Intuitive Control Algorithm Development of 4WIS/4WID Using a SpaceMouse

Tae Wook (Tyler) Kim

Kent School Guild Presentation

Kent School

1 Macedonia Rd., Kent, CT 06757, USA

kimt20@kent-school.edu

In memory of Dong Min Lee 🎗

April 16th, 2019

1 Abstract

4WIS/4WID (**4 W**heel **I**ndependent **S**teering/**D**riving) is a steering system for a four-wheeled vehicle that allows for separate speed and direction controls for each wheel. These controls enable more versatile motion for vehicles that need to navigate tight spaces, such as a wheelchair in a narrow subway car. However, controlling two parameters, direction and speed, for each wheel results in eight parameters in need of simultaneous control. This research sought to achieve an intuitive control with a SpaceMouse that abstracts away the complexity, allowing for a full realization of the vehicle's capabilities without any special training on the operator's part. A Mathematica simulation was created to have a 4WID/4WIS vehicle respond to the input from the SpaceMouse. SpaceMouse's inputs were integrated coherently by modeling the vehicle's movement as a motion along a circular path, where a straight-line motion is calculated as a motion along a circle of very large radius. Each wheel's direction was pointed to the tangential direction of that circle, and each wheel's speed was set to be proportional to their distance from the center, allowing it to have no slip/skid. The eight parameters provided with this algorithm were then tested on a 4WIS/4WID prototype to determine the algorithm's accuracy. Modeling basic motion, including straight line motion, as a travel along a concentric circular path was successful, greatly simplifying the complexity of the algorithm by making one algorithm control all motions as circular motions of different parameters.

2 Table of Content

3 Introduction

3.1 Problem Statement

4WIS/4WIS (4 Wheel Independent Steering/Driving) is a steering system for a four-wheeled vehicle that allows for separate speed and direction controls for each wheel. These controls enable more versatile motion for vehicles that needs to navigate tight spaces, such as a wheelchair in a narrow subway cars.

For example,

Figure 1 below shows a scenario when a wheelchair needs to move from the starting point to the destination. In a 2 Wheel Steering system, a significant space (colored purple) is needed to get to the destination, but 4WIS/4WID requires much less space (colored brown) to get to the destination.

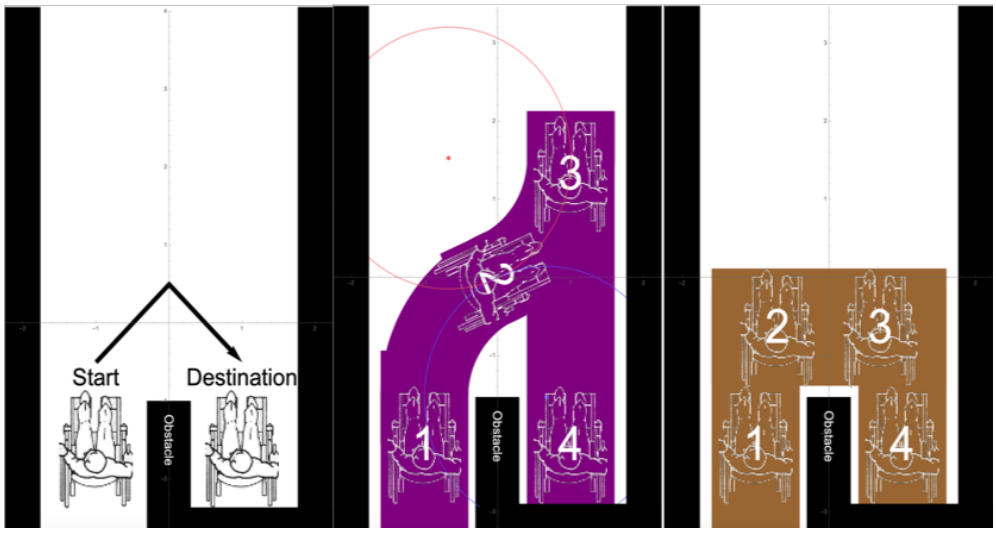


Figure 1 Wheelchair Scenario

Each wheel requires direction and speed, making them altogether eight parameters that need to be controlled coherently. To our knowledge, neither a keyboard nor a joystick, including a dual joystick, can fully convey the motions that 4WIS/4WID is capable of. This is because it cannot convey the direction, rotation(without this rotation, the vehicle can only slide always facing one direction) and speed simultaneously. In case of the keyboard, it can send a finite pre-determined direction and speed assigned to each key. Many keys could be assigned to a variety of direction, rotation and speed combination, approximating continuous control. However, the position of the keys would not be intuitive which will hamper timely response. Using two joysticks could be a solution where one is dedicated to the direction and speed, and the other one is dedicated to rotation. However, this would require two hands and making the control more complex and confusing for the user.

3.2 Objective

Our goal is to find a controller that can easily convey the driver's intention and develop an algorithm that can perform all motions possible in a 4WID/4WIS vehicle. This algorithm will need to interpret the driver's intention and control all four wheels in a cooperative manner so that they don't conflict with each other while accomplishing the intended motion. This development will allow for an intuitive maneuver through space that is difficult to navigate in a conventional two-wheel steering vehicle.

3.3 Literature Review

Different types of omnidirectional robots/vehicles have been developed for the last few decades. For indoor warehouse usage, KUKA OmniMove is a mobile, flexible transport platform for heavy loads. [1] The researchers in KUKA used multiple Mecanum wheels, one of an

omnidirectional wheel that consists of several rollers mounted at a 45-degree angle to allow full freedom of movement and improved maneuverability. To briefly talk about Mecanum wheels, they are conventional wheels with a series of rollers attached to their circumference. These rollers typically each have an axis of rotation at 45° to the plane of the wheel and at 45° to a line through the center of the roller parallel to the axis of rotation of the wheel. This structure allows them to easily move vehicles in any direction. However, because of the way these rollers are attached, this technology seems extremely vulnerable in bumpy terrains. To work properly, all the Mecanum wheels have to be touching the ground to work which is very impractical for them to function properly on off-road situation. Additionally, it is hard for them to change direction while they are on high speed. Because the key part of this technology is based on the rollers, it is hard for users to handle high speed, which is crucial to road vehicles.

Another approach to omnidirectional robots is the 4WIS (Four Wheel Independent Steering) / 4WID (Four Wheel Independent Driving) System. The 4WIS/4WID vehicle has the advantage that the rotation angle and driving torque of each wheel can be independently and accurately controlled. [2] There have been many attempts to develop a 4WIS/4WID vehicle and improve the conventional Ackerman steering mechanism. One of the teams has integrated a novel steering system consisting of three parts: an improved two-front-wheel steering (2FWS) mechanism, an omnidirectional independent steering (OIS) mechanism integrated with steer-by-wire, and a control strategy for the space-saving steering system of an electric vehicle. [3] Other researchers made attempts to improve tracking performances by applying a Coaxial Steering Mechanism. [4] The Coaxial Steering Mechanism is composed of two steering joints on the same axis, and was applied to a predictive steering system that consisted of two controllers. Washington State University used 4WIS steering system for their robotic agricultural equipment.

[5] They implemented four different strategies to allow their equipment to maneuver freely in a confined workspace: Ackermann Steering, Active Front and Rear Steering (AFRS), Crab Steering, and Spinning.

3.4 Steering Modes in the Algorithm

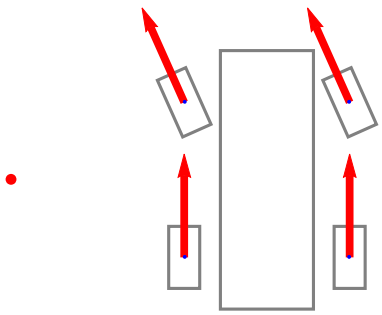


Figure 2 Ackerman Steering

Ackerman Steering:

The Ackermann steering mechanism, a mechanism that is generally used for four-wheel vehicles is a geometric arrangement of linkages in the steering of a vehicle designed to turn the inner and outer wheels at the appropriate angles. [6]

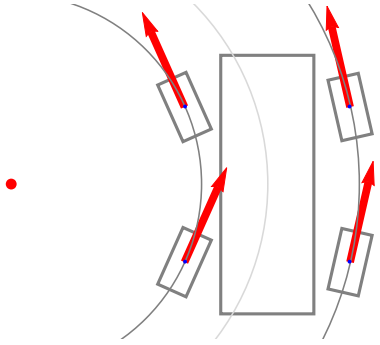


Figure 3 Active Front and Rear Steering

AFRS (Active Front and Rear Steering):

An AFRS mechanism involves front and rear wheels turning independently for a smaller turning radius and better cornering stability. In this implementation, even the front wheels are controlled separately because they have different tangential direction, and speed. When the vehicle is moving slowly, the rear wheels turn in the opposite sense of direction from the front wheels to improve maneuvering performance. At high velocity, the rear wheels turn in the same sense of direction as the front wheels to reduce yaw and improve stability. [7]

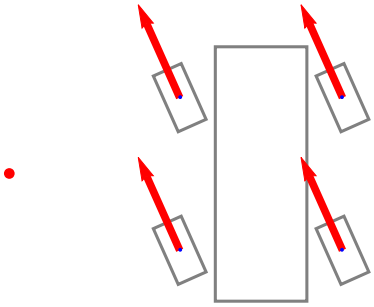


Figure 4 Crab Steering

Crab Steering:

A Crab steering mechanism is a special type of active four-wheel steering. It operates by steering all wheels in the same direction and at the same angle. This allows the vehicle to "translate" or "glide" to any direction at any time. This action is advantageous to the vehicle while changing lanes on a high-speed road. The elimination of the centrifugal effect and in consequence the reduction of body roll and cornering force on the tire, improving the stability of the car so that control becomes easier and safer. [8]

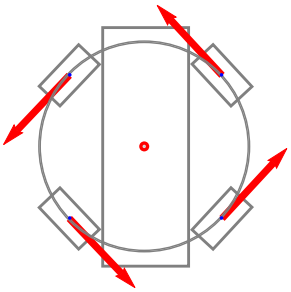


Figure 5 Spinning

Spinning:

Spinning is easily accomplished by turning all wheels perpendicular to the center diagonal line and turning in the same direction.

While the conventional steering system involves Ackerman Steering, only a 4WIS/4WID is capable of three different steering mechanisms: AFRS, Crab Steering, and Spinning. Our algorithm accomplishes all three steering mechanisms and allows the user to simultaneously control all four wheels by using one algorithmic framework of circular motion. All parameters involved in three mechanisms are computed the same way. Only the circular motion's center and its radius values are at a different, yet continuous, uninterrupted range. This approach ensures smooth navigation control without any mode changes.

4 Materials

4.1 Vehicle Description - Orbitron

The test vehicle, labeled as Orbitron, consists of four spherical wheels connected to the axle of a wooden frame, which is attached to the rigid main body. The whole vehicle is about 580mm wide × 109.3mm long, and weighs approximately 15 kg. Each of the frames can perform a maximum pan rotation of 560° with an HS-785HB multi-rotational Servo Gearbox (5:1 Ratio), and the 170-rpm Econ Gear motor attached to the axle provides stall torque of 22.04 kg × g-cm. The spherical wheel is a 60mm diameter ball made of heat-sealed, solid, closed-cell EVA foam, specifically made for Myofascial Release Therapy. Each wheel frame can freely rotate while still having a fixed axle that rotates the wheel to drive the vehicle. The rigid body is made of Foamex board for more rigidity with less weight and supported with a ¾″ PVC pipe fixed with machine screws, washers, and nuts.

