Application of Outer Product for Vehicle Collision Detection in Automatic Navigation System

Sakti Bimasena – 13523053[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523053@mahasiswa.itb.ac.id, [2]sbimasena@gmail.com

***Abstract*—Automatic navigation systems has grown rapidly because of the development of sophisticated sensing technology and computing algorithms. However, vehicle collision systems right now often use methods that require high computational power, such as deep learning, that limits real-time efficiency. This study offers a new method to detect collisions using a mathematical function known as outer product. The offered method represents the path of a vehicle as vector data then uses outer product to calculate the intersection, it offers a lightweight yet effective alternative to conventional methods. The implementation shows accuracy in detecting potential collisions while maintaining computing efficiency. Testing results show that this method succeeds in handling many scenarios, like intersecting or parallel paths. This method would likely be a promising option for real-time vehicle collision detection in automatic navigation systems**

***Keywords*—Collision detection, automatic navigation system, outer product, geometric algebra.**

I. Introduction

With the introduction of sophisticated sensing technologies and computer algorithms, the development of automatic navigation systems has evolved considerably. These technologies are essential to modern transportation because they improve efficiency and safety in a variety of driving situations. However, vehicle collision detection is still facing challenges, especially because of changes in the environment, sensor difficulties, and limited computational power that affects real-time performance and accuracy.

Intensive computational methods like deep-learning based object detection framework like YOLOv5 and Mask-RCNN are widely used in vehicle collision detection techniques [1], [2]. These methods have been proven to be very accurate in many situations. Yet, oftentimes these methods are not efficient, it makes them less fit for real-time applications. Other than that, Environmental complexities such as visual obstructions (occlusion), bad weather, and curved roads can affect the reliability of this system [3]. Sometimes, the solutions used today fail at balancing strong performance with computation simplicity, that is important to solving the problem.

This study introduces a new method to detect vehicle collisions by using the mathematical function known as outer product. This method represents vehicle paths as vector data then uses outer product to calculate the intersection point. This method offers a computationally efficient and light alternative to conventional methods. Because this method can maintain high accuracy while also reducing computing power requirements, it is expected for this method to be a strong candidate for real-time use in automatic navigation systems.

II. Theoretical Foundation

A. Geometric Algebra

By adding new products and higher-dimensional things like bivectors and trivectors, geometric algebra—which was first developed by Hermann Grassmann and then improved by William Clifford—expands conventional vector algebra. This framework provides a coherent vocabulary for geometry and physics by bringing together disparate mathematical systems.

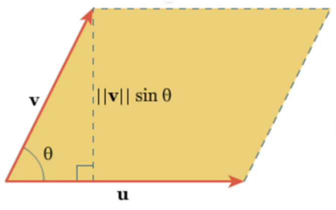


Fig. 2.1 *Parallelogram area with vector algebra*
Source: *yos3prens.wordpress.com*

Pictured above is a diagram which shows how to calculate the area of a parallelogram using vectors in vector algebra. The equation is as follows:

$$A = \| u \times v \| = \| u \| \| v \| \sin\theta$$

The area of a parallelogram made by vectors u and v is also equal to the determinant of those vectors. Since determinants can be negative, Grassman supported the concept of signed area and volume by introducing the concept of outer product.

B. Outer Product

The outer product, sometimes referred to as the wedge product, is a key notion in geometric algebra that offers a reliable tool for geometric computations and generalizes the idea of area and volume in higher dimensions.

The outer product of two vectors a and b, denoted as $\boldsymbol{a} \wedge \boldsymbol{b}$, produces a bivector representing the signed area of the parallelogram (positive or negative) spanned by these vectors. This operation is antisymmetric, meaning $\boldsymbol{a} \wedge \boldsymbol{b} = -\boldsymbol{b} \wedge \boldsymbol{a}$.

Outer product depends on the orientation in which the two vectors are located. If the path from the first vector to the second goes anti-clockwise, the end product is positive and vice versa.

