Marshall Ruse

1006 6247

CISC 499

Project: Automatic Rover Control in a Leader-Follower Simulation

Supervisor: Professor Juergen Dingel

Friday, April 5, 2019

Queen's University,

Kingston, Ontario, Canada

Introduction

Mobile autonomous robots will be increasingly present in our lives. They will take many different forms and serve numerous varied purposes in all sectors of society, whether as self-driving cars and trucks, self-guided warehouse robots, or autonomous rovers searching other planets on remote scientific expeditions, to name a few high-profile examples. The ability to rapidly prototype and simulate their control and resulting behaviours within a realistic environment subject to realistic physics will be an increasingly relevant need as more sectors of society adopt these robots.

Project Goal

The goal of the project was to design an automated control program to guide a Following rover in a Leader-Follower scenario. This scenario takes place in a simulated flat environment in which two rovers (with appearances based on NASA's Curiosity Mars rover) are able to move around freely. The simulation is built with Unity game engine, with controlling scripts for the assets of the simulation and the TCP port communication channels generated by SimGen, a rapid simulation prototyping tool. The goal of the scenario is to have the Follower stay within a precise distance of the Leader rover, not being allowed to stray too close nor too far. The rovers may only be communicated with to receive updates and issue commands through TCP communication ports with predefined commands. The challenge is complicated by the fact that the Leader's movements are random - it accelerates, brakes, and turns in random directions, all on random time intervals . The parameters for this random movement, as well as the minimal and maximal speed with which the Leader travels, can be set in a configuration file.

An extension of this project goal was to generate multiple rovers via SimGen, to build them via Unity into the simulation, and to control them all as an extension to the base Leader-Follower scenario. This extension results in a convoy-like behaviour as the rovers all follow each other.

Class Structure / Program Architecture

The external controlling program consists of two primary classes, the *Rover* and *Supervisor* classes, as well as a couple recording and analysis classes that are passed into the Supervisor classes as constructor parameters. These are the primary classes responsible for the behaviour of the Follower in the simulation, which is communicated with through predefined messages via TCP port. A diagrammatic

representation of the classes relationships to each other and to the simulated environment can be seen in **Figure 1.**

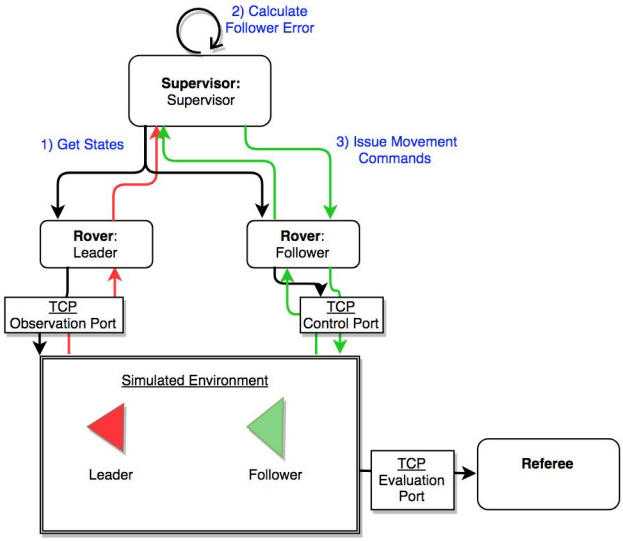

Figure 1: A diagram of the controller-program's software architecture. A *Supervisor* class monitors a pair of *Rover* classes, which communicate to the Simulation through TCP ports. A Referee program monitors the simulation externally.

Rover Class

Each rover in the simulation environment is assigned its own instance of the *Rover* class. The *Rover* class is responsible for:

- Storing an instance of the SimulationSocketInterface, a persistent socket connection to the simulator for that Rover's designated communication port.
- Storing constant values for the rover, such as (the somewhat arbitrarily assigned) wheel radius and axle-length, used in calculations for the differential drive commands, as well as the rover's name, used as a prefix of the commands sent to the simulation.
- Storing the state values of their assigned simulated rovers, such as their position (current and past), heading with respect to the simulation environment's compass, relative angular error to that rover's particular leader (if one exists), and their velocity.
- Issuing formatted commands to the simulation through their socket connection instance.