Each wheel frame can be controlled independently, allowing the vehicle to change direction with fixing its front head, go around in a circle, and move in a horizontal direction when desired.

The vehicle itself is controlled by an Arduino Mega 2560 Board, a microcontroller with an open-source electronic platform based on easy-to-use hardware and software. Before the development of this algorithm, we have developed a custom Window Forms application with C# which was used to manually control the vehicle through keyboard input. This program could send characters based on user's input such as pressing a certain key. The layout of this software can be seen in Figure 9, and the interface of the program is shown in Figure 8. On the receiver side, an xBee Wireless shield, an Arduino-compatible shield that uses an xBee Wireless module, was attached on top of the board to receive the signals from the C# application.

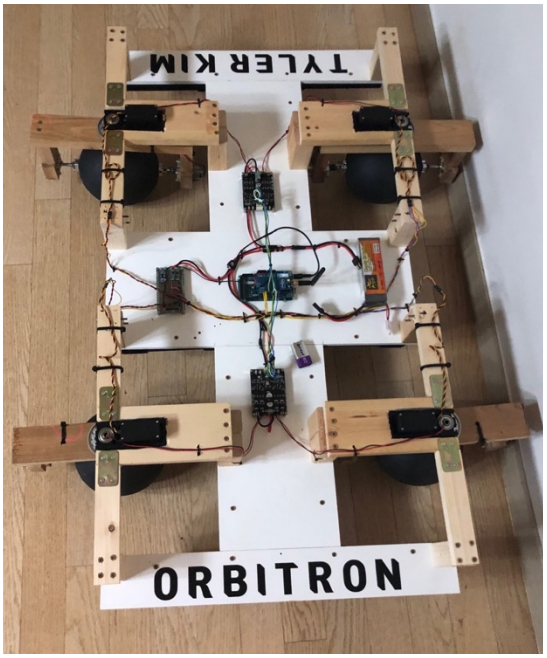


Figure 6 Orbitron (From above)

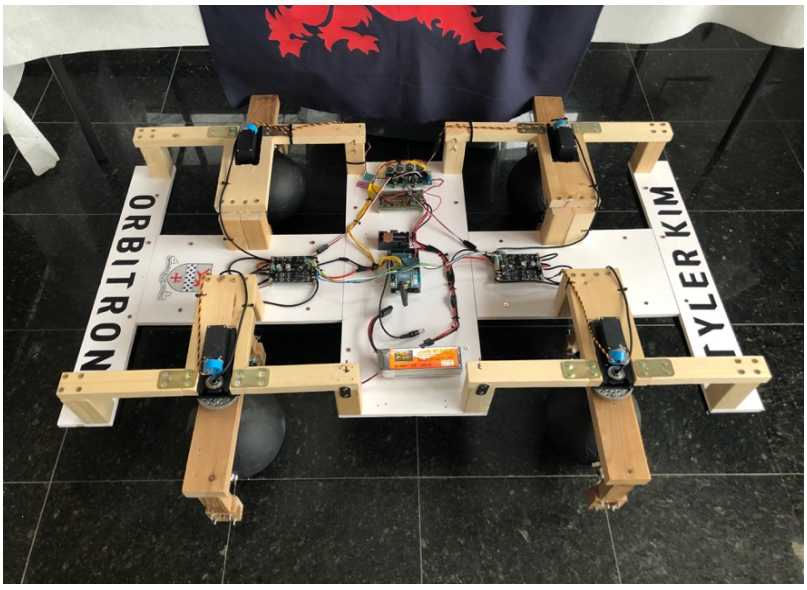


Figure 7 Orbitron (From left)

Component	Part Number	Manufacturer	Quantity
TM-785HB SERVO GEARBOX	TM-785HB-5-U	HighTech, Servocity	4
170-RPM Econ Gear Motor	638354	Actobotics	4
0.250" (1/4") x 10.00" Stainless Steel Precision Shafting	634176	Servocity	4
LiPO 3s1p 5000mAh Battery	X	X	1
Cytron Dual Channel 10A DC Motor Driver	MDD10A	Cytron Technologies	2
MP1584 Step Down 5A Regulator Module	MP1584	Monolithic Power Systems (MPS)	1
AMS1117-5V Step Down 800mA Regulator Module	EP-AMS1117-5V	Advanced Monolithic Systems, Inc	1

Table 1 Basic Components of Orbitron

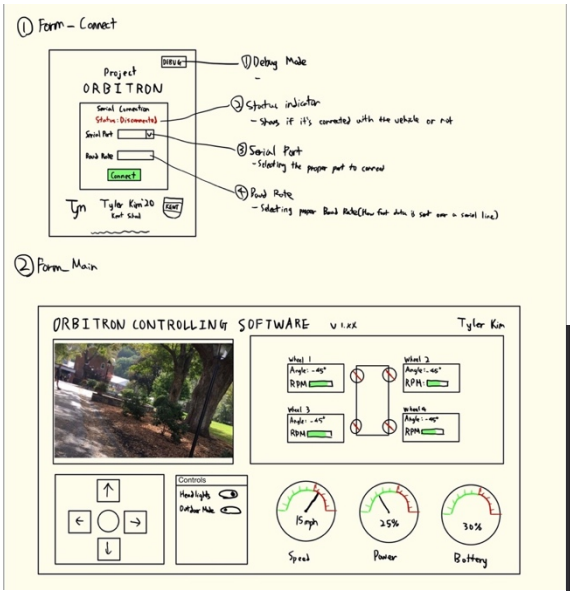


Figure 9 Software Layout

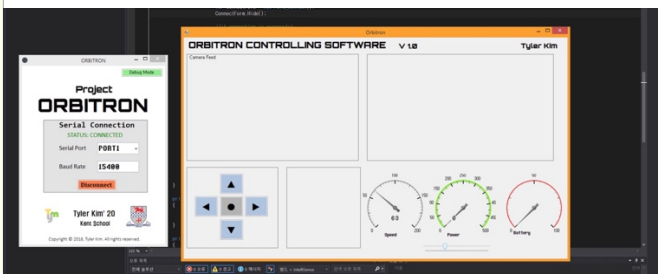


Figure 8 Orbitron Controlling Software

4.2 Controller

https://goo.gl/images/yWSNtb

Figure 10 SpaceMouse™

A 3Dconnexion's SpaceMouse™ involves 3Dconnexion's patented 6-Degrees-of-Freedom (6DoF) sensor that is specifically designed to manipulate digital content and camera positions in the industry-leading CAD applications. [9]

A SpaceMouse will convert the input from the user into a set of six numbers that range from −1 to +1. For example, one output created during the experiment was: $\{ 0.234, -0.112, 0.066, -0.422, -0.208, 0.112 \}$.

4.3 Computer Languages

Mathematica, a symbolic mathematical computation program, was used for the navigation algorithm creation and execution. It was also used to simulate the resulting motion by using its extensive real-time 3D graphics capabilities as well as its ability to read in the SpaceMouse input values (set of six numbers) seamlessly without any additional driver installed. C# was used for developing the Orbitron-Controlling software and the Serial Regulator.

4.4 Processor

A MacBook (15-inch, 2017 / 2.9 GHz Intel Core i7 / 16 GB 2133 MHz LPDDR3) was used for algorithm execution. An Arduino (MEGA 2560 REV3 - ATmega2560) was used for motor control.

5 Variables

5.1 Experiment 1: Algorithm

The independent variables will be the variables provided by the SpaceMouse. The SpaceMouse has 12 total built-in variables shown in Figure 13.

$$\{X, Y, Z, X1, Y1, Z1, X2, Y2, Z2, X3, B1, B2\}$$

However, {X, Y, Z} and {X1, Y1, Z1} share the same value, so there are only nine total independent variables that are read through the SpaceMouse's built-in driver.

$$\{X, Y, Z, X2, Y2, Z2, X3, B1, B2\}$$

The explanation for each variable is shown in Table 2.

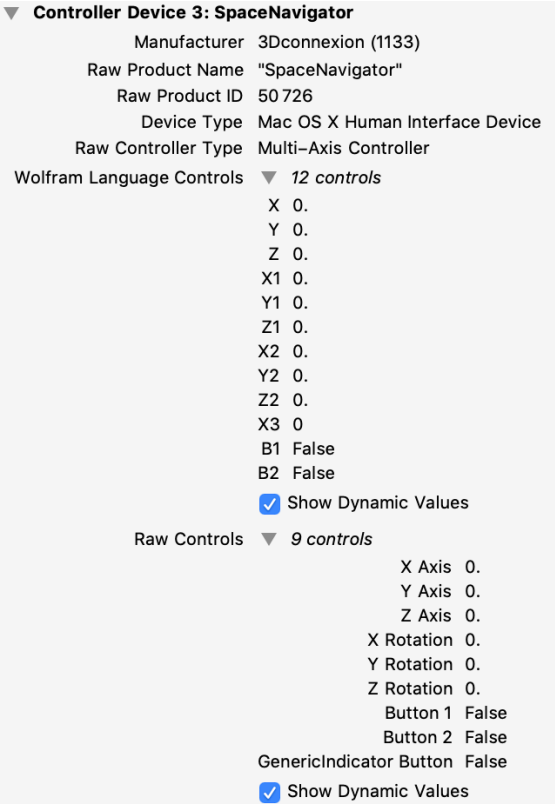


Figure 13 SpaceMouse's Variables

X	X Axis; Moving the mouse on a plane (X)
Y	Y Axis; Moving the mouse on a plane (Y)
Z	Z Axis; Lifting/Pressing the mouse
X2	X Rotation; Tilting the mouse (Front-Back)
Y2	Y Rotation; Tilting the mouse (Left-Right)
Z2	Rotation; Twisting the mouse
B1	Button 1; Boolean value for the left button
B2	Button 2; Boolean value for the right button

Table 2 Meaning of SpaceMouse's Variables

The dependent variables here are the resulting eight variables generated by the algorithm.

5.2 Experiment 2: Driving Orbitron

The independent variables in the second experiment will be the dependent variables from the first experiment: eight variables generated by the algorithm.

The dependent variables will be the resulting navigation of the Orbitron.

5.3 Controlled Variables

The controlled variables in this experiment are the prototype vehicle (Orbitron) and the experiment environment.

6 Development Procedure

6.1 Algorithm Setup

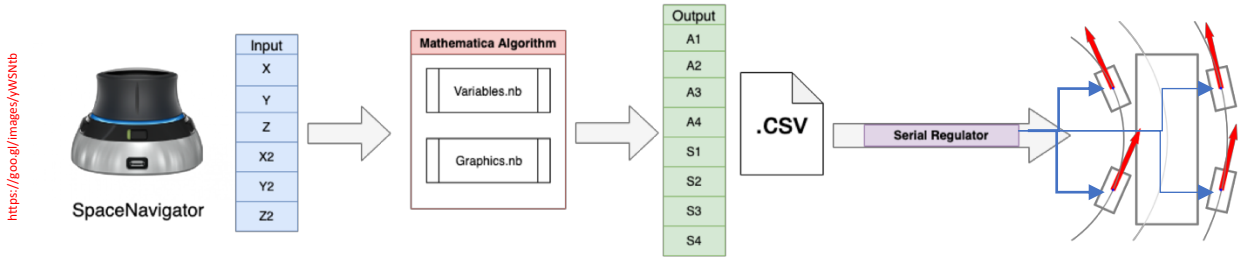


Figure 14 Algorithm Setup

The overall outline of the algorithm is shown in Figure 14 above. The algorithm's main job is to translate the six variables from the SpaceMouse into eight variables: speed and angle for each of the four wheels.