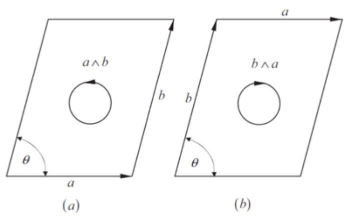


***Fig. 2.2** Representation of outer product orientation*
***Source:** informatika.stei.itb.ac.id/~rinaldi.munir/*

The magnitude of outer product represents the area of a parallelogram made by vectors a and b:

$$\| a \wedge b \| = \| a \| \| b \| \sin \theta$$

where θ is the angle between a and b. This property is particularly useful in determining the perpendicularity and area calculations in various dimensions.

C. Properties and Representation

The outer product possesses several key properties that make it advantageous for computational applications:

1. Distributivity: $a \wedge (b + c) = a \wedge b + a \wedge c$
2. Zero Product for Parallel Vectors: $a \wedge b = 0 \, if \, a \parallel b$

These characteristics make it possible to represent and manipulate geometric entities efficiently, allowing for operations like projections, rotations, and reflections to be carried out without requiring coordinate transformations. When computational resources are few in real-time systems, this efficiency is especially advantageous.

Vectors in geometric algebra are represented in terms of basis vectors $e_1, e_2, e_3, \ldots, e_n$. For example, a vector **a** in $\mathbb{R}^2$ can be written as $a = a_1e_1 + a_2e_2$. The outer product $a \wedge b$ in $\mathbb{R}^2$ can be explicitly calculated as:

$$a \wedge b = (a_1b_2 - a_2b_1)(e_1 \wedge e_2)$$

$a_1b_2 - a_2b_1$ represents the area of the parallelogram and $e_1 \wedge e_2$ is the unit bivector representing the plane spanned by e_1 and e_2. In other words, outer product $a \wedge b$ is scalar area times $e_1 \wedge e_2$ bivector unit.

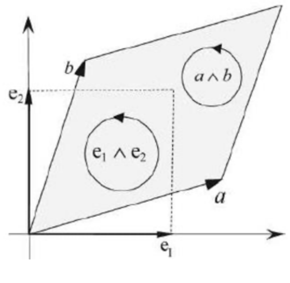


***Fig. 2.3** Diagram of vector representation in outer product*
***Source:** informatika.stei.itb.ac.id/~rinaldi.munir/*

D. Application in Calculating the Intersection of Two Lines

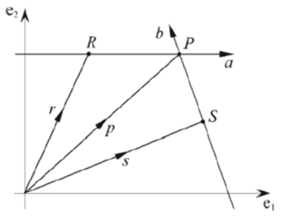


***Fig. 2.4** Intersecting lines example*
***Source:** informatika.stei.itb.ac.id/~rinaldi.munir/*

Observe the Fig. above, line a passes through point R and line b passes through point S. Both lines intersect at point P. To figure out where point P is located, first assume that $a = a_1e_1 + a_2e_2$ & $b = b_1e_1 + b_2e_2$. P is defined by $p = \alpha a + \beta b$, and its coordinates are:

$$x_p = \alpha x_a + \beta x_b$$
$$y_p = \alpha y_a + \beta y_b$$

To find α and β, we can substitute the above equation to get:

$$\alpha = \frac{x_py_b - x_by_p}{x_ay_b - x_by_a} = \frac{\begin{vmatrix} x_p & y_p \\ x_b & y_b \end{vmatrix}}{\begin{vmatrix} x_a & y_a \\ x_b & y_b \end{vmatrix}}$$

$$\beta = \frac{x_py_a - x_ay_p}{x_by_a - x_ay_b} = \frac{\begin{vmatrix} x_p & y_p \\ x_a & y_a \end{vmatrix}}{\begin{vmatrix} x_b & y_b \\ x_a & y_a \end{vmatrix}}$$

Therefore,

$$p = \frac{\begin{vmatrix} x_p & y_p \\ x_b & y_b \end{vmatrix}}{\begin{vmatrix} x_a & y_a \\ x_b & y_b \end{vmatrix}} a + \frac{\begin{vmatrix} x_p & y_p \\ x_a & y_a \end{vmatrix}}{\begin{vmatrix} x_b & y_b \\ x_a & y_a \end{vmatrix}} b \rightarrow \frac{p \wedge b}{a \wedge b} a + \frac{p \wedge a}{b \wedge a} b$$

