GJK Algorithm

David Ogunlesi 700017447

***Abstract*—The GJK algorithm is a revolutionary collision detection technique used in a wide range of applications in computer graphics, physics simulation, robotics, and computer vision. Using intuitive reductions, it takes the task of finding the intersection of two points and making it trivial to compute.**

I certify that all material in this dissertation which is not my own work has been identified.

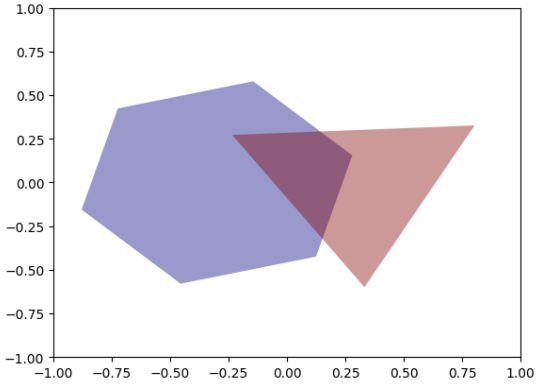
Signature:  _____

I. Introduction

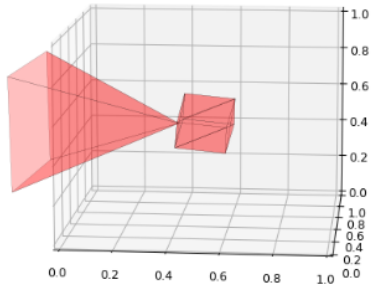
The GJK algorithm revolutionized the way we think about and interact with objects in the digital world. Before its discovery, determining the intersection of two objects was a complex and computationally expensive task. The GJK algorithm offered a simple and efficient solution, allowing for the creation of more realistic and interactive simulations in fields such as computer graphics, physics, and engineering.[2] This, in turn, has had a profound impact on the way we design and build everything from virtual reality experiences and video games to aerospace and automotive systems. In short, the GJK algorithm has fundamentally changed how we understand and manipulate objects in the digital realm and has paved the way for countless innovations in science and technology.

II. The problem

The problem at hand is not difficult to express and appears to be deceptively simple at first:



Do these two shapes intersect?



Do these?

As humans, we can visually see an intersection, but a question that can be verified easily does not lead to an easily solvable question. I prompt you. Try to come up with a solution. You may succeed, but then try making it work for concave shapes. An algorithm that can solve this problem with ease is the Gilbert Johnson Keerthi (GJK) algorithm, and it solves it with style. And like the algorithm, this report won't be a straight line, but full of twists and turns, eventually leading to an intuitive understanding.

III. Intuition and Insights

All shapes can be divided into two classes. Concave and Convex. Concave shapes are easy to work with. Convex are something we avoid. However:

Intuition 1: Any Convex shape can be described by a series of concave shapes.

A. Convexity and Infinite Points

We can try an interesting thought experiment that may help solve this problem. Imagine we represent shapes by a set of infinite points. We can now solve this problem since if there is one point in common between the shapes, there is an intersection *by definition*. We would need a magic computer of course, but this highlights intuition 2:

Intuition 2: If we can find two points whose difference is equal to the origin, there is an intersection.

B. Minkowski Sums and Differences

This leads to an interesting idea. What if we could subtract every point from one shape and subtract it from the other? This is called the Minkowski difference.[2] Formally defined as

$$\underbrace{A \otimes B = \{a + b | a \in A, b \in B\}}_{\text{Minkowski sum of A and B}}$$

For the algorithm what we care about is the Minkowski difference, which is derivable from the sum:[3]

$$\underbrace{A \otimes B = \{a + (-b) | a \in A, b \in B\}}_{\text{Minkowski difference of A and B}}$$

There are only two properties that we care about, which bring us new insight and intuition.

Insight 1: If both shapes are convex, the Minkowski difference must be convex

Insight 2: If the Minkowski difference contains the origin, the shapes must intersect.

Observe these until you intuitively grasp them, because they form the basis of the algorithm. We have shifted the perspective and now transformed this problem from an intersection to checking whether the Minkowski difference contains the origin. However, infinite points are computationally infeasible to compute. What can we do?

C. Simplexes

If we take 3 samples of points on the border of a shape and compute the Minkowski difference, we produce a simplex. Recalling **insight 1**, if both our shapes are convex, then the difference is convex. which brings us to more intuitions:

Intuition 4: If the simplex is convex, its edges must fall within the Minkowski difference by *definition* of convexity.

Intuition 5: If this simplex contains the origin, then the Minkowski difference must also contain it.

This brings us to a core insight into the algorithm:

Insight 3: If we can build a simplex that surrounds the origin, then we know the shapes intersect.