The algorithm is divided into two Mathematica Notebook files: Variables.nb and Graphics.nb. Variables notebook is evaluated first to calculate the variable transformations based on the input from the SpaceMouse. The Graphics notebook is then responsible for showing the vehicle's expected path and for recording the sets of timestamped variables in a CSV file. An example of the processed CSV file is shown in Figure 15 below.

	A	B	C	D	E	F	G	H	I	J
1	Time	Center Speed	Front Right Speed	Back Right Speed	Front Left Speed	Back Left Speed	Front Right Angle	Back Right Angle	Front Left Angle	Back Left Angle
2	1.9345489	0.00298023	0.991032	0.991032	1.00903	1.00903	-0.00805744	0.00808751	-0.00791371	0.00794324
3	2.0545949	0.0170007	0.994749	0.988652	1.01136	1.00537	-0.398408	-0.383491	-0.391503	-0.376792
4	2.1990434	0.0495891	0.995009	0.988559	1.01145	1.00511	-0.422288	-0.407524	-0.414994	-0.400427
5	2.3347072	0.0633635	0.994438	0.988774	1.01124	1.00568	-0.369336	-0.354245	-0.36291	-0.348036
6	2.4537884	0.0679353	0.993456	0.989236	1.01079	1.00665	-0.274732	-0.259163	-0.2699	-0.254581
7	2.5982255	0.0846261	0.992676	0.989698	1.01034	1.00741	-0.195127	-0.179266	-0.191672	-0.17608
8	2.7260825	0.0773817	0.992518	0.989804	1.01024	1.00757	-0.178387	-0.162478	-0.175225	-0.159588
9	2.8627233	0.0935052	0.995903	0.992968	1.00704	1.00414	-0.292942	-0.282975	-0.289608	-0.279741
10	3.0198568	0.155153	1.00674	1.01255	0.987482	0.993397	-0.318851	-0.335779	-0.325297	-0.342514
11	3.145761	0.113227	1.01589	1.03221	0.967932	0.985041	-0.34619	-0.387637	-0.364122	-0.407267
12	3.2638569	0.131455	1.03068	1.05755	0.942964	0.972259	-0.296947	-0.371069	-0.325542	-0.40544

Figure 15 Example of CSV file

While turning each wheel's direction happens instantly in the simulation, the actual prototype had a certain delay to achieve the desired angle. In order to consider the time difference in turning the wheels' directions, we developed a Serial Regulator which is capable of delivering set of variables at a proper time without overfeeding data to the prototype. This slight delay does not cause problems in actual use because humans can easily adjust to these types of variabilities. In the program, one can select a specific CSV file which the user wants to use to control the vehicle. Then the program calculates the time needed for Orbitron to apply each variable's values and waits for that amount of time before sending another signal. The interface of a Serial Regulator is shown in Figure 16.

c.

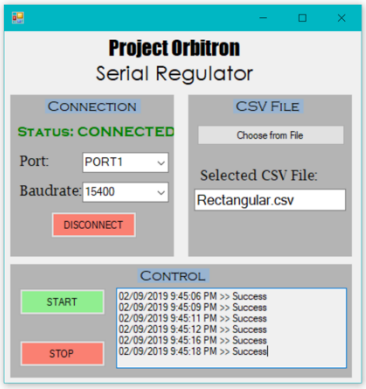


Figure 16 Serial Regulator

6.2 Preliminary Simulations

Before the algorithm development, there were several preliminary simulations developed to visualize how different steering modes actually work.

In the beginning, we developed a 3D simulation (Figure 17, Figure 18) with Mathematica to visualize how different steering modes actually work.

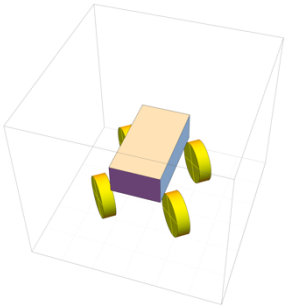


Figure 17 3D Simulation 2

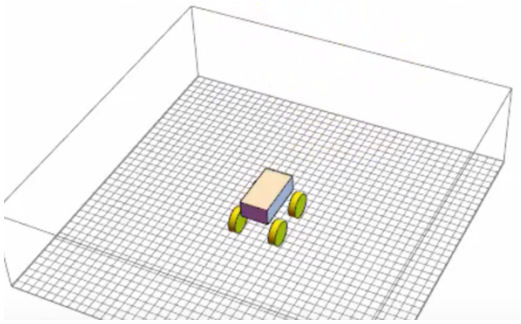


Figure 18 3D Simulation 1

After the development of a 3D simulation, another simulation was developed which calculated the radius of curvature depending on the twisting motion of a SpaceMouse. This later became a crucial step to develop the core concept of our algorithm: considering every motion to be a circular motion. Finally, we developed all simulations in 2D so that we could calculate each wheel's speed and angle values more accurately.

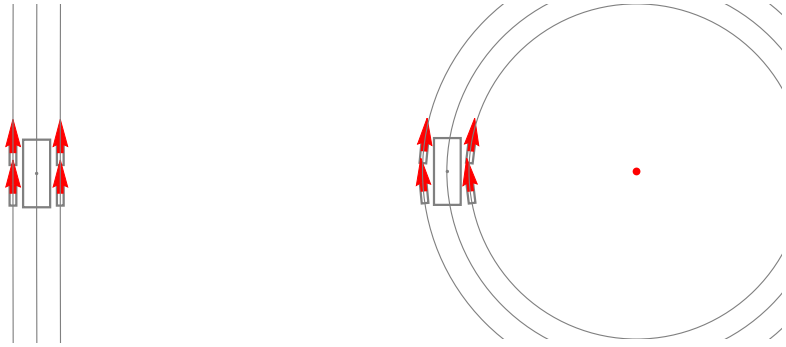


Figure 19 Radius of Curvature Simulation

6.3 Core Theory of the Algorithm

There are two core theories in this algorithm that simplified its development process. The first core concept is that every motion is considered to be a circular motion. With this concept, there is no need to separate straight motion and curve motion because straight motion can be considered to be a circular motion with an infinite radius of curvature.

The second core concept is that each wheel's angle and speed always depend on the circles. For the angle variables, the wheels' angle is always tangent to the circle. This will prevent wheels' motion to conflict with each other as well as allowing the user to accomplish desired motion easily.

$$\omega = \frac{\theta}{t}$$

Equation 1 Angular Velocity

$$v = r\omega$$

Equation 2 Linear Velocity

For the speed variables, the Equation 2 is used to simplify the computation. The velocity of each wheel (v) depends on the radius of the circle (r), radius of curvature in this case, and the angular velocity (ω). However, the center motion's speed is always normalized to 1, so the ω stays the constant. This means only the r determines the linear velocity for each wheel, so the wheel's speed is always a ratio to the center motion's speed as well as a ratio to the radius of a circle for vehicle's center. This greatly simplifies the entire algorithm's computation since only changing the center point and radius of the curvature circle from SpaceMouse's values can control the vehicle's versatile motions.

6.4 Simulation Interface

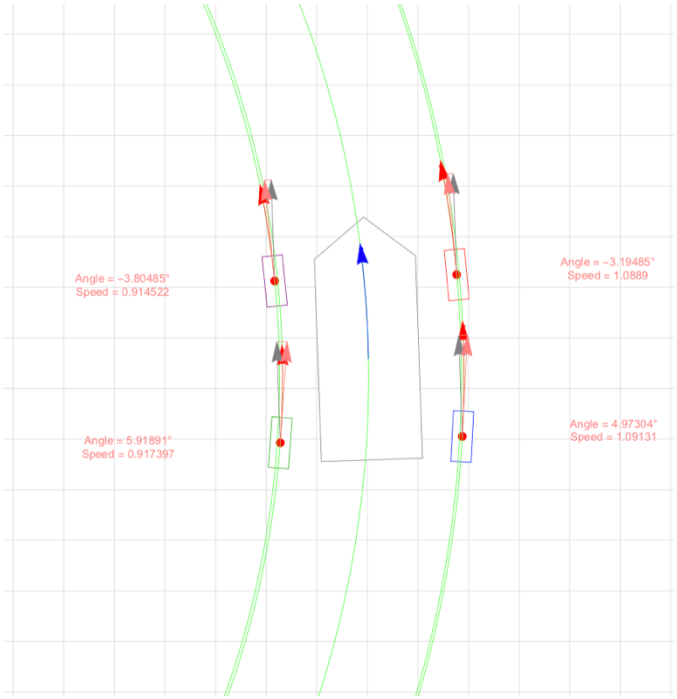


Figure 20 Graphics.nb Interface

When Graphics.nb is executed, the interface shown in Figure 20 will be continuously updated based on the user's input.

All the arrows shown in this interface is color coded differently to distinguish each from others easier, and four labels are shown next to each wheel which displays the updated angle and speed value. Additionally, there are always five different green circles that each represent the radius of curvature of each wheel and the center of the vehicle. This can be applied to vehicles with any number of wheels, but just with more curvature circles.

For the arrows, the blue arrow in the center of the diagram represents the motion of the center of the vehicle. Then there are three arrows in each wheel. The gray arrow is always fixed along the vehicle body's angle and acts as a base for other two arrows. The pink arrows are the tangent line of the green circle, and the angle between the gray arrow and the pink arrow is used to determine each wheel's angle. Lastly, the red arrow represents the actual trajectory of the vehicle. As the user controls the vehicle, the length of the red arrow will always represent the relative velocity of each wheel by differing its length.

6.5 Integration of Crab Steering Mode

Our algorithm uses the Crab steering mode whenever the user slides the SpaceMouse on the plane. In these examples shown in Figure 21, all four whetels are angled in the direction the mouse is pointing at. All four wheels are computed as the tangential velocity of a circle with a very large radius.

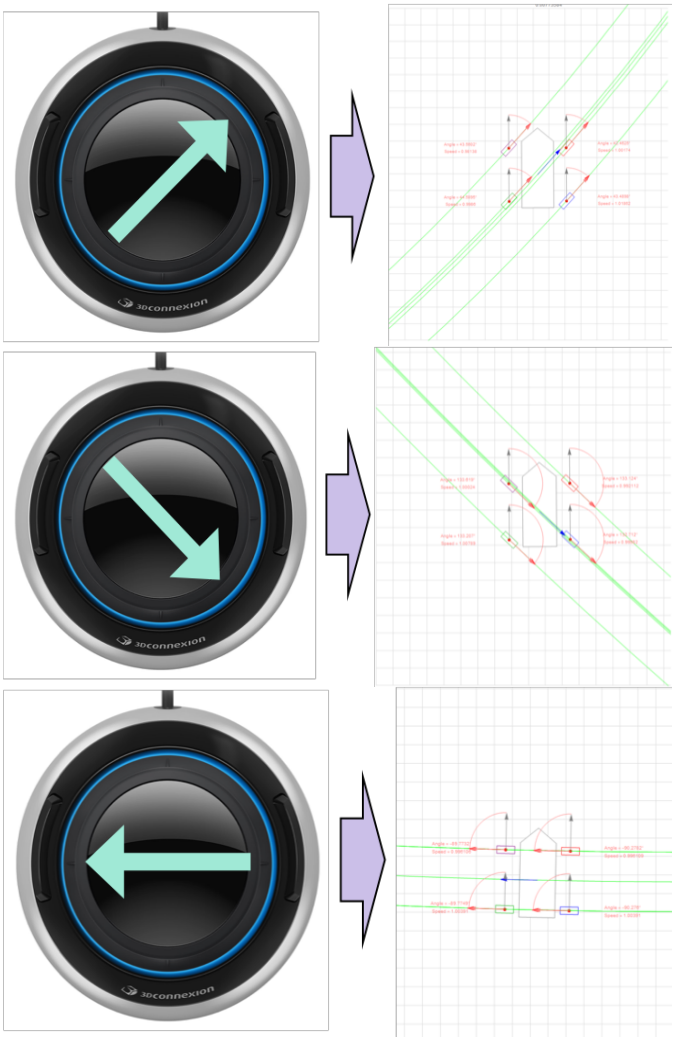


Figure 21 Crab Steering Simulations

6.6 Integration of AFRS Mode

$$r \propto \frac{1}{\theta}$$

Equation 3 Radius of Curvature

The twisting motion of the SpaceMouse is responsible for changing the radius of the vehicle's curvature. The core of our algorithm is that it considers every motion as a circular motion and computes each wheel's velocity and angle tangent to that circle. The radius of curvature (r) is calculated by the equation shown above, and the angle value (θ) is controlled with the twisting motion of the mouse.