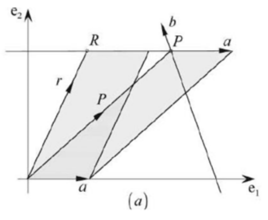


***Fig. 2.5** Intersecting line example picture (a)*
***Source:** informatika.stei.itb.ac.id/~rinaldi.munir/*

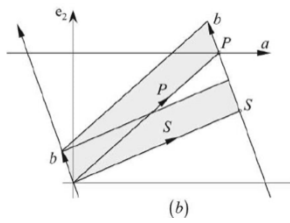


***Fig. 2.6** Intersecting line example picture (b)*
***Source:** informatika.stei.itb.ac.id/~rinaldi.munir/*

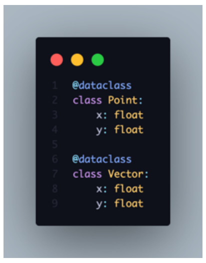
Observe both of the above pictures. In picture a, we can see that $p \wedge a$ is identic to $r \wedge a$. In picture b, we can see that $p \wedge b$ is identic to $s \wedge b$. Therefore,

$$p = \frac{p \wedge b}{a \wedge b} a + \frac{p \wedge a}{b \wedge a} b \rightarrow p = \frac{s \wedge b}{a \wedge b} a + \frac{r \wedge a}{b \wedge a} b$$

III. Implementation Program

The outer product is used to identify intersections between line segments in the collision detection program, which is based on the geometric characteristics of vehicle routes in a 2D plane. The implementation of the collision detection system It uses Python, with numpy for numerical operations and matplotlib for graphical output. The program is structured into modular classes, each focusing on specific aspects of collision detection and visualization. The technique guarantees accuracy and computing efficiency in detecting possible collisions by describing pathways as vectors.

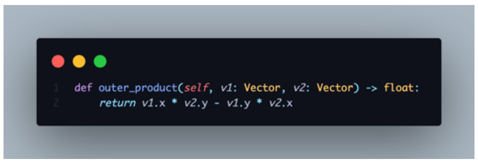
The program's main data structures are based on the representation of vehicle path vectors. Each path is defined by two points, the start and end points. To show coordinates in 2D space, these points are wrapped in a custom Point class. Besides that, vehicle paths are modeled using a Vector class that operations like scalar multiplication and vector addition. To calculate the intersection point between the vehicles, those operations are important.



```python
@dataclass
class Point:
    x: float
    y: float

@dataclass
class Vector:
    x: float
    y: float
```

Fig. 3.1 *Custom class used in the program*
Source: *Author*

The core of this program is the VehicleCollisionDetector class, that has the feature to detect collisions. Outer product calculations between two vectors is the main function of this class. Outer product has an important role in determining if two vectors are parallel or intersecting. This is how this is implemented:

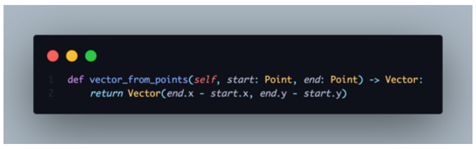


```python
def outer_product(self, v1: Vector, v2: Vector) -> float:
    return v1.x * v2.y - v1.y * v2.x
```

Fig. 3.2 *Outer product calculation implementation*
Source: *Author*

If the outer product of the two vectors are zero, then the vectors are parallel, meaning there is no chance of intersecting. On the contrary, if the outer product has a value other than zero, then the two vectors have a chance in intersecting somewhere along the way.

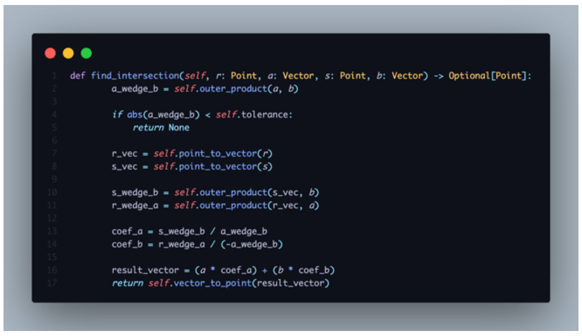
Moreover, the program provides utility functions that help convert points into vector and vice versa. For example, vector_from_points method can make vectors based on two points:



```python
def vector_from_points(self, start: Point, end: Point) -> Vector:
    return Vector(end.x - start.x, end.y - start.y)
```

Fig. 3.3 *Vector_from_point method*
Source: *Author*

These fundamental techniques are used in the program to calculate the intersection point of two line segments. To find the location of intersection, find_intersection function takes the coefficient from outer product into the vector formula. This implementation is explicitly linked to the theoretical foundation discussed in section II.B. Other than that, special situations, like parallel line segments, are considered in this method.