The *Rover* class is further divided into subclasses of *Leader* and *Follower*, the former of which only differs in the format of the specific commands that it sends to the simulation , and in some of the math used to derive its state values (the Leader does not have a built-in command to obtain its compass heading, for instance, which must be derived from its direction of movement).

Supervisor class

Each pair of *Rover* class instances is given an instance of a *Supervisor* class to which it reports. The *Supervisor* class is responsible for:

- Issuing commands to the Rover class instances to update their states based on values reported directly by the simulation, or from calculations based on such values. For both the Leader and the Follower, these include:
    - position - the simulation world coordinates position of the rovers
    - heading vector - direction of movement in world coordinates,
    - compass angle - the angle relative to the world coordinates' positive z-axis,
    - the rovers' velocities - ie. speed and direction
    - the rovers' speeds.
- Receiving updates about the states from the Rover class instances which are assigned to it.
- Calculating the error between the desired position of the Follower rover relative to the Leader. These errors include:
    - the angular error - the angular deviation of the Leader from the Follower's own coordinate system's positive z-axis,
    - the linear error - the linear deviation from the optimal Euclidean distance from the Leader (optimal simply being half the distance between the minimal and maximal distance).
- Issuing commands to the Follower to correct for the error calculated.

The *Supervisor* class contains a function, *execute()*, that coordinates these responsibilities in the order listed above. This function is called on every iteration of a while-loop in the controller's *main* function (after a specified time interval, set as a global variable, has elapsed) for the duration of the simulation (as specified in the *config* text file). This function calls all others in the *Supervisor* class, which each handle one of the responsibilities above.

Analysis Classes

There are a couple of classes, *SimulationRecorder* and *PostRunAnalysis,* which are instantiated and passed into the *Supervisor* classes, and which record various state, error, and controller gain values for the rovers once per time interval. Such recorded values could be used in analysis of the performance of the Follower for a given set of control parameters, and are also used for the replay animation at the end of each run that shows the rovers' paths through the environment, as well as the Follower's deviance over time.

Controlling the Follower

Control of the Follower is implemented through the use of a Proportional-Integral-Derivative (PID) controller, by far the most used control algorithm for practical feedback loops[1]. The PID controller is given the inputs of the angular and linear error from the Follower's goal destination for the given iteration, and returns the appropriate values for the Follower's angular and linear velocity. It should be noted that, since operations are performed in discrete time intervals, that it is not *true* integrals and derivatives used, but simply discrete approximations. The derivative is simply calculated as the difference between the error value at the current time point, and that immediately prior. The integral is approximated with a Riemann Sum of all of the previous errors until that point.

On each iteration of the *Supervisor.execute()* function, messages are sent to the simulation to retrieve the coordinates and orientation of the Leader-Follower pair. Based on the Follower's coordinates and compass orientation in the simulation's frame of reference, a frame transformation is calculated to produce a frame of reference centered-on and oriented-to the Follower's location and direction of movement. The Leader's position is then translated from the simulation's reference frame to the Follower's. The angular error of the Follower, $\boldsymbol{\theta}$, is calculated as the positive (clockwise) or negative (counter-clockwise) angle that the Leader's position makes with the Follower's origin and positive z-axis. The linear error of the Follower, **d**, is calculated as the Euclidean distance of the Follower from a point along a straight line connecting the Follower and Leader that is half the distance between the minimal and maximal allowed distances from the Leader. This is illustrated in **Figure 2** below:

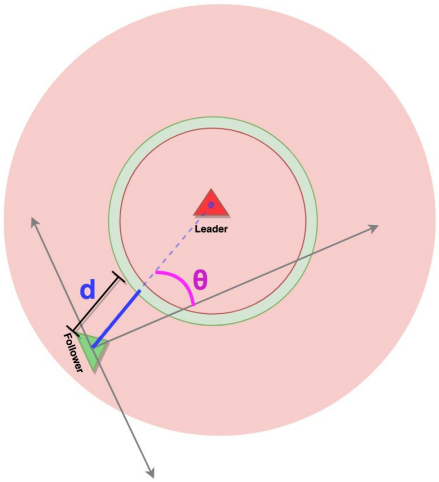

Figure 2: The angular and linear error of the Follower relative to the Leader. The red shaded areas indicate distances too-close or too-far from the Leader, the green indicates the optimal distance.