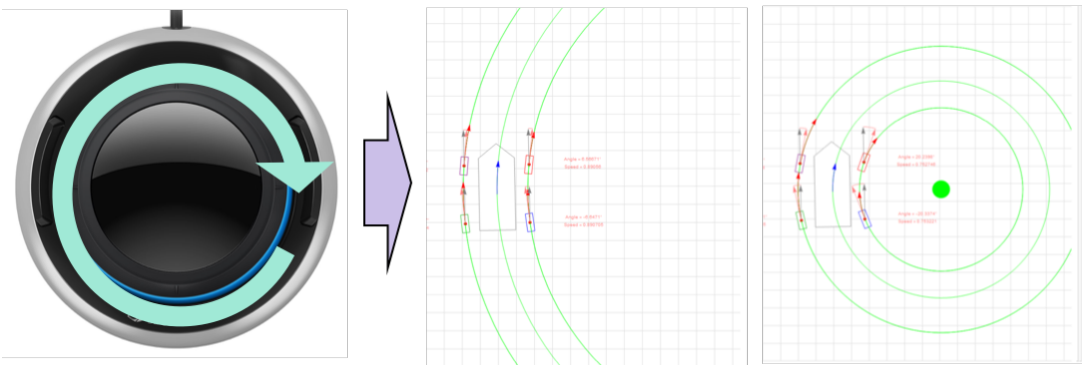


Figure 22 AFRS Simulation (Twisting clockwise)

As shown in Figure 22, when the mouse is twisted clockwise, and held, θ is continually increased, resulting in an even smaller curvature radius.

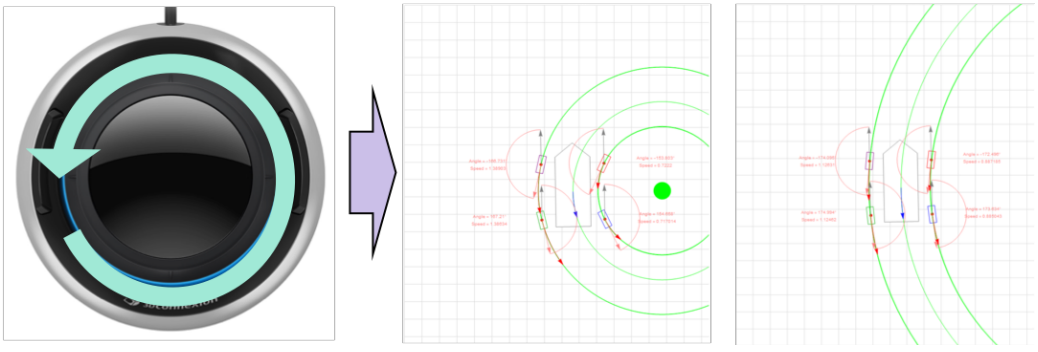


Figure 23AFRS Simulation (Twisting counterclockwise)

As shown in Figure 23, when the mouse is released from clockwise twist, θ moves toward zero, resulting in a larger curvature radius. Negative θ will make the center of the circle to move to the other side of the vehicle and repeat the same process.

This principle also applies to straight motion in scenarios such as Crab Steering mode. Setting the twisting of the mouse near center will make θ very small, causing the turning radius to be almost infinite. The algorithm detects this small value range, and assign the radius to be 10,000. This makes the vehicle's motion to still follow the circle but the radius is so large that the path can be considered a straight-line motion.

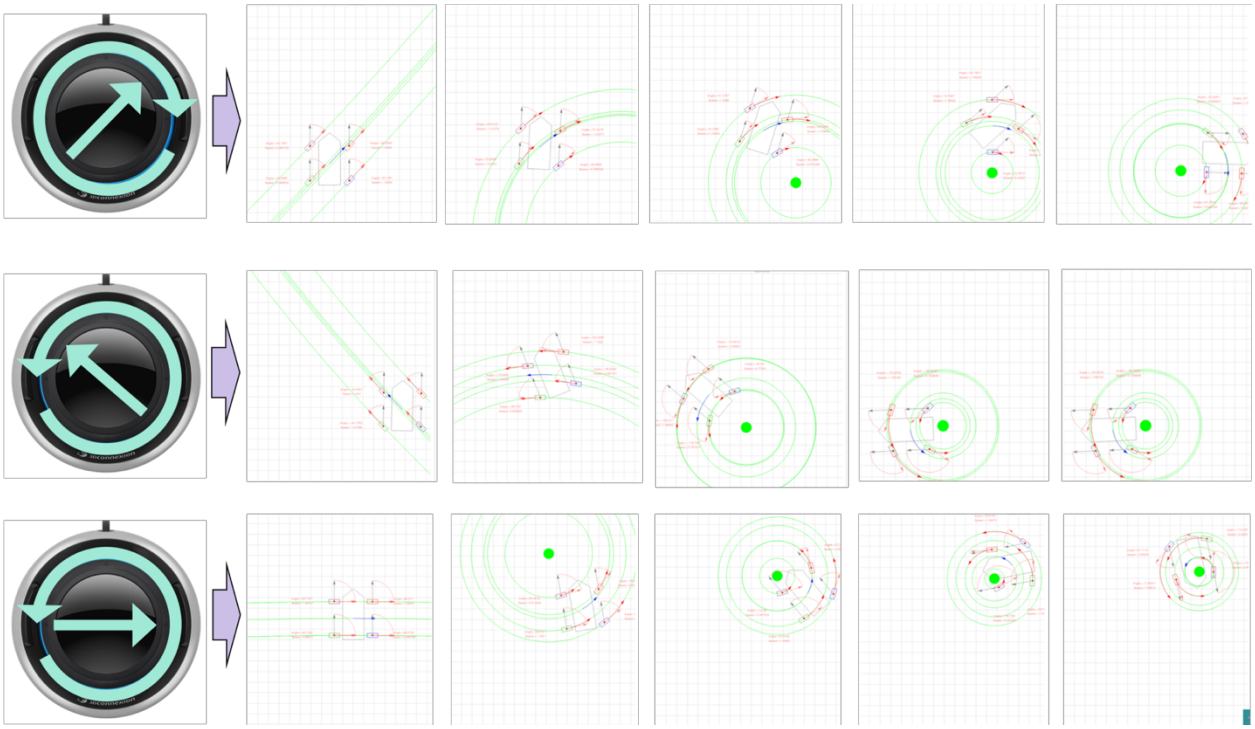


Figure 24 Crab Steering + AFRS

In Figure 24, straight motion and curving motion are accomplished simultaneously. When twisting and shifting the mouse occurs at the same time, the vehicle can perform more complicated motion, such as moving forward with a gradually decreasing turning radius.

6.7 Integration of Spinning Mode

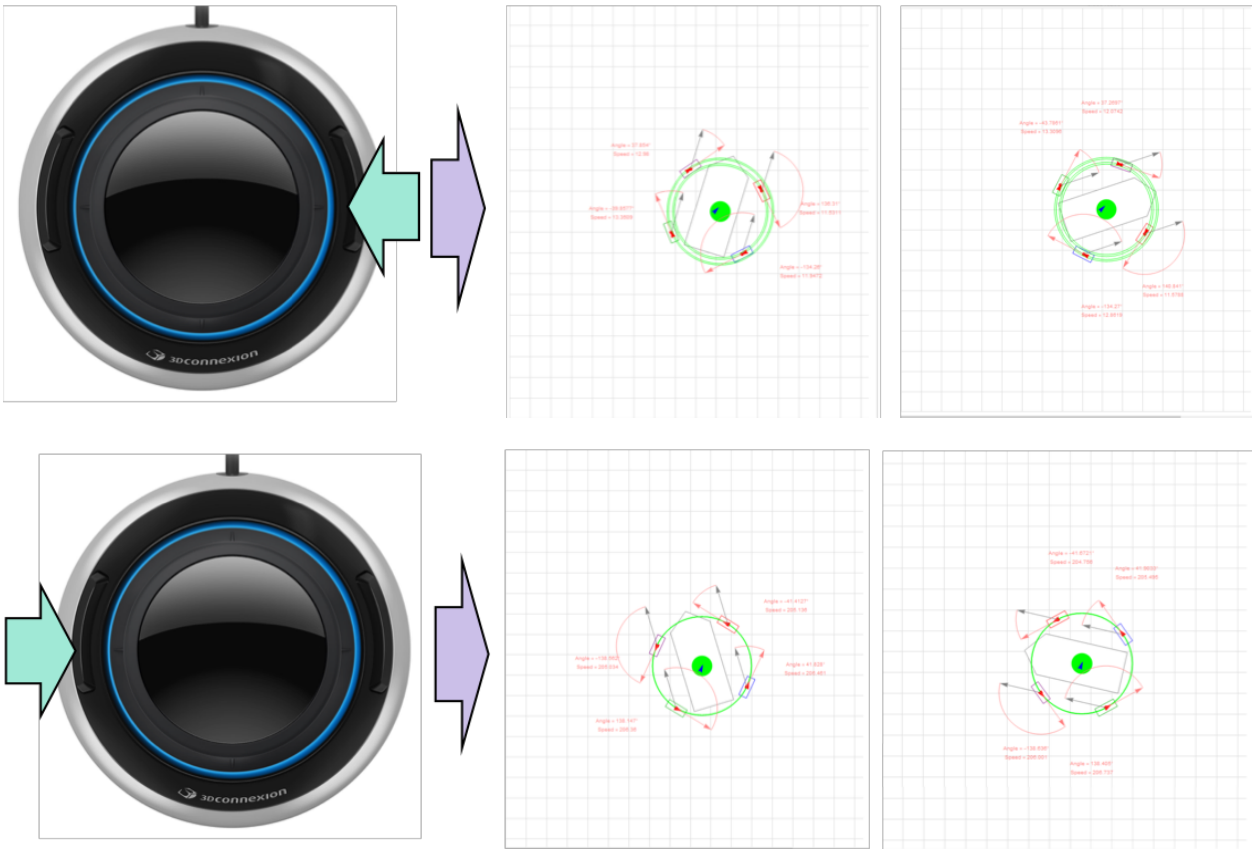


Figure 25 Spinning Simulation

The spinning motion can be executed by twisting the controller all the way, bringing the center of circular path to the center of the vehicle. However, it takes too much time. Therefore, a button-press shortcut was provided in addition to the "all paths are circular" frame of control.

When one of the two buttons on the side of the SpaceMouse is pressed, the vehicle rotates in the respective direction. This is accomplished by moving the center of concentric circles very close to the vehicle's center, making the turning radius very small. As shown in Figure 25, the vehicle turns in a clockwise direction when the right button is pressed, and the vehicle rotates in a counterclockwise direction when left button is pressed.