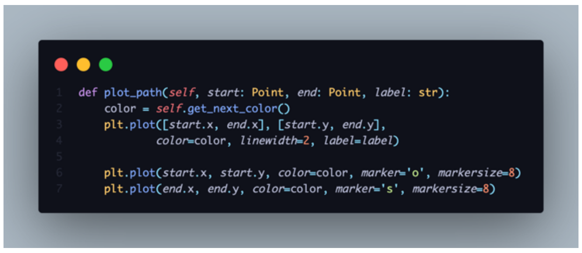


```python
def find_intersection(self, r: Point, a: Vector, s: Point, b: Vector) -> Optional[Point]:
        a_wedge_b = self.outer_product(a, b)

        if abs(a_wedge_b) < self.tolerance:
            return None

        r_vec = self.point_to_vector(r)
        s_vec = self.point_to_vector(s)

        s_wedge_b = self.outer_product(s_vec, b)
        r_wedge_a = self.outer_product(r_vec, a)

        coef_a = s_wedge_b / a_wedge_b
        coef_b = r_wedge_a / (-a_wedge_b)

        result_vector = (a * coef_a) + (b * coef_b)
        return self.vector_to_point(result_vector)
```

Fig. 3.4 *find_intersection method*
Source: *Author*

With the help of is_point_in_segment function, this program also validates wether the intersection point that was calculated is actually within both line segments. This makes sure that the intersection that was found is valid geometrically and not the product of extrapolation outside the defined line segment boundaries.

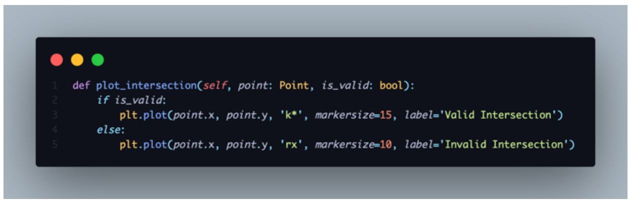
To support the collision detection logic, VehiclePathVisualizer class provides visual representation of vehicle paths and intersection points. To understand the output of the detection algorithm, visualization is very important, especially in complex situations. Paths are plotted in the plot_path method, that marks the start and end points clearly and uses different colours for each path:



```python
def plot_path(self, start: Point, end: Point, label: str):
    color = self.get_next_color()
    plt.plot([start.x, end.x], [start.y, end.y],
             color=color, linewidth=2, label=label)

    plt.plot(start.x, start.y, color=color, marker='o', markersize=8)
    plt.plot(end.x, end.y, color=color, marker='s', markersize=8)
```

Fig. 3.5 *plot_path method*
Source: *Author*

When an intersection is detected, it is highlighted on the plot with a clear star symbol, as shown in the plot_intersection method:



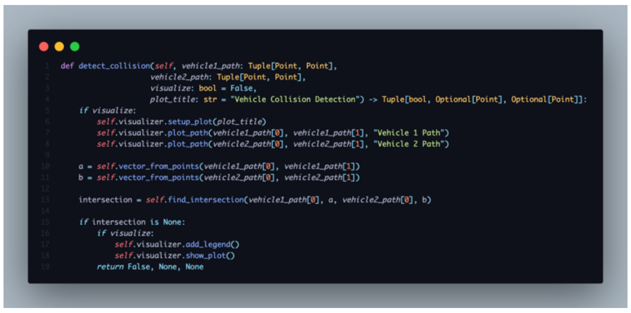
```
def plot_intersection(self, point: Point, is_valid: bool):
    if is_valid:
        plt.plot(point.x, point.y, 'k*', markersize=15, label='Valid Intersection')
    else:
        plt.plot(point.x, point.y, 'rx', markersize=10, label='Invalid Intersection')
```

__Fig. 3.6__ plot_intersection method
***Source:** Author*

To improve readability, the visualiser also controls grid layouts, legends, and plot titles. Each plot is guaranteed to be self-explanatory and intuitive thanks to the setup_plot and add_legend functions.