These values are used to modulate the linear (**v**) and angular (***ω***) velocity of the Follower, in a simplification known as a *unicycle model* [2]. The unicycle model relates these values by the system of equations:

$$\dot{z} = v * cos(\theta)$$
$$\dot{x} = v * sin(\theta)$$
$$\dot{\theta} = \omega$$

The value of the linear velocity is first considered in isolation, as if the Follower were already facing the Leader (***θ*** = 0). The implementation of this in Supervisor is in fact only a proportional-derivative controller, as the integral of the accumulated error does not add any desired behaviour.

The only command sent to the Follower to control its velocity at any given time is *setLRPower(left, right)*, which sets the rover's velocity directly as a function of power applied to the left and right wheels, values that could be designated *pL* and *pR*. To move in a straight line, both values are equivalent, which will simply be denoted *pW*. If *pW* is set to the maximum value of 100, the rover accelerates maximally, and with *pW* set to 0, the rover quickly comes to a stop (the simulation has very little momentum for the rovers, cutting power is effectively the same as braking). Thus, at any given time

point, the rover can be made to stop or accelerate to maximum velocity by setting *pW* somewhere in the range of [0, 100].

The proportional part of the controller is implemented by setting these values in relation to a function of the linear error of the Follower, **d**, which enables the appropriate stopping or acceleration of the Follower in response to the magnitude of **d**. A function such as the logistic function, $f(d) = \frac{1}{1 + e^{-k(d-1)}}$, is suitable to return a value in the range of [0,1] based on the input, **d**. Highly negative (ie. too-close) or highly positive (ie. too-far) linear distance errors result in the outer values of 0 or 1, respectively, which, if multiplied by the maximal power value for the wheels, 100, result in a *pW* that results in stopping or maximal acceleration. Error values nearer to 0 result in less-extreme values of *pW*, meaning that smooth acceleration and braking near the optimal distance can be achieved.

The derivative part of the controller is implemented in order to modulate the braking and acceleration based on the rate of change of the error. If the Leader begins to accelerate rapidly away, or stops rather suddenly, *pW* can be further modified to ensure a similar response in the Follower. Using the absolute value of the derivative of the linear error, d**d** / dt, as an input to a hyperbolic tangent function has a similar effect as the logistic function, but producing an output in the range of [-1,1], which, when scaled by a coefficient, **c**, and added to *pW* produces a more appropriate acceleration response.

The final result is the equation,

$$v = \left(\frac{1}{1 + e^{-k(d-1)}}\right) * v_{max} + c\left(\frac{2}{1 + e^{-2\left(\frac{dd}{dt}\right)}} - 1\right)$$

The calculation of angular velocity, $\boldsymbol{\omega}$, is also the result of a PID-controller, applied separately based on the angular error, $\boldsymbol{\theta}$. The implementation is more straightforward, in that each component of the PID-controller is given a separate coefficient (or *gain*), and these are all summed together.

$$\omega = k_P * \theta + k_I * \sum_{i=0}^{t} \theta_i \Delta t + k_D * \frac{d\theta}{\Delta t}$$

Of course, it is known even intuitively that to make safe, measured turns, linear velocity must be proportionally reduced as angular velocity increases. Thus the two values are related by:

$$v = \frac{v}{\sqrt{|\omega| + 1}}$$

Finally, these two values, **v** and ***ω***, must be converted to values to be sent to the left and right wheels. Control of a mobile robot through separate values assigned to the left and right side is known as a *differential drive model* of control. The differential model uses the following set of equations[3]:

$$\dot{z} = \frac{R}{2}(v_r + v_l)cos(\theta)$$
$$\dot{x} = \frac{R}{2}(v_r + v_l)sin(\theta)$$
$$\dot{\theta} = \frac{R}{L}(v_r - v_l)$$

Which can be combined with the equations from the unicycle model as:

$$v * cos(\theta) = \frac{R}{2}(v_r + v_l)cos(\theta)$$
$$v * sin(\theta) = \frac{R}{2}(v_r + v_l)sin(\theta)$$
$$\omega = \frac{R}{L}(v_r - v_l)$$

Cancelling out the *cos(**θ**)* and *sin(**θ**)* on both sides of the equation yields:

$$v = \frac{R}{2}(v_r + v_l) \Rightarrow \frac{2v}{R} = v_r + v_l$$
$$\omega = \frac{R}{L}(v_r - v_l) \Rightarrow \frac{\omega L}{R} = v_r - v_l$$

At last, the values for the left and right wheel can be found via:

$$v_r = \frac{2v + \omega L}{2R}$$
$$v_l = \frac{2v - \omega L}{2R}$$

These values are the arguments given to *setLRPower()* at the end of each iteration of *Supervisor.execute()*.