7 Results and Discussion

Figure 26 Pre-E center

In order to evaluate the algorithm's functionality, movement of the actual path of Orbitron was compared to the expected path from the algorithm. This experiment was held in Kent School's Pre-Engineering Center, with a setup shown in Figure 26. While the Orbitron was controlled using the presented algorithm, two different cameras filmed the Orbitron's movement to accurately determine its movement. One camera was an iPhone attached to the extension pole filming Orbitron from the second floor. Shown in Figure 28, this allowed us to have an aerial view of the Orbitron's movement. Another camera was an iPad set on the table to film it from a ground level. During video analysis, combination of the two perspectives was greatly helpful in evaluating movement of Orbitron.

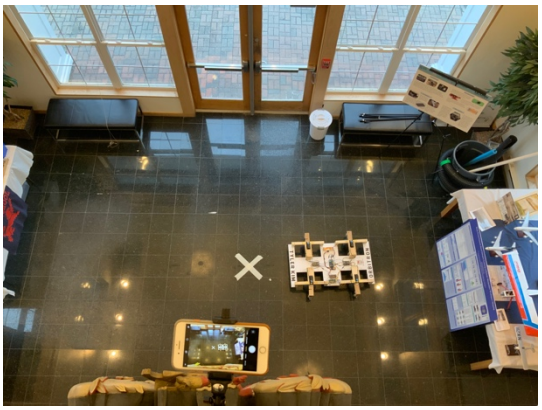


Figure 28 Camera 1 (iPhone)

Figure 27 Camera 2(iPad)

7.1 Rectangular Path

First experiment was to use the crab steering mode to make the vehicle move in a rectangular path without changing the direction Orbitron is facing. All four wheels always turned together in the corner of a rectangle and the speed was constant for all four wheels. The sequence of screenshots from video footage and simulation is shown in Figure 29.

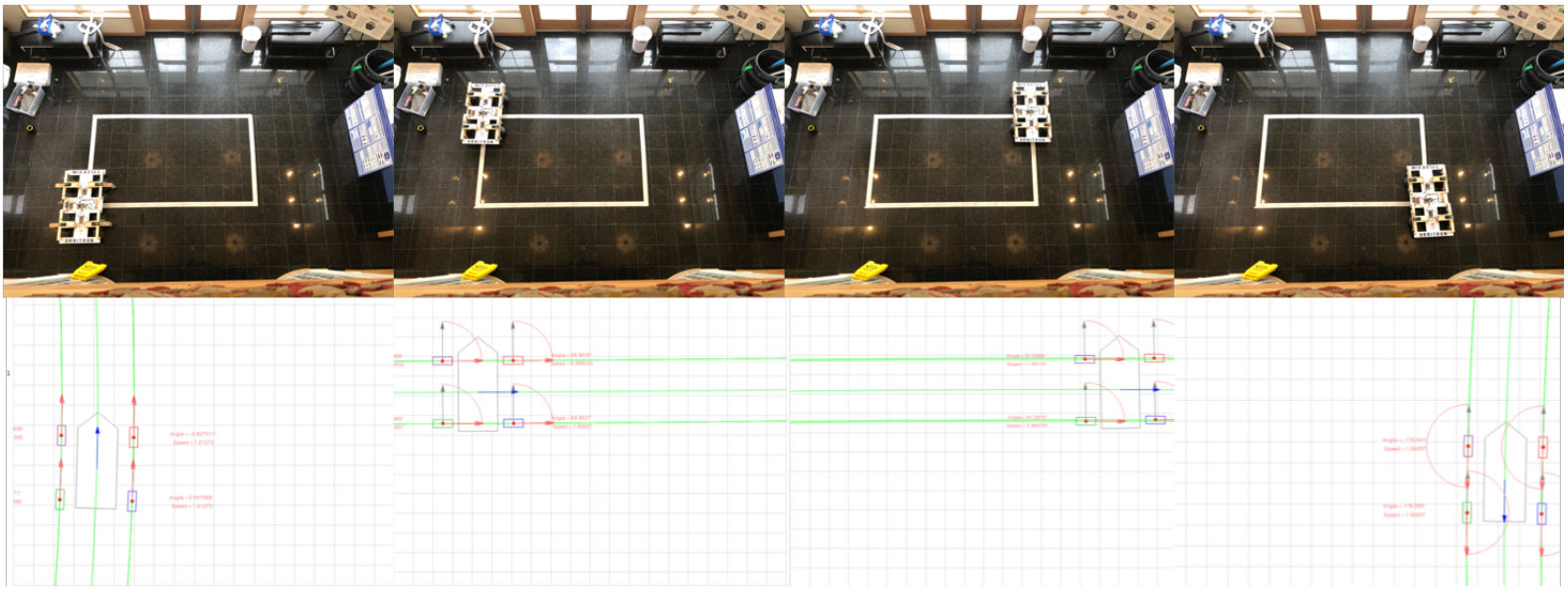


Figure 29 Rectangular Path

While the Orbitron was following the variables generated by the algorithm, radius of curvature was constantly considered to be very large, leading the Orbitron to always follow a straight motion. Only the center of the curvature moved to the perpendicular direction to the motion.

7.2 Circular Path

Second experiment was to use AFRS mode to drive the vehicle in a circular path, but facing one direction at all times. Each wheel was fixed in an angle that is tangent to the turning

direction, and different speed values were given for each wheel to account for the different distance from the center.

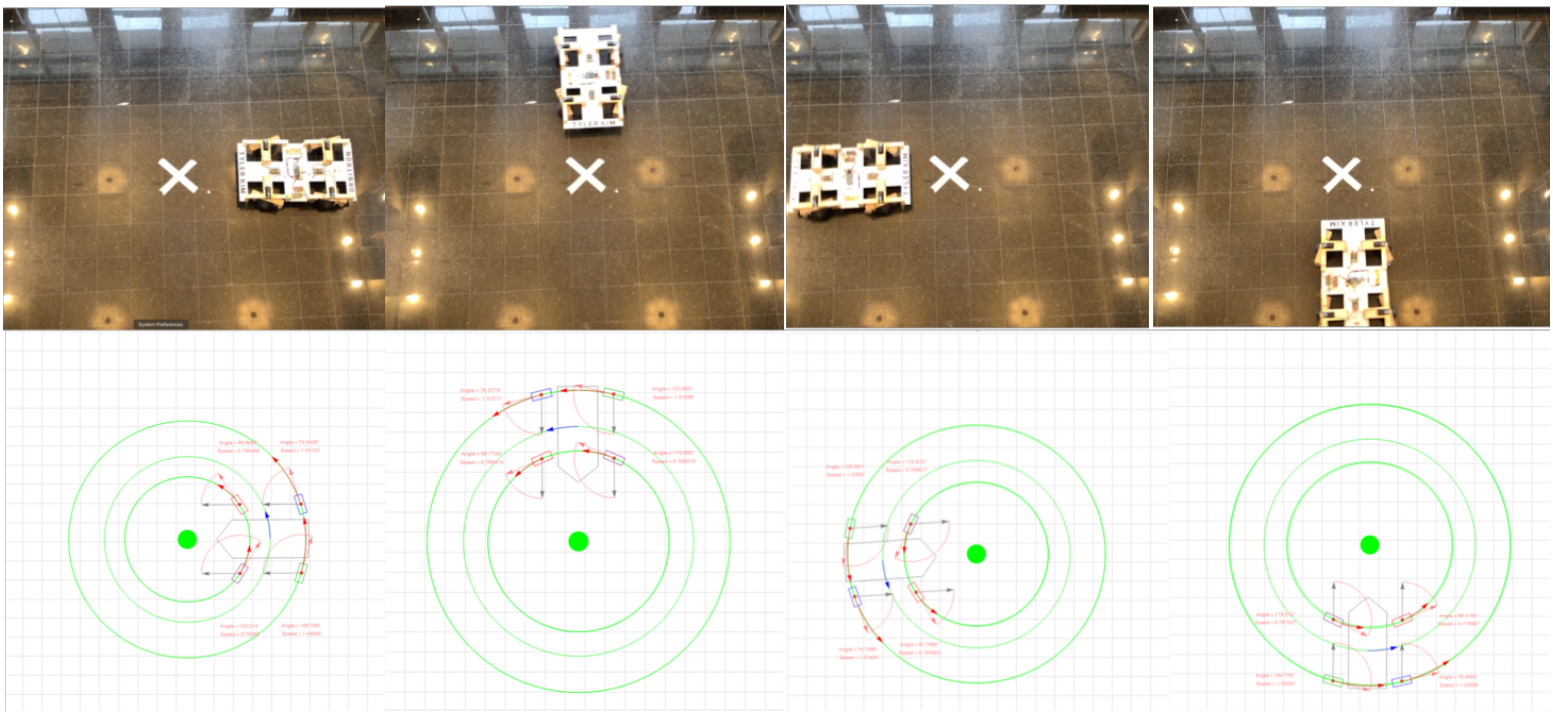


Figure 30 Circular Path

In the motion shown in Figure 30, the angle of each wheel was set to be tangent to the green circle in the simulation screenshot. While the Orbitron was following this circular path, each wheel's angle relative to the body did not change.

For the velocity of each wheel, the velocity of the wheels in outer circle was set to be faster than the velocity of inner wheels to prevent slipping during the movement. This was simple to calculate because each wheel needs to move at a speed proportional to the radius from the circle's center.

7.3 Spinning

Third experiment was to use the Spinning mode to rotate the vehicle with zero radius. It is similar to the circular path experiment, but the center of curvature was set very close to the center of the vehicle rather than outside of the vehicle. As a result, all wheels were set to be tangent to the turning circle and speed was constant while turning.