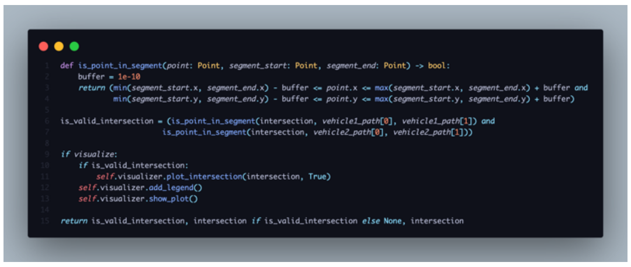
Initialising vehicle pathways as tuples of Point objects, which indicate their start and finish coordinates, is the first step in the entire operation. After that, the VehicleCollisionDetector receives these trajectories for examination. The detector computes the outer product, evaluates whether an intersection exists, and computes the direction vectors for each pair of pathways. An intersection is noted and shown if it is located and verified.

The detect_collision method orchestrates the detection and visualization process:



```
def detect_collision(self, vehicle1_path: Tuple[Point, Point],
                     vehicle2_path: Tuple[Point, Point],
                     visualize: bool = False,
                     plot_title: str = "Vehicle Collision Detection") -> Tuple[bool, Optional[Point], Optional[Point]]:
    if visualize:
        self.visualizer.setup_plot(plot_title)
        self.visualizer.plot_path(vehicle1_path[0], vehicle1_path[1], "Vehicle 1 Path")
        self.visualizer.plot_path(vehicle2_path[0], vehicle2_path[1], "Vehicle 2 Path")

    a = self.vector_from_points(vehicle1_path[0], vehicle1_path[1])
    b = self.vector_from_points(vehicle2_path[0], vehicle2_path[1])

    intersection = self.find_intersection(vehicle1_path[0], a, vehicle2_path[0], b)

    if intersection is None:
        if visualize:
            self.visualizer.add_legend()
            self.visualizer.show_plot()
        return False, None, None
```

__Fig. 3.7__ detect_collision method (a)
***Source:** Author*



```
def is_point_in_segment(point: Point, segment_start: Point, segment_end: Point) -> bool:
    buffer = 1e-10
    return (min(segment_start.x, segment_end.x) - buffer <= point.x <= max(segment_start.x, segment_end.x) + buffer and
            min(segment_start.y, segment_end.y) - buffer <= point.y <= max(segment_start.y, segment_end.y) + buffer)

is_valid_intersection = (is_point_in_segment(intersection, vehicle1_path[0], vehicle1_path[1]) and
                         is_point_in_segment(intersection, vehicle2_path[0], vehicle2_path[1]))

if visualize:
    if is_valid_intersection:
        self.visualizer.plot_intersection(intersection, True)
    self.visualizer.add_legend()
    self.visualizer.show_plot()

return is_valid_intersection, intersection if is_valid_intersection else None, intersection
```

__Fig. 3.8__ detect_collision method (b)
***Source:** Author*

The results are displayed graphically and printed to the console, providing both a visual and textual understanding of the collision.

IV. ANALYSIS

The performance and resilience of the collision detection algorithm were assessed across a variety of scenarios. Every scenario was created to resemble actual situations and evaluate how accurate the mathematical model is

Two vehicle trajectories crossed at a distinct crossing point in the first scenario, which featured intersecting routes. The intersection was correctly identified by the computer, which also indicated the collision point in the visualisation. For example, when Vehicle 1 travelled from (0,0) to (4,4) and Vehicle 2 travelled from (0,4) to (4,0), the program successfully located the junction at (2,2) and presented the outcome. This proved that the program was accurate in simple situations.

.

