

# Simulation of Variable Turning Speed in Robotics using USARSim

**Brandon Burger**  
Dept. of Computer  
Science  
Hood College  
Frederick, MD  
[brb3@hood.edu](mailto:brb3@hood.edu)

**Kristopher Reese**  
Dept. of Computer  
Science  
Hood College  
Frederick, MD  
[kwr2@hood.edu](mailto:kwr2@hood.edu)

**Xinlian Liu**  
Dept. of Computer  
Science  
Hood College  
Frederick, MD  
[xliu@pluto.hood.edu](mailto:xliu@pluto.hood.edu)

## ABSTRACT

*Unified System for Automation and Robotics Simulator (USARSim) and Mobility Open Architecture Simulation and Tools (MOAST) offer a framework that is designed to provide an accurate simulation environment for multiple forms of Robots. While running a robot using MOAST, a developer can see that turning may cause many issues while running at high speeds. This paper attempts to alleviate issues that are associated with running a robot at high speeds by offering an algorithm that can be used to allow the robot to run variable speeds while turning by slowing down and speeding up when entering and exiting a turn respectively.*

**Keywords:** Robotics, USARSim, MOAST, Robotic Movement, Robotic Simulation, Variable Turning Speed

## I. Introduction

Mobility Open Architecture Simulation and Tools (MOAST) is a framework that has been designed for multiple development environments, as well as on real robotics systems. Since its creation, the National Institute of Standards and Technology (NIST) have taken considerable care to assure that the MOAST framework offers its developers an accurate simulation environment within the Unified System for Automation and Robot Simulation (USARSim) environment. This joint framework combines full 3 dimensional environments with physics-based interaction between objects using the Karma physics engine. [1]

MOAST is the primary tool for development of simulations of robots, and controls all of the primary systems on the robot, including movement, sensory data, and world modeling. MOAST can be broken down into 4 echelons, each controlling a specific portion of the robot. The bottom echelon of MOAST is the Primitive Echelon, which controls the majority of determining the velocity of the robot. Sensor Processing and World Modeling are very limited in this echelon.

The second layer of MOAST is the Autonomous Mobility Echelon. This echelon extends the Primitive echelon by controlling the movement of the robot through the world. It is in this echelon that the robot will handle turning. World Modeling and

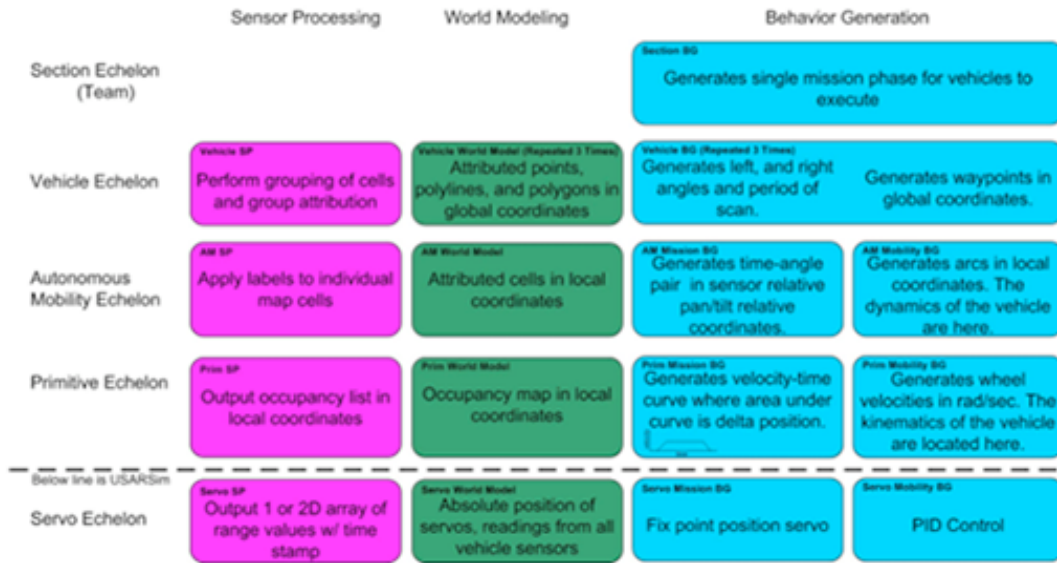


Figure 1: MOAST echelons

Sensor Processing are again very limited. However in the third echelon, the Vehicle Echelon, we see less emphasis on movement and more on sensor processing and world modeling. The fourth echelon allows a developer to create missions for the robot using the other three echelons described here.