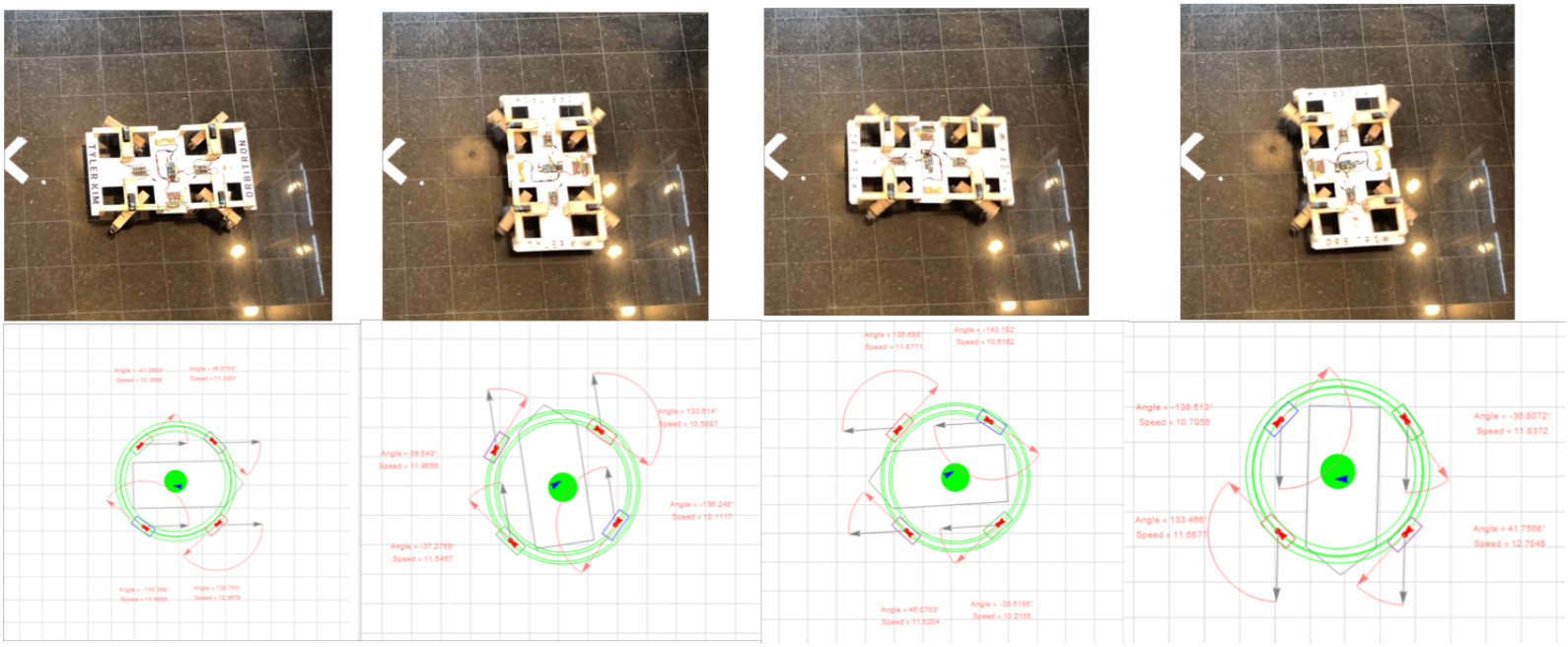


Figure 31 Spinning

8 Conclusion

We developed an algorithm, implemented in Mathematica, that allows for a full realization of a 4WIS/4WID vehicle's capabilities without any special training on the operator's part. Our work shows that the "all motions are circular" theory of motion can successfully and smoothly control a 4WIS/4WID vehicle simply by using the center of the circle of motion as a reference for direction and speed of each wheel. This unifying theory can accommodate any number of wheels since all wheels have to be at a certain distance from the center and must move in a tangential direction, automatically making the computation simple and elegant. We built a prototype vehicle, which we named Orbitron, and experimentally confirmed that our algorithm successfully processes the driver's intention conveyed by a SpaceMouse and controls all four wheels in a cooperative manner so that they don't conflict with each other to accomplish the intended motion. With the presented algorithm, attempts to apply the theory of "all motions are spherical" to 3D motions for more versatile can be made in future works.

9 Acknowledgements

I would like to thank my Calculus teacher and advisor Dr. Ben Nadire for his guidance and support throughout the entire research process. As the director of Kent School's Pre-Engineering Department, he also provided the workspace to build the prototype vehicle Orbitron and provided access to the Pre-Engineering Center to conduct experiments and collect data.

I sincerely thank Mr. Saxton for his AP Computer Science A class and for proofreading this report. His AP Computer Science A class not only made me more interested in coding, but also led to improve my computational problem-solving skills which greatly influenced the algorithm development. Additionally, feedbacks from someone who are involved in computer science field were crucial to make this report logically clear and understandable.

Last but not least, I would like to thank one of my truest friends, Dong Min Lee. When I first met him in an engineering summer camp, he was a brilliant leader who were seriously involved in computer science and engineering. His great passion eventually led me to become interested in engineering, and I believe this research would have not been possible if I never met him. Unfortunately, on April 16th 2014, Dong Min was one of the 304 passengers who were not rescued in the greatest tragedy in South Korea: Sinking of MV Sewol. In memory of Dong Min, I have decided to present this guild paper on April 16th 2019, five years after the tragedy. I will always remember 🎗.

10 Bibliography

[1] "KUKA," KUKA, 2014. [Online]. Available: https://www.kuka.com/en-au/technologies/omnimove-drive-technology. [Accessed 2019].

[2] H. Zheng and S. Yang, "A Trajectory Tracking Control Strategy of 4WIS/4WID Electric Vehicle with Adaptation of Driving Conditions," *Applied Science,* vol. 9, no. 1, 2019.

[3] Z. Z. e. al, "an improved two-front-wheel steering (2FWS) mechanism, an omnidirectional independent steering (OIS) mechanism integrated with steer-by-wire, and a control strategy for the space-saving steering system of an EV," *IEEE Transactions on Intelligent Transportation Systems,* vol. 18, no. 1, 2016.

[4] M. e. a. Itoh, "A Hierarchical Model Predictive Tracking Control for Independent Four-Wheel Driving/Steering Vehicles with Coaxial Steering Mechanism," *MOVIC(International Conference on Motion and Vibration Control),* 2016.

[5] Y. e. a. Ye, "Steering Control Strategies for a Four-Wheel-Independent-Steering Bin-managing Robot," *IFAC,* 2016.

[6] J.-S. e. a. Zhao, "Design of an Ackermann-type Steering Mechanism," *Journal of Mechanical Engineering Science,* Vols. 203-210, 2013.

[7] B. Fijalkowski, Automotive Mechatronics: Operational and Practical Issues Volume II, Springer.

[8] P. Amol and B. Nandkishor, "Study of Four Wheel Steering Mechanism," *IJMER,* vol. 6, no. 10, 2016.

[9] "3dconnexion," 3dconnexion, [Online]. Available: https://www.3dconnexion.com/spacemouse_compact/en/. [Accessed 2019].

11 Appendix: Source Code

Master 4WD/4WS vehicle control algorithm v4 - Variables.nb

This file contains 2D representation of vehicle control with SpaceMouse. In this construction, all dimensions will be parameterized, that is, everything will be done with with variables so that they can be changed any time.

phy prefix is for physical dimensions

```
In[•]:= phyCarHalfWidth = 1; phyCarHalfLength = 2; phyGapBetweenCarAndWheel = 0.8;
phyWheelX = phyCarHalfWidth + phyGapBetweenCarAndWheel;
phyWheely = 0.8 phyCarHalfLength;
phyWheelHalfThickness = 0.2;
phyWheelRadius = 0.5;
phyDrivingFieldX = phyDrivingFieldY = 10;
phyDrivingFieldGridLineGap = 1;
phyWheelFrontRight = {phyWheelX, phyWheely};
phyWheelBackRight = {phyWheelX, -phyWheely};
phyWheelFrontLeft = {-phyWheelX, +phyWheely};
phyWheelBackLeft = {-phyWheelX, -phyWheely};
```

Current moving directions relative to its vehicle's coordinate system, i.e., from the driver's point of view. Directions are specified with unit vectors.

```
In[•]:= dirBody = {0, 1};
```

These are variable for the current positions

```
In[•]:= posCurveCenter = {-1000, 0};
posBody = {0, 0};
posWheelFrontRight := RotationTransform[rotBody, {0, 0}][phyWheelFrontRight];
posWheelBackRight := RotationTransform[rotBody, {0, 0}][phyWheelBackRight];
posWheelFrontLeft := RotationTransform[rotBody, {0, 0}][phyWheelFrontLeft];
posWheelBackLeft := RotationTransform[rotBody, {0, 0}][phyWheelBackLeft];
```

Rotations. Global angle in the frame of reference of the background grid. This is not needed in actual driving but is needed to show the motion correctly. Rotations are specified with angles in radian

```
In[•]:= rotBody = 0;
```

col prefix is for colors

```
In[•]:= colCarBody = White; colCarEdge = Gray;
colWheelBody = White; colWheelEdge = Black;
colVelocityArrow = Red; colCurve = Green;  colGridLines = LightGray;
```

parameters These are each wheel's angle relative to the main body, and each wheel's speed to be sent to the motors.

```
In[•]:= angleFrontRight = 0; angleBackRight = 0; angleFrontLeft = 0; angleBackLeft = 0;
speedFrontRight = 0; speedBackRight = 0; speedFrontLeft = 0; speedBackLeft = 0;
```