Results

Towards the end of development of the controller, once the Follower had achieved consistent tracking of the Leader, two analysis classes, *SimulationRecorder* and *PostRunAnalysis,* were created to record the results of various runs of the simulation. The results of all of the adjustable parameters, such as the angular and linear error gains, the positions of the rovers, the time interval on which Supervisor ran, etc., were recorded, as were the overall performance metrics (percentage of time too far, too close, and at optimal distance) for that run. These values were recorded with the purpose of potentially tuning the controller for optimal gain values that would enable the highest percentage score for time within optimal distance. Unfortunately, there were a few factors that impinged on this ideal data collection:

1. The length of the simulation - a length of one minute for the simulation makes iterations of testing time-costly (especially since multiple runs at each set of values are necessary to get an average performance for that set of parameter values). The time duration of the simulation could have been shortened, but one minute is already a relatively short window for gathering data, and could potentially bias towards higher scores, as the rovers start off at optimal distance from each other, and take a relatively significant time to accelerate.
2. The combinatorial explosion of the parameter values. Assuming even conservative amounts of testing, with only 5-10 values for each of the 6 parameters (3 angular gains, logistic function growth rate, logistic function bias value, linear derivative coefficient) to be set, that is 15 625 combinations on the low-end, 1 000 000 on the high-end. This may be feasible if the result of the combinations could be calculated in some sped-up, GUI-less simulation in the background, but in conjunction with the first factor, this testing was totally non-feasible.
3. The simulation itself is intensive on hardware. Running the simulation for even one iteration tends to cause the laptop it is ran on to overheat and highly engage the cooling-fan. Even if these simulation runs were to be scripted to try just a fraction of the combinations possible, it may be detrimental to the hardware used to run it.
4. Due to the random nature of the Leader's movements, there was no consistency in testing environment in any case. As will be seen in figures to come, the results seem much less to depend on the values of parameters chosen, and more on the particular movements the

Leader made during that run, such as whether it moved in a predominantly straight direction or was continuously turning, or whether it braked and accelerated or maintained a steady pace. Obviously a run with more turns and a poor result could be evidence that an adjustment in one of the angular control gains is necessary, for instance, but there is no guarantee that this situation would ever be repeated, and therefore provide evidence that the adjustment was the correct one.

These points are only mentioned to justify the imprecise, non-systematic, manner in which parameters were tuned, which was primarily trial-and-error, and only in combinations which seemed at the time to be reaping the most gain in optimal distance percentage. Further, recordings were made as all of these parameters were being tuned, so the target outcome, percentage of time at optimal distance, is itself not truly representative of best performances.

With all of these caveats out of the way, a summary result of the runs-to-completion is recorded is shown in **Figure 3**:

% Optimal Distance

```
count    138.000000
mean      52.028154
std       19.970724
min       12.063912
25%       35.538713
50%       53.091831
75%       69.691578
max       87.834916
Name: corrected_goal_dist_percent
```

% Too Far

```
count    138.000000
mean      33.145981
std       17.452443
min        0.921659
25%       18.792902
50%       32.984665
75%       46.551724
max       74.358974
Name: follower_too_far_percent
```

% Too Close

```
count    138.000000
mean      33.145981
std       17.452443
min        0.921659
25%       18.792902
50%       32.984665
75%       46.551724
max       74.358974
Name: follower_too_close_percent
```

Figure 3: The descriptive statistics for all runs of the controller program, with varying Parameter values.