We found limitations in MOAST that created many issues while attempting to run multiple robots in the world. The primary limitation was the lack of support for communication between the robots. This limited the running of the robots to receiving and processing its own sensory data to determine the location of other robots. This method creates many more problems in the system by not being able to prioritize or consistently know the locations of other robots. This could cause robots to become trapped in areas where it will not be able to reverse (due to another robot following the first into the area.) The best solution to these issues is to create some form of communication system between robots.

Another issue arose during simulations of turning in MOAST and USARSim. It was determined that this problem must first be resolved before multiple robots could run in the same world. The Primitive echelon of MOAST allows its developers control over the robots speed. We found that during a sharp turn, the robot may have to constantly start, stop, and reverse in order to make it around a sharp turn while running at high speeds. When a speed is set, it will not dynamically change as one might expect while taking a sharp turn or moving in an arc.

This paper attempts to solve the variable turning speed issue. Section 2 describes the problem in greater detail. Section 3 offers an explanation of our solution, as well as pseudo-code for the algorithm that we have used for variable turning speed. Section 4 concludes the paper and discusses possible future developments that could make our algorithm perform much more efficiently.

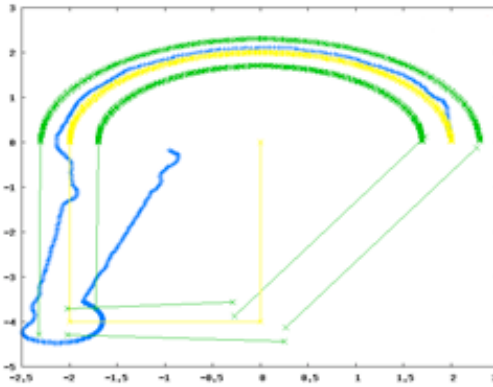


Figure 2: Turning at Speed 2

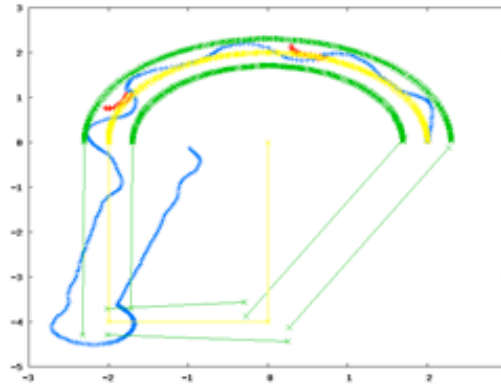


Figure 3: Turning at Speed 3

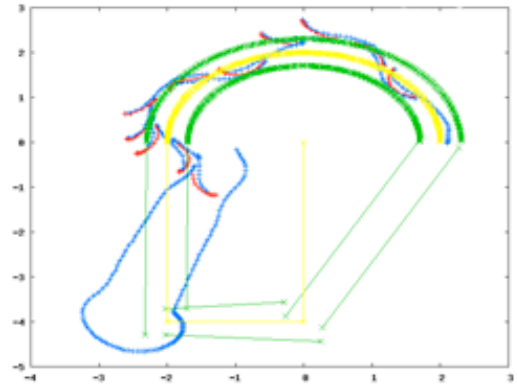


Figure 4: Turning at Speed 5

## II. The Problem

Generally, vehicles take turns by decelerating into and accelerating out of the curve. This is to account for centripetal force of the vehicle. By decreasing the velocity of the vehicle, this allows for greater use of the Instantaneous Center of Rotation (ICR). With the currently used robot (Unit Loader) the ICR is fixed in its arc rotation, it is only given one degree of freedom. Thus, the manipulation of speed on any given and throughout a turn allows the Unit Loader to be able to take any curve at the best-suited speed.