IV. The Algorithm: A Board Perpsective

From the insight in the previous section we have realised that to solve this problem, what we need to do is answer this question:

Can we build a simplex out of the Minkowski difference that surrounds the origin?

Interestingly we can generalise this to N-dimensions as simplexes are N-dimensional objects.

A. A Roadblock

However we have run into a problem, the Minkowski difference is a set of infinite points, so we want to select a finite set so that we can build our simplex.

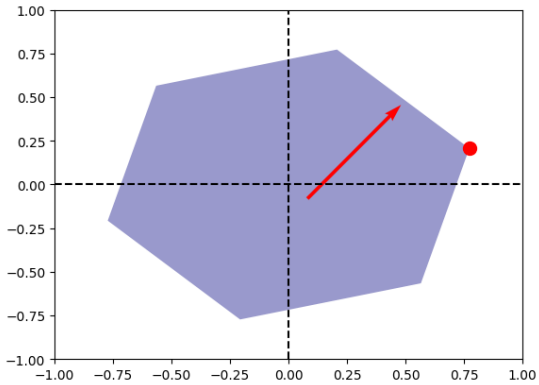
But how do select these points?

We would also like to select these points in a clever way to maximise the chance of building a simplex that surrounds the origin as quickly as possible.

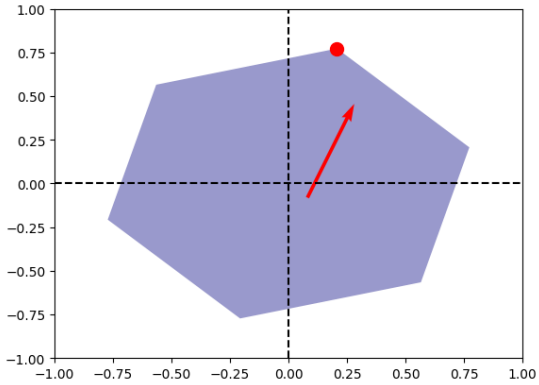
V. A Detour: Support Functions

Convex shapes have a useful property:

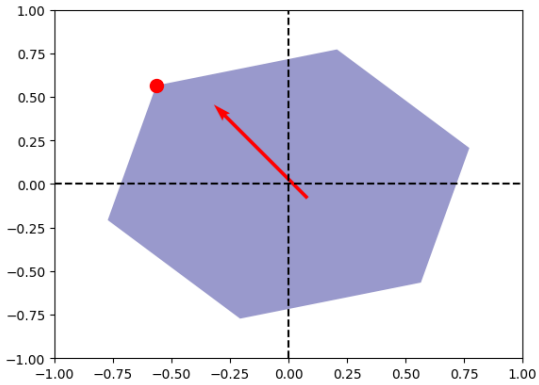
Insight 4: For any point on a shape, there is a direction in which it is the farthest away point.



(a). A direction of (1,1)



(b). A direction of (0.5,1)



(c). A direction of (-1,1)

Fig. 1: Support function visualisation in 2D

This is known as *support point*. Here is a key insight:

Insight 5: Taking the difference of the support points in the opposing directions gives a support point of the Minkowski difference.

Intuitively:

Intuition 6: We are finding the most extreme points in the Minkowski difference by taking support points from both shapes that would give the greatest difference.

VI. The Algorithm: A deeper dive

The algorithm is not complicated, its elegance is in its simplicity. The best way is to explain it step by step:

A. Step-by-Step Overview (2D)

1) We first start by picking a random direction vector and calculate the support point for both shapes
2) To select our next point, it would make the most sense to have a direction that points to the origin from the point we just selected.
3) Calculating this support point with this direction, we calculate whether the point is in a region beyond the origin (given the direction) as a sanity check.
4) We can then pick the normal vector (facing the origin) of the resultant line as our next direction. This makes sense as gives us the highest chance of the support point we select to be beyond the origin.
5) We calculate the final support point and do the same sanity check to make sure it is the past origin.
6) Now we have a completed simplex, we check whether it contains the origin.
7) If it doesn't we find a support point in the direction of the normal of the closest side to the origin
8) If this new simplex contains the origin the shapes are intersecting.

VII. Unasnwered Questions

There are still many unanswered questions [1]:

How do we know when the point has passed the origin?