As stated, these figures factor in all runs made at varying values for the parameters. A much more select subset, at the values judged (imprecisely, through trial and error) to be most favourable for achieving the highest optimal distance percentage, are shown in **Figure 4**:

% Optimal Distance *% Too Far* *% Too Close*

```
count    12.000000
mean     69.958709
std      12.440038
min      48.271106
25%      60.637332
50%      75.043853
75%      78.522159
max      82.982370
Name: corrected_goal_dist_percent,
```

```
count    12.000000
mean     17.832995
std      10.906698
min       5.773672
25%      10.218449
50%      13.714565
75%      27.851969
max      34.675615
Name: follower_too_far_percent,
```

```
count    12.000000
mean      1.195285
std       1.990942
min       0.228311
25%       0.232970
50%       0.454567
75%       0.476064
max       6.040268
Name: follower_too_close_percent
```

Figure 4: The descriptive statistics for the subset of runs at the observationally-best parameter values.

These runs are the result of the following parameters:

- Time Interval: 0.1 seconds
- Angular Proportional Gain: 30
- Angular Integral Gain: 0.5
- Angular Derivative Gain: 0.1
- Logistic Function Growth Rate: 10
- Logistic Function Bias Value: 1
- Linear Error Derivative Coefficient: 3

An example depiction of such a run is demonstrated in **Figure 5**. While this run is admittedly cherry-picked, it is an example of what could be achieved with the right parameter settings, and demonstrates that true optimization is theoretically possible.

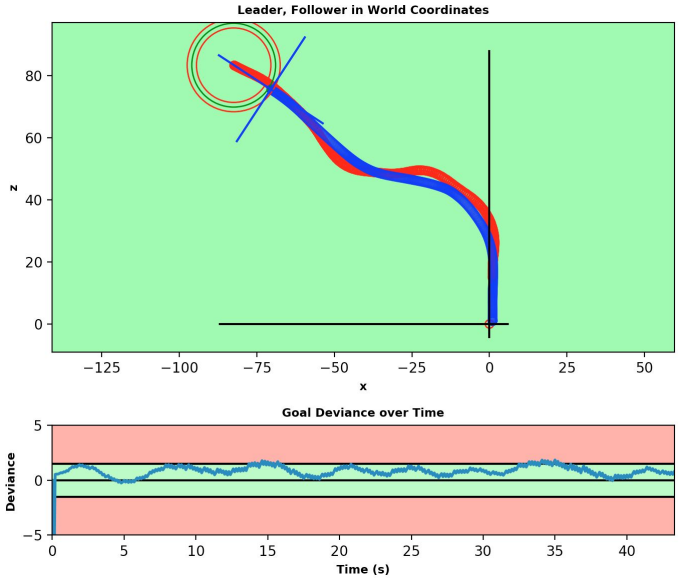

Figure 5: The Post Run Analysis of a run at "optimal" parameters. Note the deviation from optimal distance stays almost entirely within the

acceptable boundaries.

Such results could be improved in the future by implementing a better means of testing and refinement of the parameters, such that they could be optimized in a systemic, timely, and hardware-sparing manner. One suggestion would be to optimize the simulation itself, so that it were not so taxing on the CPU and could be run automatically on a script for long periods of time and with numerous iterations. An even faster means of training the optimal parameters would be to build a model of the Leader and simulation itself, one detached from the Unity-built app and without a GUI. Such a model would need to be built with not only the same constraints on randomly generated movement, but with similar acceleration and momentum properties as those imparted by Unity (but purely mathematically, that is) so that the movement properties were identical. With a similarly disembodied but accurate model of the Follower, the simulation could be run millions of times faster and the optimal parameters found easily through a grid or random search, as is typically done with many machine-learning model hyperparameters.

Another possible means of tuning the parameters would be to have a set of scripts that the Leader could follow that would variously test the different parameters in unique ways. For instance, the angular error controller gains could be tested with a script that has the Leader turn aggressively, or the derivative gain on the linear error could be tested with a Leader that brakes sharply and accelerates rapidly. These scenarios would have to be tuned to each parameter, and would have to contain some variation so that the parameter values are not overfit to the exact simulation, but they would provide more predictable scenarios to optimize the parameter values.

Extension to the Project - Multiple Rovers and Convoy Behaviour

SimGen enables a relatively easy extension to the simulated scenario (some bugs aside). Additional instances of the RoverMetaObject class, along with the TCP channels that will be used to communicate with them, can be declared in the .prototype file that SimGen uses to create Unity Asset scripts. Building these scripts in Unity allows rapid generation of a new simulation environment.