A general issue with manipulation of speed on any given turn is the timing of deceleration and acceleration of the vehicle. If one were to decelerate too soon into a turn, the vehicle would be given too much ICR control and then the problem becomes an issue of performance. Decelerating too late would cause the vehicle to lose control of its ICR and ultimately lose stability. In this case, the Unit Loader would have to initiate reverse movements because the robot cannot make the next waypoint. Thus, the major problem that one must deal with is angular velocity.

Figure 1-3 represent Unit Loader movement in a constant curvature arc starting from a fixed position. The plot (comprised of blue, red, yellow, and green) is broken down into different movements and thresholds. Blue corresponds with forward movement, whereas, red corresponds to reverse movement. Green constitutes the outer bounds of the path that the Unit Loader follows, where yellow represents the goal line that the robot tries to stay as close to as possible. As shown, the slower the speed of the robot yields greater precision of path management. When the speed increases, the robot has difficulty making it to the next waypoint.

### III. The Solution

The solution that is used for this problem is broken down into two major characteristics. One must answer: “What is considered a sharp enough turn and secondly, when should the robot slow down and speed back up?” The argument over how much the robot should decelerate is not necessary, as this current system does not focus on speed and performance. Rather, the USARSim/MOAST system focuses on reliable transportation and sensory processing. To best explain the solution to variable velocity in turns is the presented pseudo code (more simplified code for easier clarity). It encompasses the entire problem in a rather simple format.

```

list(logx, logy) = getLastPosition();
list(x, y) = getCurrentPosition();

if((x - logx) == 0) {
    slope = (y - logy)/(x - logx);
}
else {
    slope = 0;
}

if(!isset(logSlope) || ((logSlope - slope) == 0)) {
    logSlope = slope;
}
else {
    lx = start of turn (x)
    ly = start of turn (y)

    if (|x| >= |lx|+t) || (|y| >= |ly|+t) { //t = threshold constant
        slowdown();
        while((|x| >=|lx|+t) || (|y| >=|ly|+t)) {
            if((|x| <= |lx|+t) || (|y| <= |ly|+t)) {
                speedUp();
                break;
            }
        }
    }
}
}

```

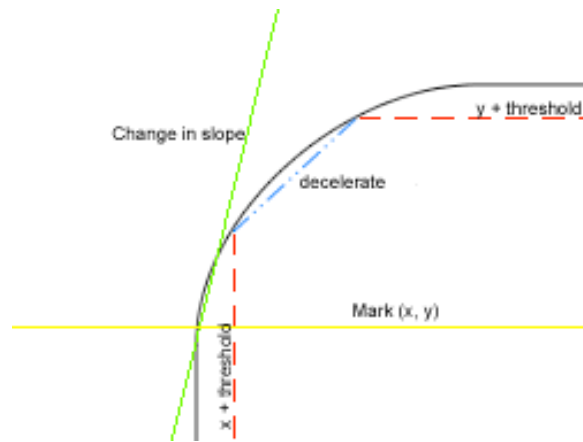
To simplify the solution down further, one can look at the pseudo code in sections.

```

list(logx, logy) = getLastPosition();
list(x, y) = getCurrentPosition();

if((x - logx) == 0) {
    slope = (y - logy)/(x - logx);
}
else {
    slope = 0;
}

```



**Figure 5: When a change in slope that surpasses the allowable slope (green) we mark the x & y coordinates. When the robot hits the x + threshold (red-dotted) the robot begins to decelerate through the curve (blue-dotted). After the Robot hits the y + threshold (red-dotted) the robot begins to accelerate to full speed again.**

This section provides slope coordinates from the current position and the previous position and calculates the current slope of the Unit Loader's trajectory. This is pivotal later on as it determines when the system should check on the sharpness of the turn. If there is no slope or the Unit Loader is either just starting or finishing its transportation, the slope is set to zero to become more lightweight. It would be much more computationally expensive to consistently check for turns with just a set threshold, which is why no movement should have slope set to zero, as it would never check its threshold while not moving.