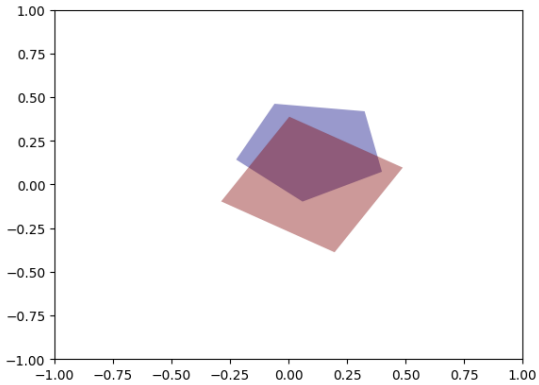
How do we calculate the direction once we have a line simplex?

How do we identify whether the simplex contains the origin?

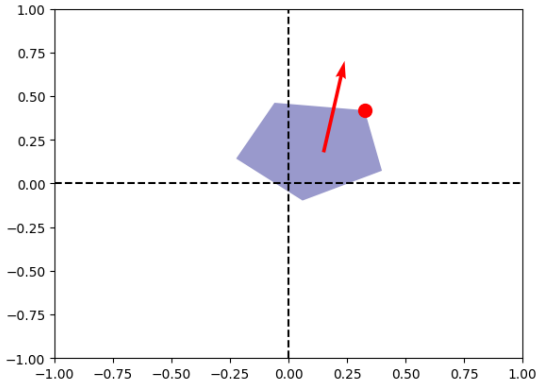
How do we pick a new direction if the simplex doesn't contain the origin?

These questions are best answered by showing the pseudocode of the algorithm.[1]

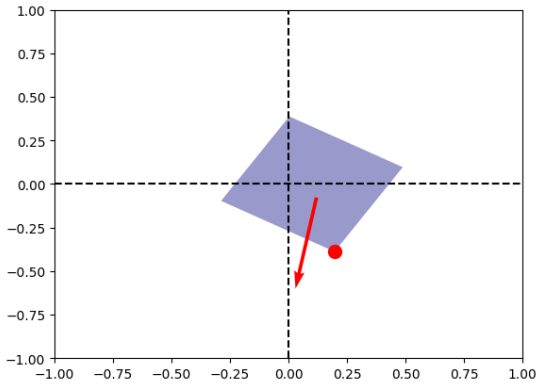
VIII. Implementation



(a). Problem



(b). Step 1



(c). Step 1

Fig. 2: Visualisation of GJK algorithm in 2D

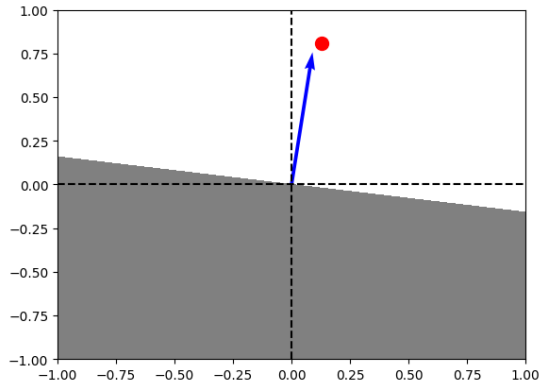
Algorithm 1 GJK algorithm

procedure GJK
  $d \leftarrow$ normalise$(s1.center - s2.center)$
  $simplex \leftarrow [support(s1, s2, d)]$
  $d \leftarrow ORIGIN - simplex[0]$
  while $True$ **do**
    $A \leftarrow support(s1, s2, d)$
    if $dot(A, d) < 0$ **then**
      return false
    $simplex.append(A)$
    if $handleSimplex(simplex, d)$ **then**
      return true

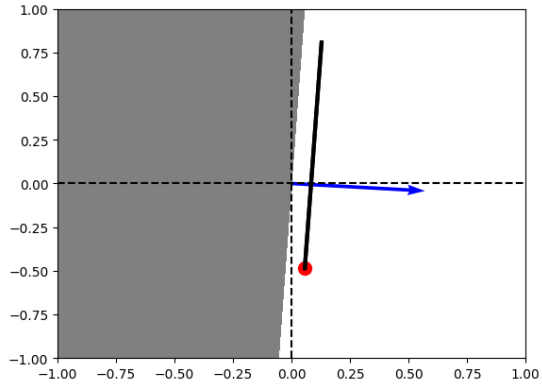
Algorithm 2 GJK algorithm

procedure SUPPORT(s1, s2, d)
  return $s1.furthestPoint(d) - s2.furthestPoint(-d)$