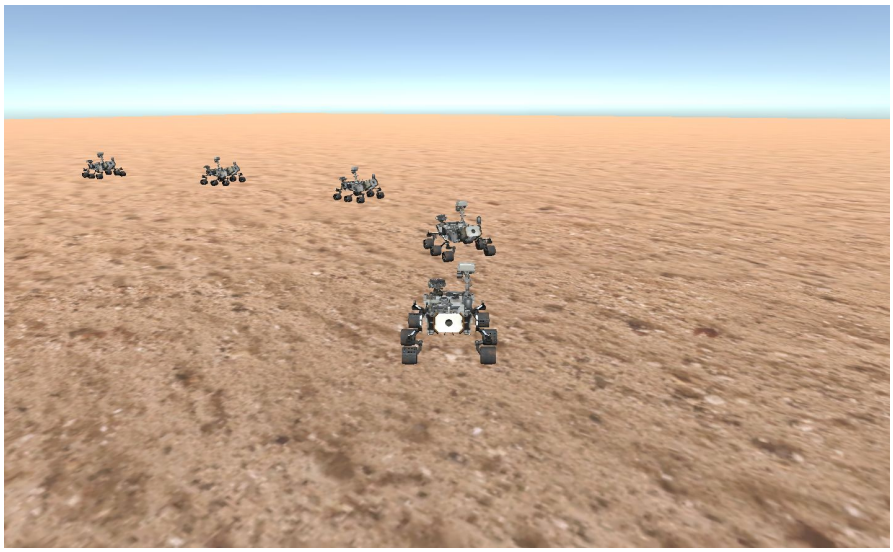

Figure 6: Multiple rovers following each other in a convoy-like behaviour.

With very few tweaks to the control program, these multiple rovers can be made to follow each other, in what emerges as a convoy-like behaviour. The architecture of a Supervisor class monitoring pairs of Rovers allows for easy extension of control of multiple robots with respect to the actions of their respective Leaders. Supervisor classes can be daisy chained to arbitrarily many Leader-Follower pairs, with the Follower of the $(i-1)^{th}$ pair acting as the Leader for the i^{th} pair.

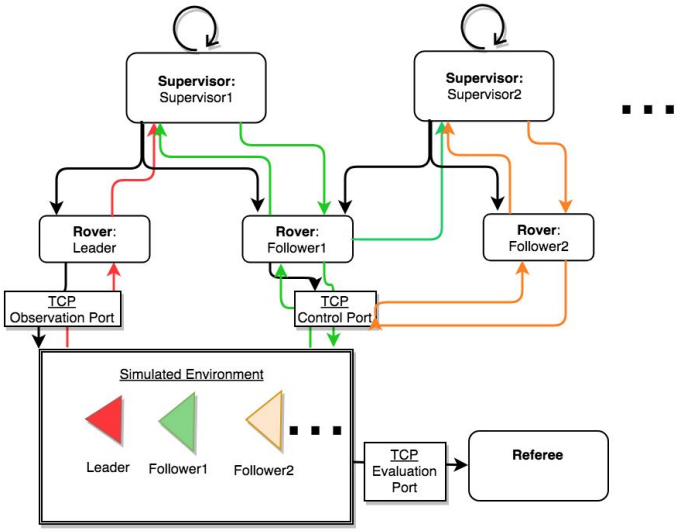

Figure 7: The daisy-chained *Supervisors*, with the Follower of the $(i-1)^{th}$ pair acting as the Leader for the i^{th} pair. This could theoretically be extended indefinitely.

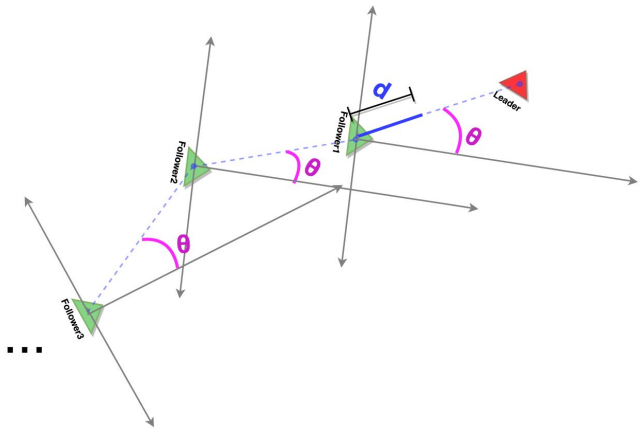

Figure 8: A diagram illustrating that the same principles are applied to each subsequent rover.