Secondly:

```

if(!isset(logSlope) || ((logSlope - slope) == 0)) {
    logSlope = slope;
}
else {
    lx = start of turn (x)
    ly = start of turn (y)
}

```

This section of the pseudo code is important for the fact that it checks on any given changes in slope, if there are no changes, the previous slope used as a reference for the next slope check. If the slope has changed, we take the coordinates of both the X and Y (in reference to the map that Unit Loader is presently on). These coordinates are crucial for setting up a threshold that will be explained later.

Finally:

```

if (|x| >= |x|+t) || (|y| >= |y|+t) {
    slowdown();
    while((|x| >= |x|+t) || (|y| >= |y|+t)) {
        if((|x| <= |x|+t) || (|y| <= |y|+t)) {
            speedUp();
            break;
        }
    }
}
}

```

The most complex of all the sections, this explains the coding of the threshold barrier that is used to essentially “tell” the Unit Loader to slow down. By using the absolute value of both current X and Y versus the “start of turn (x)” and “start of turn (y)” plus an absolute value t, “tells” the robot when the speed should decrease, as seen in Figure 4. The Unit Loader should slow down at the pinnacle of the turn (where most of the angular velocity is present).

The Unit Loader should also accelerate to its previous speed. The only issue is the Unit Loader must know when to accelerate. Typically, when the slope normalizes on the opposite coordinate line (X versus Y) is when the Unit Loader can resume its previous speed. To account for both X and Y and to reduce the calling of the specific coordinate, the code accounts for both X and Y at the same time. So at any turn, the Unit Loader checks both X and Y on both ends of the threshold (both deceleration and acceleration).

Slope describes the gradient of a line on an X and Y-axes and is found via the equation  $m = \Delta y / \Delta x$  (or the change of the y coordinate over the change in the x coordinate). Slope was used for this solution as it presents a nice way to clearly see a change in direction. Any turn can be seen as a change in slope (whether in a sharp or soft manner). So at the onset the Unit Loader “sees” this change in slope, it can begin its threshold check. Slope checking provides clarification as to when a turn actually takes place.

The more complex feature of this solution is the threshold barrier. After the Unit Loader has distinguished a turn, an imaginary threshold is set. The main functioning part that sets this up in code is if “(|x| >= |x|+t) || (|y| >= |y|+t) {“, the variable “t” is the offset used to distinguish the threshold at which the Unit Loader should slow down and speed back up. The threshold can be best pictured when the Unit Loader passes it, it slows down or speeds up depending on its status in the turn. Thus the threshold provides to the answer to “when to slow down and speed back up” and the sharpness of the turn.

We can see this solution work when placed under tests. As the speed decreases on a turn, the amount that a Unit Loader uses reverse movement to place it at a coordinate that allows it to make the next planned waypoint is less. Thus, the Unit Loader can move at a speed of 5 on straight turns and use a given speed that is comfortable and prevent reverse

movement that not only causes performance issues but also could prevent further Unit Loader communication issues.

#### **IV. Conclusion and Future Work**

It can be seen that our solution offers a viable option to allow robots to run at full speed on straight-aways and allow the robot to run at much slower speeds on hard turns that would cause issues if running at high speeds. From a performance perspective, the Unit Loader will be able to handle a factory environment more efficiently. Due to time constraints, a communications system was not completed in time. This is the next step to allowing multiple robots to work in the same environment and is something that must be worked on in the future. This algorithm requires a lot of storage of previous locations on a map and could become much more memory efficient if the derivative of the location was taken instead of the slope. We could also see that the derivative has the potential to be much more accurate than slope would be. However, determining an equation to use for the derivative may prove very challenging, and therefore is something to look at for the future. Another future development for the algorithm would be variable thresholds dependent on speed. This would alleviate any issues of slowing down while the robot could potentially make the turn without deceleration. This future work would enhance performance under simulations both virtual and physical.

#### **V. References**

[1] Balakirsky, Stephen “Mobility Open Architecture Simulation and Tools Reference Manual.” *National Institute of Standards and Technology*. Rev. 3.3: March 11, 2008. < [http://sourceforge.net/project/showfiles.php?group\\_id=148555&package\\_id=215082](http://sourceforge.net/project/showfiles.php?group_id=148555&package_id=215082)>