherical particles, to enhance the fidelity of the simulation [15]. Each grain is meshed using triangles with a minimum edge length $H_{\min}$. The CPU time spent by openGJK for different values of $H_{\min}$ and numbers of grains is reported in Table 1.

The second application links openGJK to the software Neper [16], which is used in engineering and material science to study polycrystal microstructures. For example, Neper can reproduce the morphology of an experimental sample in a digital format suitable for finite element analysis. Herein, the beta Titanium alloy sample from [17] is used; Fig. 5(a) and 5(b) illustrate the experimental and digital morphologies, respectively. To reproduce a microstructure Neper essentially solves an optimisation problem. It minimises the target function f by solving a large number of distance queries with the GJK algorithm [18]. Because this process requires high accuracy, Neper currently implements the slower but more accurate version of the GJK algorithm described in [3]. This can now be replaced by openGJK, which is equally accurate but remarkably faster. In fact, Fig. 5(c) shows that with openGJK the same minimum value f is reached, but in less computing time. On a single thread the method originally implemented in Neper requires 174 min, whereas openGJK only 133 min, thus offering a reduction of CPU time equal to 25%.

Table 1
Overall CPU time spent by openGJK for contact detection in models with 100, 1200 and 2400 grains using different edge lengths $H_{\min}$.

$H_{\min}$ (µm)	Number of grains	CPU time (s)
0.15	100	0.08
0.15	1200	0.22
0.15	2400	6.25
0.10	100	0.16
0.10	1200	0.78
0.10	2400	15.78
0.07	100	0.17
0.07	1200	1.31
0.07	2400	28.30

5. Conclusions

OpenGJK aims to provide researchers with an efficient and easy-to-use implementation of the GJK algorithm. The library allows to solve distance queries quickly, accurately and robustly; it is therefore designed to serve a broad range of applications, from scientific computing to computer graphics. Spheres, point clouds and any polytope may be treated, including simplices (e.g. line segments and triangles) and degenerated geometries (e.g. needles). Examples are provided to facilitate the use of openGJK and to demonstrate its superior performance over existing implementations of the GJK algorithm.

Acknowledgements

The financial support of Innovate UK is gratefully acknowledged. The first author wishes to thank Dr Romain Quey for sharing useful insight on the software Neper.

[1] See Matlab documentation for the requisites to compile mex functions.

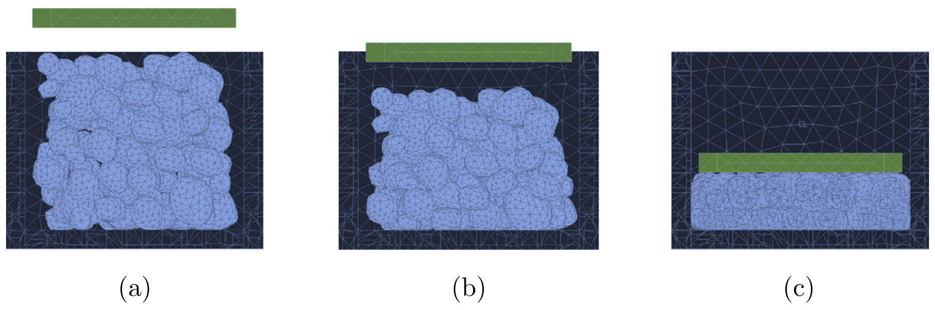

Fig. 4. Deposition of 100 grains in a confined box and compressive weight falling under gravitational load.