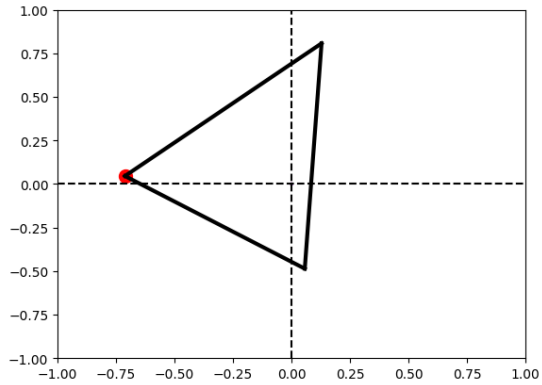
procedure HANDLESIMPLEX(simplex, d)
  if $len(simplex) == 2$ **then**
    return $lineCase(simplex, d)$
  if $len(simplex) == 3$ **then**
    return $triangleCase(simplex, d)$
  if $len(simplex) == 4$ **then**
    return $tetrahedronCase(simplex, d)$



(a). Step 2



(b). Step 4/5



(c). Step 6+)

Fig. 3: Visualisation of GJK algorithm in 2D

Algorithm 3 GJK algorithm

procedure LINECASE(simplex, d)
  $B, A \leftarrow simplex$
  $AB, AO \leftarrow B - A, ORIGIN - A$
  $ABperp \leftarrow tripleProduct(AB, AO, AB)$
  $d \leftarrow ABperp$
  return false

procedure TRIANGLECASE(simplex, d)
  $C, B, A \leftarrow simplex$
  $AC, AB, AO \leftarrow C - A, B - A, ORIGIN - A$
  $ABperp \leftarrow tripleProduct(AC, AB, AB)$
  $ACperp \leftarrow tripleProduct(AB, AC, AC)$
  if $dot(ABperp, AO) > 0$ **then**
    $simplex.remove(C)$
    $d \leftarrow ABperp$
    return false
  if $dot(ACperp, AO) > 0$ **then**
    $simplex.remove(B)$
    $d \leftarrow ACperp$
    return false
  return True

IX. Time Complexity

The GJK algorithm is truly amazing. It retains a completely linear time complexity [4], only being dependent on the number of vertices of the objects in the problem. Regardless of the dimension of space, it retains its complexity and can be generalised to N dimensions without any performance loss in terms of time complexity. This is evident by the fact that there is only one loop that runs until a valid simplex is found, but due to the nature of the algorithm, there is a maximum number of iterations it can run for until it has explored every possible simplex.

X. Limitations

Though GJK is very elegant, like any algorithm, it has its limitations.

The first limitation is that it can't be used to determine the exact point of intersection, it can only determine whether two objects are just intersecting. An extension to the algorithm would need to be added to make this possible.

Another limitation is that it can sometimes fail to converge to a solution if the objects are almost touching. The algorithm may return false positives or false negatives in these scenarios.

Overall, while the GJK algorithm is a powerful tool for detecting intersections between convex objects, it is not a perfect solution and can be limited in certain scenarios.

XI. Applications

One common application of the GJK algorithm is within the field of computer graphics. It forms the basis of object collision detection in a virtual environment, and is an integral part of game engines; but also other 3D graphical software such as CAD.

Another application is in the field of physics simulation, similarly to computer graphics applications, GJK can be used to determine whether 2 objects are colliding. This can be used to simulate the behaviour of objects due to physical forces and has various applications in engineering, robotics and molecular dynamics.

Expanding on the field of robotics, GJK can be used as a collision detection for robots, when they resolve their surroundings. Object manipulation, autonomous navigation and collision avoidance all benefit from the simplicity, speed and elegance of the GJK algorithm.

XII. Conclusion

In conclusion, the Gilbert-Johnson-Keerthi (GJK) algorithm is a widely used and powerful tool for detecting intersections between arbitrary convex objects. Its use of intuitive simplifications has made it a quick computation whilst being highly accurate and has pushed it into the midst of many fields of computer science such as physics simulation and robotics. While it does have some limitations, its speed is highly valuable; and extensions can easily be made to the algorithm to combat some of its pitfalls. Overall, the GJK algorithm continues to be a valuable contribution to the field of computational geometry and itd many applications.

References

[1] *A Strange But Elegant Approach to a Surprisingly Hard Problem (GJK Algorithm)*. YouTube, Mar. 2021. URL: https://www.youtube.com/watch?v=ajv46BSqcK4&t=1646s.

[2] Christer Ericson. "The Gilbert-Johnson-Keerthi algorithm". In: *URL https://www. fit. vutbr. cz/study/courses/VGE/private/reading/2/SIGGRAPH04_Ericson_GJK_notes. pdf* (2005).

[3] Elmer G Gilbert, Daniel W Johnson, and S Sathiya Keerthi. "A fast procedure for computing the distance between complex objects in three-dimensional space". In: *IEEE Journal on Robotics and Automation* 4.2 (1988), pp. 193–203.

[4] Patrick Lindemann. "The gilbert-johnson-keerthi distance algorithm". In: *Algorithms in Media Informatics* (2009).