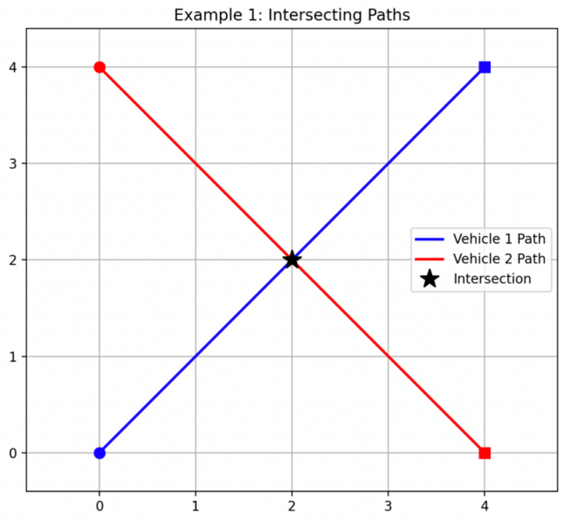
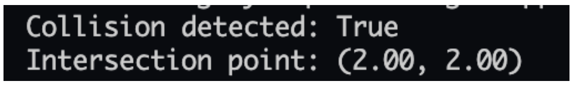


__Fig. 4.1__ Visualization result for the first example
***Source:** Author*



```
Collision detected: True
Intersection point: (2.00, 2.00)
```

__Fig. 4.2__ Text result for the first example
***Source:** Author*

To make sure the algorithm doesn't mistakenly detect collisions when there isn't an intersection, parallel pathways were evaluated in the second instance. For instance, the routes taken by Vehicles 3 and 4 from (0,0) to (4,4) and (0,1) to (4,5) were examined. The software generated no false positives and accurately determined that the pathways did not overlap. The program's dependability in managing parallel paths was confirmed by this case.

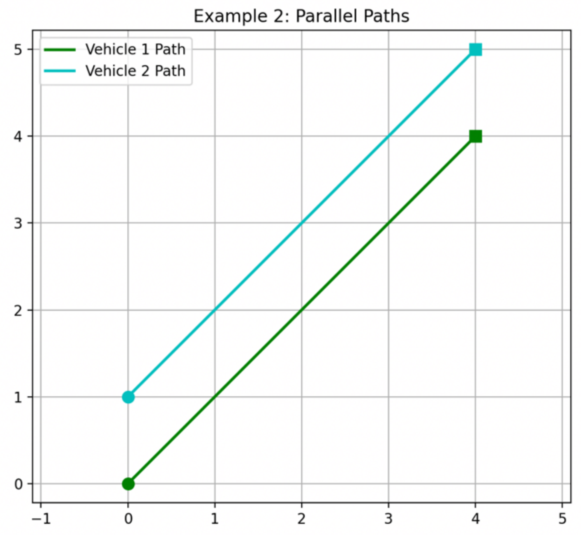
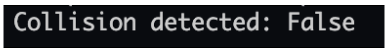


__Fig. 4.3__ Visualization result for the second example

Source: *Author*



```
Collision detected: False
```

Fig. 4.4 *Text result for the second example*
Source: *Author*

The third case was non-overlapping routes, in which the segments did not overlap even though the extended lines of the trajectories should have. The path of Vehicle 5 was, for instance, specified from (0,0) to (2,2), but the path of Vehicle 6 extended from (4,0) to (6,−2). The theoretical intersection point was estimated by the software to be at (2,2). However, the algorithm correctly deduced that there was no collision because this point is outside the boundaries of both segments.

Two pathways were represented in the visualization without any junction markers, which supported the idea that the trajectories did not overlap inside their designated segments. This illustrated the program's capacity to differentiate between hypothetical and real intersections, guaranteeing precise collision detection outcomes even in edge situations where pathways approach but do not physically collide.