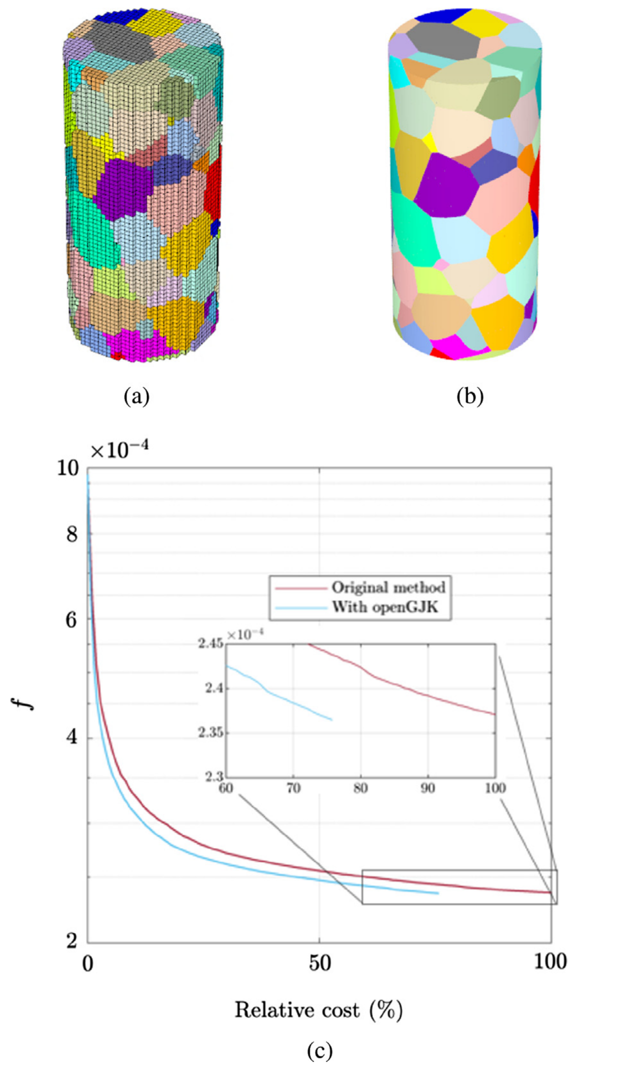

Fig. 5. Neper can receive an experimental diffraction contrast tomography image (a) and generate a virtual sample of it (b) by solving an optimisation problem. The aim is to minimise the function f and, for this particular morphology, openGJK reduces the computing time by approximatively 25%.

Conflict of interest

The authors have no conflict of interest to declare.

References

[1] Gilbert E, Johnson D, Keerthi S. A fast procedure for computing the distance between complex objects in three-dimensional space. IEEE J Robot Autom 1988;4(2):193–203. http://dx.doi.org/10.1109/56.2083.
[2] Ong C-J, Gilbert E. Fast versions of the Gilbert-Johnson-Keerthi distance algorithm: Additional results and comparisons. IEEE Trans Robot Autom 2001;17(4):531–9. http://dx.doi.org/10.1109/70.954768.
[3] Ericson C. Real-time collision detection. Morgan Kaufmann; 2004.
[4] Chen ER, Klotsa D, Engel M, Damasceno PF, Glotzer SC. Complexity in surfaces of densest packings for families of polyhedra. Phys Rev X 2014;4(1):011024. http://dx.doi.org/10.1103/PhysRevX.4.011024.
[5] Tasora A, Anitescu M. A matrix-free cone complementarity approach for solving large-scale, nonsmooth, rigid body dynamics. Comput Methods Appl Mech Engrg 2011;200(5):439–53. http://dx.doi.org/10.1016/j.cma.2010.06.030.
[6] Cameron S. A comparison of two fast algorithms for computing the distance between convex polyhedra. IEEE Trans Robot Autom 1997;13(6):915–20. http://dx.doi.org/10.1109/70.650170.
[7] Turnbull C, Cameron S. Computing distances between NURBS-defined convex objects. In: Proceedings of IEEE International Conference on Robotics and Automation, vol. 4; 1998. p. 3685–90. http://dx.doi.org/10.1109/ROBOT.1998.681406.
[8] van den Bergen G. Collision detection in interactive 3D environments. Morgan Kaufmann; 2003.
[9] Cameron S. Enhancing GJK: Computing minimum and penetration distances between convex polyhedra. In: Proc. of IEEE International Conference on Robotics and Automation, vol. 4; 1997. p. 3112–7. http://dx.doi.org/10.1109/ROBOT.1997.606761.
[10] van den Bergen G. Solid: software library for interference detection. 2017, http://www.dtecta.com/.
[11] Erwin C. Bullet: Real-time physics simulation. 2017, http://bulletphysics.org/.
[12] Montanari M, Petrinic N, Barbieri E. Improving the GJK algorithm for faster and more reliable distance queries between convex objects. ACM Trans Graph 2017;36(3):30:1–30:17. http://dx.doi.org/10.1145/3083724.
[13] Shewchuk JR. Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete Comput. Geom. 1997;18(3):305–63. http://dx.doi.org/10.1007/PL00009321.
[14] Haustein M, Gladkyy A, Schwarze R. Discrete element modeling of deformable particles in YADE. SoftwareX 2017;6:118–23. http://dx.doi.org/10.1016/j.softx.2017.05.001.
[15] Smeets B, Odenthal T, Vanmaercke S, Ramon H. Polygon-based contact description for modeling arbitrary polyhedra in the discrete element method. Comput Methods Appl Mech Engrg 2015;290(Suppl. C):277–89. http://dx.doi.org/10.1016/j.cma.2015.03.004.
[16] Quey R, Dawson P, Barbe F. Large-scale 3D random polycrystals for the finite element method: Generation, meshing and remeshing. Comput Methods Appl Mech Engrg 2011;200(17–20):1729–45. http://dx.doi.org/10.1016/j.cma.2011.01.002.
[17] Ludwig W, King A, Reischig P, Herbig M, Lauridsen EM, Schmidt S, et al. New opportunities for 3D materials science of polycrystalline materials at the micrometre lengthscale by combined use of X-ray diffraction and X-ray imaging. In: Special topic section: Probing strains and dislocation gradients with diffraction. Mater Sci Eng A 2009;524(1):69–76. http://dx.doi.org/10.1016/j.msea.2009.04.009.
[18] Quey R, Renversade L. Optimal polyhedral description of 3D polycrystals: Method and application to statistical and synchrotron X-ray diffraction data. Comput Methods Appl Mech Engrg 2018;330(Suppl. C):308–33. http://dx.doi.org/10.1016/j.cma.2017.10.029.