combined commands
pos is Position

In[•]:=
```
comBackground =
  {
   GridLines -> {Table[i,
      {i, -phyDrivingFieldX, phyDrivingFieldX, phyDrivingFieldGridLineGap}], Table[
      i, {i, -phyDrivingFieldY, phyDrivingFieldY, phyDrivingFieldGridLineGap}]},
   GridLinesStyle → Directive[colGridLines],
   PlotRange → {phyDrivingFieldX {-1, 1}, phyDrivingFieldY {-1, 1}},
   ImageSize → 800
  };
Clear[comCar]

comCar[posBody_, dirBodyAngle_] := Module[{},

   {EdgeForm[colCarEdge], colCarBody,

    Rotate[

     (*Rectangle[-{phyCarHalfWidth,phyCarHalfLength}+posBody,
      {phyCarHalfWidth,phyCarHalfLength}+posBody],*)
     Polygon[# + posBody & /@ {{-phyCarHalfWidth, -phyCarHalfLength},
        {-phyCarHalfWidth, phyCarHalfLength},
        {0, 1.4 phyCarHalfLength}, {phyCarHalfWidth, phyCarHalfLength},
        {phyCarHalfWidth, -phyCarHalfLength}}],
     rotBody, posBody]

    }
  ];
```

```
In[•]:= Clear[comWheels]
comWheels[posBody_, dirBodyAngle_, posCurveCenter_] :=
 {
  dirFrontRight = ArcTan @@ (posCurveCenter - posWheelFrontRight) + If[z2 > 0, Pi, 0];
  dirBackRight = ArcTan @@ (posCurveCenter - posWheelBackRight) + If[z2 > 0, Pi, 0];
  dirFrontLeft = ArcTan @@ (posCurveCenter - posWheelFrontLeft) + If[z2 > 0, Pi, 0];
  dirBackLeft = ArcTan @@ (posCurveCenter - posWheelBackLeft) + If[z2 > 0, Pi, 0];
  (* common to all wheels *)
  EdgeForm[colWheelEdge], colWheelBody,

  (*front right*)

  EdgeForm[Red],
  Rotate[Rectangle[
    posWheelFrontRight + {phyWheelHalfThickness, -phyWheelRadius} + posBody,
    posWheelFrontRight + {-phyWheelHalfThickness, phyWheelRadius} + posBody],
   dirFrontRight, posWheelFrontRight + posBody],

  (*back right*)
  EdgeForm[Blue],
  Rotate[
   Rectangle[posWheelBackRight + {phyWheelHalfThickness, -phyWheelRadius} + posBody,
    posWheelBackRight + {-phyWheelHalfThickness, phyWheelRadius} + posBody],
   dirBackRight, posWheelBackRight + posBody],
  (*front left*)
  EdgeForm[Purple],
  Rotate[
   Rectangle[posWheelFrontLeft + {phyWheelHalfThickness, -phyWheelRadius} + posBody,
    posWheelFrontLeft + {-phyWheelHalfThickness, phyWheelRadius} + posBody],
   dirFrontLeft, posWheelFrontLeft + posBody],

  (*back left*)
  EdgeForm[Darker[Green]],
  Rotate[
   Rectangle[posWheelBackLeft + {phyWheelHalfThickness, -phyWheelRadius} + posBody,
    posWheelBackLeft + {-phyWheelHalfThickness, phyWheelRadius} + posBody],
   dirBackLeft, posWheelBackLeft + posBody],

  PointSize → Large, Red,
  Point[posWheelFrontRight + posBody], Point[posWheelBackRight + posBody],
  Point[posWheelFrontLeft + posBody], Point[posWheelBackLeft + posBody]

 }
```

```
In[•]:= Clear[comDrawCircles]
comDrawCircles[posBody_, dirBodyAngle_, posCurveCenter_] :=
 {
  (*Circle that passes through the center of the vehicle*)
  colCurve,
  {PointSize → 0.05, Point[posCurveCenter + posBody]},

  bodyRadius = Norm[posCurveCenter];
  wheelFrontRightRadius = Norm[posCurveCenter - posWheelFrontRight];
  wheelBackRightRadius = Norm[posCurveCenter - posWheelBackRight];
  wheelFrontLeftRadius = Norm[posCurveCenter - posWheelFrontLeft];
  wheelBackLeftRadius = Norm[posCurveCenter - posWheelBackLeft];

  Circle[posCurveCenter + posBody, bodyRadius],
  (*front right wheel's path circle*)
  Circle[posCurveCenter + posBody, wheelFrontRightRadius],

  (*back right wheel's path circle*)
  Circle[posCurveCenter + posBody, wheelBackRightRadius],

  (*front left wheel's path circle*)
  Circle[posCurveCenter + posBody, wheelFrontLeftRadius],
  (*back left wheel's path circle*)
  Circle[posCurveCenter + posBody, wheelBackLeftRadius],

  arcLength = 2;
  arrowLength = 0.1;
  Arrowheads[0.02],

  Blue,
  bodyRadius = Norm[posCurveCenter];
  angle = ArcTan @@ (posCurveCenter) ;
  angleEnd = angle + arcLength Sign[z2] / radius;
  direction = -arrowLength Sign[z2] {-Sin[angleEnd], Cos[angleEnd]};
  Circle[posCurveCenter + posBody, bodyRadius, {angle, angleEnd} + Pi],
  arcTip =
   posCurveCenter + bodyRadius {Cos[angleEnd + Pi], Sin[angleEnd + Pi]} + posBody;
  Arrow[{arcTip, arcTip + direction}],

  (*front right wheel direction*)
  Red,
  If[b1 || b2,
   If[b1, reductionFactor = 2000, reductionFactor = 2000], reductionFactor = 1];
```

```
 posFrontRightWheelCenter = (posWheelFrontRight + posBody);
 radius = Norm[posCurveCenter - posWheelFrontRight];
 angle = ArcTan @@ (posCurveCenter - posWheelFrontRight) ;
 angleEnd = angle + arcLength / reductionFactor Sign[z2] / bodyRadius;
 direction = - arrowLength Sign[z2] {- Sin[angleEnd], Cos[angleEnd]};
 Circle[posCurveCenter + posBody, radius, {angle, angleEnd} + Pi],
 arcTip = posCurveCenter + radius {Cos[angleEnd + Pi], Sin[angleEnd + Pi]} + posBody;
 Arrow[{arcTip, arcTip + direction}],

 (*back right wheel direction*)

 posBackRightWheelCenter = (posWheelBackRight + posBody);
 radius = Norm[posCurveCenter - posWheelBackRight];
 angle = ArcTan @@ (posCurveCenter - posWheelBackRight) ;
 angleEnd = angle + arcLength / reductionFactor Sign[z2] / bodyRadius;
 direction = - arrowLength Sign[z2] {- Sin[angleEnd], Cos[angleEnd]};
 Circle[posCurveCenter + posBody, radius, {angle, angleEnd} + Pi],
 arcTip = posCurveCenter + radius {Cos[angleEnd + Pi], Sin[angleEnd + Pi]} + posBody;
 Arrow[{arcTip, arcTip + direction}],

 (*front left wheel direction*)
 posFrontLeftWheelCenter = (posWheelFrontLeft + posBody);
 radius = Norm[posCurveCenter - posWheelFrontLeft];
 angle = ArcTan @@ (posCurveCenter - posWheelFrontLeft) ;
 angleEnd = angle + arcLength / reductionFactor Sign[z2] / bodyRadius;
 direction = - arrowLength Sign[z2] {- Sin[angleEnd], Cos[angleEnd]};
 Circle[posCurveCenter + posBody, radius, {angle, angleEnd} + Pi],
 arcTip = posCurveCenter + radius {Cos[angleEnd + Pi], Sin[angleEnd + Pi]} + posBody;
 Arrow[{arcTip, arcTip + direction}],

 (*back left wheel direction*)
 posBackLeftWheelCenter = (posWheelBackLeft + posBody);
 radius = Norm[posCurveCenter - posWheelBackLeft];
 angle = ArcTan @@ (posCurveCenter - posWheelBackLeft) ;
 angleEnd = angle + arcLength / reductionFactor Sign[z2] / bodyRadius;
 direction = - arrowLength Sign[z2] {- Sin[angleEnd], Cos[angleEnd]};
 Circle[posCurveCenter + posBody, radius, {angle, angleEnd} + Pi],
 arcTip = posCurveCenter + radius {Cos[angleEnd + Pi], Sin[angleEnd + Pi]} + posBody;
 Arrow[{arcTip, arcTip + direction}]
}
```

```
In[•]:= Clear[comWheelLabels]
comWheelLabels[posBody_, dirBodyAngle_] :=
 {
  (* common to all wheels *)
  Arrowheads[0.02],
  (*front right*)
  LightGray,
  Arrow[{1.1 phyWheelFrontRight + posBody, 1.9 phyWheelFrontRight + posBody}],
  (*back right*)
  Arrow[{1.1 phyWheelBackRight + posBody, 1.9 phyWheelBackRight + posBody}],
  (*front left*)
  Arrow[{1.1 phyWheelFrontLeft + posBody, 1.9 phyWheelFrontLeft + posBody}],
  (*back left*)
  Arrow[{1.1 phyWheelBackLeft + posBody, 1.9 phyWheelBackLeft + posBody}],

  Gray,
  Text[Style["Stuff", 18], 2 phyWheelFrontRight + posBody],
  Text[Style["Stuff", 18], 2 phyWheelBackRight + posBody],
  Text[Style["Stuff", 18], 2 phyWheelFrontLeft + posBody],
  Text[Style["Stuff", 18], 2 phyWheelBackLeft + posBody]
 }
```

This will show the calculation for each wheel's speed and angle relative to the main body

```
In[•]:= Clear[comDrawCalculations]
comDrawCalculations[posBody_, dirBodyAngle_, posCurveCenter_] :=
 {


  (* First draw where the direction 0 would be for each wheel *)
  Arrowheads[0.02],
  (* front right wheel *)

  angleFrontRight = (dirFrontRight - rotBody);
  angleFrontRight = Mod[angleFrontRight + Pi, 2 Pi] - Pi;
  angleBackRight = (dirBackRight - rotBody);
  angleBackRight = Mod[angleBackRight + Pi, 2 Pi] - Pi;
  angleFrontLeft = (dirFrontLeft - rotBody);
  angleFrontLeft = Mod[angleFrontLeft + Pi, 2 Pi] - Pi;
  angleBackLeft = (dirBackLeft - rotBody);
  angleBackLeft = Mod[angleBackLeft + Pi, 2 Pi] - Pi;
  speedFrontRight = wheelFrontRightRadius / bodyRadius;
```

```
speedBackRight = wheelBackRightRadius / bodyRadius;
speedFrontLeft = wheelFrontLeftRadius / bodyRadius;
speedBackLeft = wheelBackLeftRadius / bodyRadius;

Gray,
Rotate[Arrow[
  {posWheelFrontRight + posBody, posWheelFrontRight + posBody + {0, arcLength}}],
 rotBody, posWheelFrontRight + posBody],
Rotate[Arrow[{posWheelBackRight + posBody, posWheelBackRight +
    posBody + {0, arcLength}}], rotBody, posWheelBackRight + posBody],
Rotate[Arrow[{posWheelFrontLeft + posBody, posWheelFrontLeft + posBody +
    {0, arcLength}}], rotBody, posWheelFrontLeft + posBody],
Rotate[Arrow[{posWheelBackLeft + posBody, posWheelBackLeft + posBody +
    {0, arcLength}}], rotBody, posWheelBackLeft + posBody],
Pink,
Rotate[Arrow[
  {posWheelFrontRight + posBody, posWheelFrontRight + posBody + {0, arcLength}}],
 rotBody + angleFrontRight, posWheelFrontRight + posBody],
Rotate[Arrow[{posWheelBackRight + posBody,
   posWheelBackRight + posBody + {0, arcLength}}],
 rotBody + angleBackRight, posWheelBackRight + posBody],
Rotate[Arrow[{posWheelFrontLeft + posBody,
   posWheelFrontLeft + posBody + {0, arcLength}}],
 rotBody + angleFrontLeft, posWheelFrontLeft + posBody],
Rotate[Arrow[{posWheelBackLeft + posBody,
   posWheelBackLeft + posBody + {0, arcLength}}],
 rotBody + angleBackLeft, posWheelBackLeft + posBody],

circleRadius = 2;
Rotate[Circle[posWheelFrontRight + posBody, circleRadius,
  {0, angleFrontRight} + Pi / 2], rotBody, posWheelFrontRight + posBody],
Text["Angle = " <> ToString[-angleFrontRight 180 / Pi] <> "°" <>
  "\nSpeed = " <> ToString[wheelFrontRightRadius / bodyRadius],
 RotationTransform[rotBody, posWheelFrontRight + posBody][
  posWheelFrontRight + 3 {1, 0} + posBody]],

Rotate[Circle[posWheelBackRight + posBody, circleRadius,
  {0, angleBackRight} + Pi / 2], rotBody, posWheelBackRight + posBody],
Text["Angle = " <> ToString[-angleBackRight 180 / Pi] <> "°" <>
  "\nSpeed = " <> ToString[wheelBackRightRadius / bodyRadius],
 RotationTransform[rotBody, posWheelBackRight + posBody][
  posWheelBackRight + 3 {1, 0} + posBody]],

Rotate[Circle[posWheelFrontLeft + posBody, circleRadius,
```

```
    {0, angleFrontLeft} + Pi / 2], rotBody, posWheelFrontLeft + posBody],
  Text["Angle = " <> ToString[-angleFrontLeft 180 / Pi] <> "°" <>
    "\nSpeed = " <> ToString[wheelFrontLeftRadius / bodyRadius],
   RotationTransform[rotBody, posWheelFrontLeft + posBody][
    posWheelFrontLeft + 3 {-1, 0} + posBody]],

  Rotate[Circle[posWheelBackLeft + posBody, circleRadius, {0, angleBackLeft} + Pi / 2],
   rotBody, posWheelBackLeft + posBody],
  Text["Angle = " <> ToString[-angleBackLeft 180 / Pi] <> "°" <>
    "\nSpeed = " <> ToString[wheelBackLeftRadius / bodyRadius],
   RotationTransform[rotBody, posWheelBackLeft + posBody][
    posWheelBackLeft + 3 {-1, 0} + posBody]]

}
```

Master 4WD/4WS vehicle control algorithm v4 - Graphics.nb

Initialize the positions and directions. Reset the controller

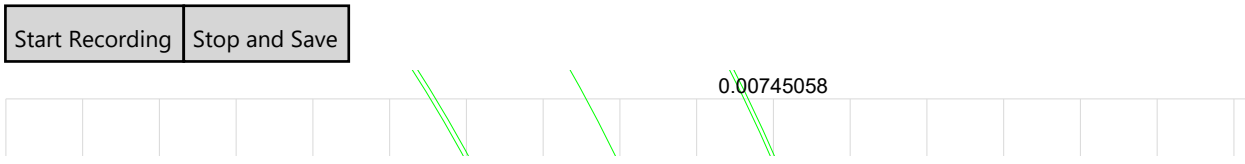
In[41]:=
```
(* In order to avoid dividing by zero,
a zeroRange was defined.  Any number that falls between -
 zeoRange and +zeroRange will be calculated separately.*)
zeroRange = 0.00001;

(*This is to reset the controller.  When this line is run,
the rest of program acts as if the controller was pulled out then plugged back in.*)
{x1, y1, z1, x2, y2, z2} =
  latest = ControllerState["SpaceNavigator", {"X1", "Y1", "Z1", "X2", "Y2", "Z2"}];
(*If z2 is really zero, then the car does not show the arrows because
 the routine assume it is on a curvature of some sort.  So,
create a very slight rotation so that the arrows show up.  Without this line,
there are not arrows until the controller experiences some rotation.*)
latest[[-1]] += zeroRange / 100;

(*create a routine to store last ten positions*)
posBodyHistory = Table[{x1, x2}, 10];
(*default initial direction of movement of the body*)
dirBody = {0, 1.};
(*default last direction of movement of the body.  Last direction
 is important because if the car didn't move between two Dynamic,
then the newly calculated direction would result in no direction.  In that case,
the old direction has to be used to poin the car to the direction it was pointing to.*)
dirBodyLast = {0, 1.};
(*The direction of the body's movement and how the car is rotated are two
 seperate matters in 4WD/4WS. rotBody is where the car's front is pointing to,
not where it is heading to.  Zero is north.*)
rotBody = 0.;
(*The controller's reading is so low that they have to be
 boosted.  Since translation and rotation produces different scale of numbers,
or they are used different on the program, they require different boost
 factors.  Notice that 0.5 was to used to make the rotation not so sensitive.*)
factor = 100000 {2, 2, 2, 1, 1, .5};
(*Where the car was located initially*)
posBody = {x1, y1};
```