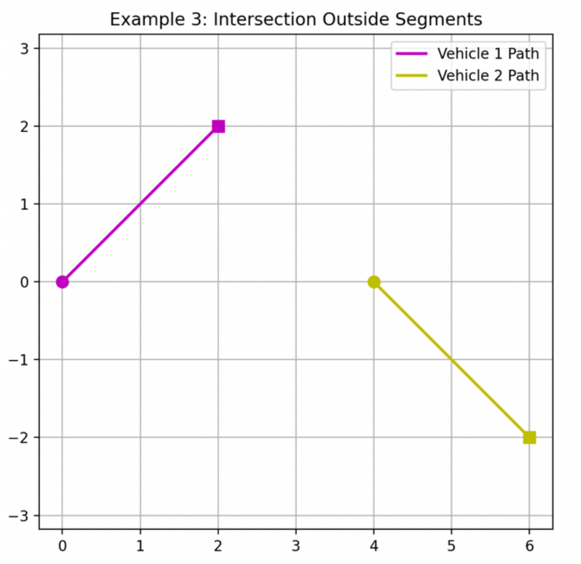
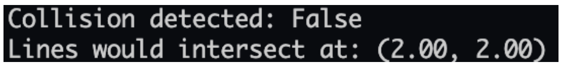


Fig. 4.5 *Visualization result for the third example*
Source: *Author*



```
Collision detected: False
Lines would intersect at: (2.00, 2.00)
```

Fig. 4.6 *Text result for the third example*
Source: *Author*

The results' communication was greatly aided by the visualisation module. The start and finish points of each trajectory were clearly indicated and colour-coded. Because the collision points were marked with a conspicuous star, the output was easy to understand. For instance, the visualisation supported the program's analytical findings in the non-overlapping pathways scenario by displaying the place of contact with a black star marker at the computed coordinates.

Some restrictions were noted, despite the program's strong performance in the test cases. It may not accurately depict curved or non-linear trajectories in real-world situations since it assumes that all pathways are straight-line segments. Furthermore, floating-point arithmetic precision problems were observed, particularly when almost parallel pathways with little angular deviations were involved.

All things considered, the application performed well in correctly identifying collisions, producing insightful visualizations, and managing a range of test situations. These findings identify areas for further study and refinement while showing the approach's potential for car navigation systems.

V. Conclusion

In this study, vehicle collision paths are found and analyzed by using mathematical methods that use outer product. By modelling vehicle paths as vectors and using vector arithmetic to find the intersection points, this program succeeds in differentiating theoretical intersections and actual collisions in defined segments. Its ability to handle many scenarios, like intersecting paths, parallel paths, and non-overlapping but theoretically intersecting paths, shows the accuracy of this implementation.

This method offers a light and computationally efficient solution to vehicle collision detection, that makes it a strong candidate to be integrated to automatic navigation systems where time performance is important. This program effectively proves its strength in many situations, ensuring accurate collision detection even in special situations.

Future research could see the development of this method to handle more complex paths geometries, like curved paths, and to incorporate uncertainty management into practical applications where the influence of the environment and sensor difficulties are critical. However, the framework presented here is a starting point in improving vehicle collision detection algorithms in contemporary transportation networks.

VI. Appendix

The source code for this program can be accessed at this link

VII. Acknowledgment

As the author of this paper, I would like to express my sincere gratitude to all parties who have provided support and inspiration during the writing process so that I can complete this paper entitled “Application of Outer Product for Vehicle Collision Detection in Automatic Navigation System” well. I would like to thank:

1. Ir. Rila Mandala, M.Eng., Ph.D. and Dr. Ir. Rinaldi, M.T. as the lecturers of IF2123 Aljabar Linier dan Geometri for the teaching of materials that have been shared in the Informatics Engineering class.

2. Both my parents for always supporting me. Their presence and positive affirmations always gives me the strength to finish this paper well.
3. My friends at Informatics Engineering class, who always cheer me up during the stressful times of creating this paper.

REFERENCES

[1] Redmon, J., & Farhadi, A. (2018). "YOLOv5: You Only Look Once - Object Detection.

[2] He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). "Mask R-CNN." IEEE Transactions on Pattern Analysis and Machine Intelligence.

[3] Badue, C., et al. (2021). "Self-Driving Cars: A Survey." Expert Systems with Applications.

[4] Munir, Rinaldi. (2023). "Aljabar Geometri (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-27-Aljabar-Geometri-Bagian1-2023.pdf

[5] Munir, Rinaldi. (2023). "Aljabar Geometri (Bagian 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-28-Aljabar-Geometri-Bagian2-2023.pdf

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 28 Desember 2024

[illegible]

Sakti Bimasena - 13523053