This is the main routine that loop through

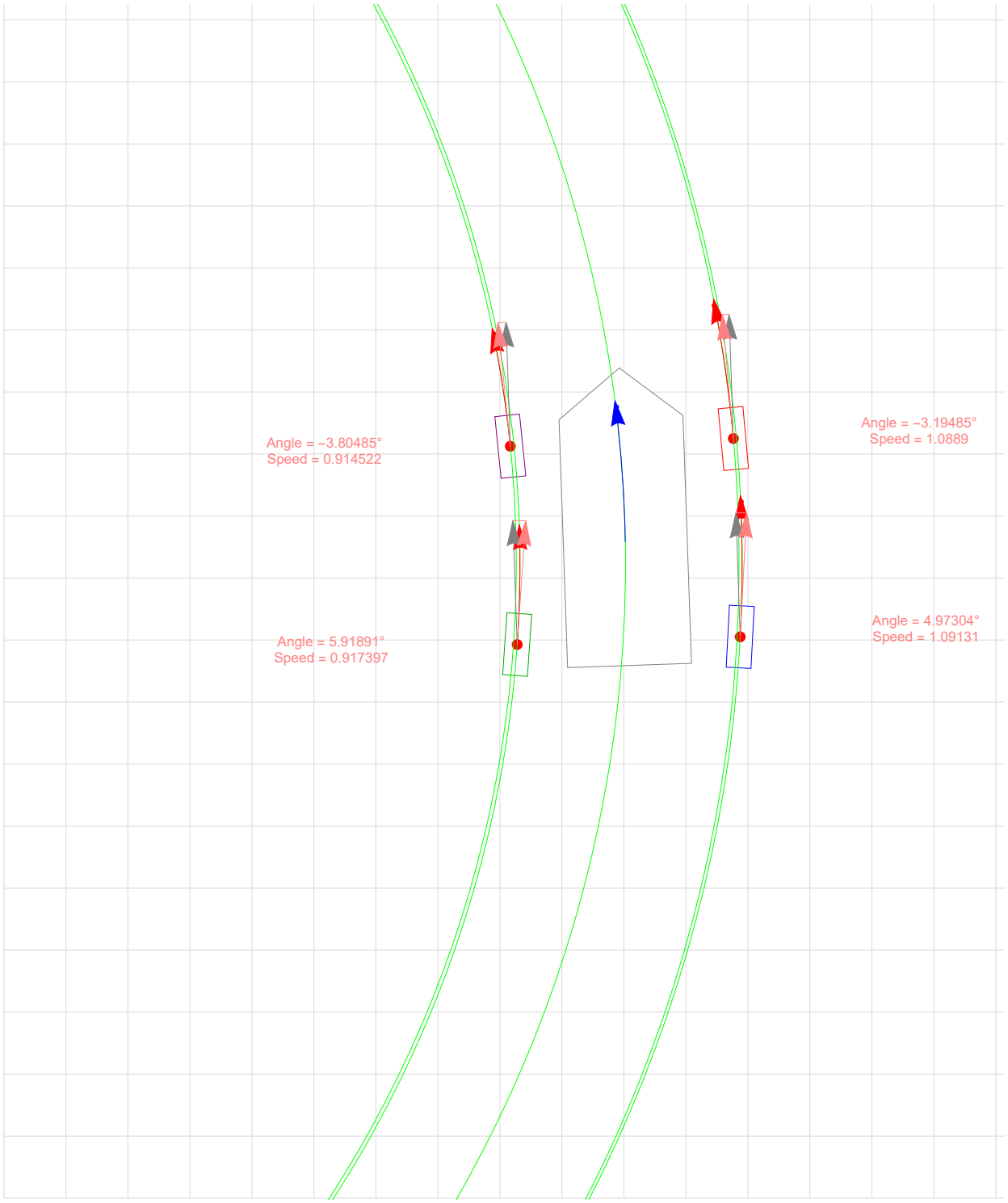
In[50]:=
```
Dynamic[
 (* read in the fresh input from the space mouse,
```

```
subtract the "latest" to get only the values that changed since the "latest".  Then
 multiply by the factor to adjust the sensitivities of each degree of freedom.*)
{x1, y1, z1, x2, y2, z2} = factor
   (ControllerState["SpaceNavigator", {"X1", "Y1", "Z1", "X2", "Y2", "Z2"}] - latest);
(* read the buttons to be used in case there is a rotation*)
{b1, b2} = ControllerState["SpaceNavigator", {"B1", "B2"}];

(* This is to collect last ten inputs.  posBodyHistory stores
 the lat readings.  RotateRight moves all the numbers to the right,
shifting them to the past by one step.*)
posBodyHistory = RotateRight[posBodyHistory];
(* then it fills the most recent spot with the latest data*)
posBodyHistory[[1]] = {x1, y1};

(*Now the vehicle has to travel along the circle.*)
(* Check how long it moved in x and y direction.  The x1 and
 y1 is the reading of the spacemouse.  The mouse's direction agrees
 with the vehicle only when the car is facing north.  Other times,
the intended direction such a forward, given by the driver has to be
 translated as the direction on the simulation by rotating it.  This step
 is necessary only in the screen.  The actual vehicle should not do this.*)
{dx, dy} = RotationTransform[rotBody][posBodyHistory[[1]] - posBodyHistory[[2]]];
(* Find out how far it traveled by finding the hypothenuse *)
distance = Norm[{dx, dy}];
(* Find out the radius of curvature of the motion so
 that the car's trajectory along that circle can be calculated. *)
bodyRadius = Norm[posCurveCenter];
(* now that we know how much it moved since last time "distance" and the radius of
 the circle "bodyRadius" we can calculate how many radians that motion is on the
 circle.  This is necessary because we cannot let the vehicle move straight.  We need
 to make it follow the circle because it has to follow the radius of curvature *)
dTheta = Sign[z2] distance / bodyRadius;
(* find out at what angle the center of the vehicle is
 relative to the center of curvature.  Or find out at what angle
 the center of curvature is relative to the vehicle's center. *)
theta = ArcTan @@ posCurveCenter;
(* now the car moves along the circle by moving tangential to the
 circle of curvature.  {-Sin[theta],Cos[theta]} is the perpendicular
 direction to a circle that is drawn with {Cos[theta],Sin[theta]} *)
posBody += -Sign[z2] distance {-Sin[theta], Cos[theta]};
(* Since the vehicle is going around a circular path,
the vehicle would appear to be spinning when seen from the top *)
rotBody += dTheta;
(* if b1 is pressed, then spin left. *)
If[b1, rotBody += 0.05;];
(* if b1 is pressed, then spion right *)
If[b2, rotBody -= 0.05;];

(* if the spacemouse was not touched,
which is detected by the size of the disatnce, then keep the last position *)
If[distance < 0.01, dirBody = dirBodyLast,
```

```
  (* but if it was touched then the direction becomes
   {dx, dy} and the last direction is also stored as {dx, dy}*)
  dirBody = dirBodyLast = {dx, dy}];
 (* adjust which direction  *)
 rotBodyDirectionAngle = ArcTan @@ dirBody + Pi / 2 + Pi;
 (* this is where the center of circular path
  is.  This is computed only when there is any z2 rotation. *)
 posCurveCenter = If[-zeroRange < z2 < zeroRange,
    (* if z2 was too small, then just assume that
     the center of the circular path is 10000 units away to the right,
    the rotate it for the case when it is not directly on the side of the vehicle. *)
    RotationTransform[rotBodyDirectionAngle, {0, 0}][{10000, 0}],
    (* if z2 value was not near zero, then calculate the center of circular path *)
    RotationTransform[rotBodyDirectionAngle, {0, 0}][{-1 / z2, 0}]];

 (*now comes actual painting of the simulation*)
 Column[{Row[{Button["Start Recording", startTime = AbsoluteTime[];
        recordMoves = True;
        fname = FileNameJoin[{NotebookDirectory[], "testfile.csv"}];
        stream = OpenWrite[fname];
        WriteString[stream, "Time, Center Speed, Front Right Speed, Back Right
           Speed, Front Left Speed, Back Left Speed, Front Right Angle,
    Back Right Angle, Front Left Angle, Back Left Angle\n"];],
       Button["Stop and Save", recordMoves = False;
        Close[stream]]}], Graphics[{(*Draw the body.comCar needs to know where
        the center of the body is,and what the rotation is*)comCar[posBody, rotBody],
       (*Draw the wheels.It needs to know where the body is,how rotated the body is,
       and where the center of circular path is because the wheels will align in the
        tanglential direction to that circular path*)comWheels[posBody, rotBody,
        If[b1, -0.001 posCurveCenter, If[b2, 0.001 posCurveCenter, posCurveCenter]]],
       (*The command below is to draw the lables for each wheel.*)(*comWheelLabels[
        {x1,y1},rotBody],*)(*Draw the trajectory which are all circles.Even straight
        lines are circles with very large radii.*)comDrawCircles[posBody, rotBody,
        (*if button 1 is pressed,then assume that the center of the circle is almost at
         the center of the body of the vehicle but not quite.It is at 0.001 radius from
         the center to preserve the direction of rotation*)If[b1, -0.001 posCurveCenter,
          (*On the other hand,if button 2 is pressed then it should spin right*)If[b2,
           0.001 posCurveCenter, posCurveCenter]]], (*Finally compute all 8 parameters
        (angles and speed for each wheel) and then draw the values on the graphcis.*)
       comDrawCalculations[posBody, rotBody, If[b1, -0.001 posCurveCenter, If[b2,
          0.001 posCurveCenter, posCurveCenter]]], comBackground, PlotLabel → distance]}]
If[distance > 0.001 && recordMoves, WriteString[stream,
    StringTake[ToString[{AbsoluteTime[] - startTime, distance, speedFrontRight,
        speedBackRight, speedFrontLeft, speedBackLeft, angleFrontRight,
        angleBackRight, angleFrontLeft, angleBackLeft}], {2, -2}] <> "\n"]]
]
```



Out[50]=




```
  WriteString[stream, StringTake[ToString[{AbsoluteTime[] - startTime, distance,
        speedFrontRight, speedBackRight, speedFrontLeft, speedBackLeft,
        angleFrontRight, angleBackRight, angleFrontLeft, angleBackLeft}], {2, -2}] <>
]]
```

In[51]:=
```
Dynamic[{z2, posCurveCenter}]
```

Out[51]= {0.0485278, {−20.6036, −0.363166}}

In[52]:= **Dynamic@distance**

Out[52]= 0.00745058