All of the same operations detailed in the preceding sections apply to these additional instances, as illustrated in **Figure 8**. The only additional necessary attention that must be paid in order to ensure the convoy behaviour is to what order the *Supervisor.execute()* functions are called in, as Follower[i+1] depends on the up-to-date location information stored in Follower[i].

There are some issues with the performance of the controller program when multiple rovers are being controlled (**Figure 9**). The first is that the controller is single-threaded, and thus the update, calculate error, and issue command loop is executed in sequence for each additional *Supervisor*. This leads to a highly inefficient delay between the updating of states and the necessarily-quick reaction for any given rover, as it has to wait for all additional rovers to finish their own control loops. Giving each *Supervisor* its own thread would allow the controller program to respond to each rover's needs at virtually the same time, as this would allow each *Supervisor* to make calls to the simulation at nearly the same time (relative to current implementation), calculate the errors in parallel, and then issue the commands in a similarly relatively quick manner. The second issue is that the controller program is written in Python, which, as an interpreted language, is not known for its quick reaction time. While Python proved useful for its simplicity in development, future iterations of this project could translate the controller-program to a more appropriate language, such as C++.

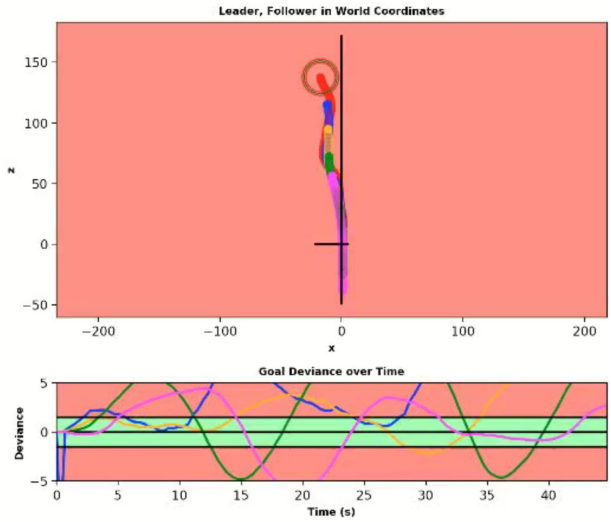

Figure 9: The rovers are not controlled as precisely as the single-pair scenario. This is due to the single-threaded, sequential order of execution of each Supervisor's control loop.

Future Directions for Work

There are a number of ways this project could be expanded upon in future work. As stated previously, implementing the simulation more efficiently, either as a better optimized version of the current Unity-simulation, or as a program to mathematically emulate the current rover behaviour, could allow for rapid tuning of the optimal parameters for controlling the following rovers. The multiple rovers could be more efficiently implemented with multithreaded Supervisors in a more time-efficient language.

An additional area for expansion would be to reconfigure which rover-pairs are given to various *Supervisors* in the multi-rover scenarios. Such an experiment could be implemented with minor-to-moderate alterations to the current design of the controller-program, in which one Leader has multiple Followers, and in which this one-to-many relationship is repeatedly numerous times for all of the rovers. The alterations needed would be for the rover's to have offset following patterns, as currently they would converge on the same path and be side-swiping each other. Another difficulty would be the current, slow, implementation of the multiple controllers, as this idea could not scale well with many rovers producing complex patterns of motion.

Another area for expansion would be the inclusion of more variables to account for in the simulation itself. For instance, rocks, cliffs, valleys, or other such obstacles could be included, which would necessitate the implementation of different controller behaviours in the mobile robot. The

simulation environment is rather unrealistic, in that any real-world mobile autonomous robot must have some form of goal-avoidance behaviour that is transitioned to in response to encountering an obstacle. The terrain of the environment could be varied, whether it be by elevation changes or the composition of the ground itself. Unity has a built-in physics system, which could easily be exploited to make moving up or down-hill or on patches of varying friction appropriately difficult, necessitating more complex movement calculations, or again transitions to entirely new behaviours in response to the changing environment.

Finally, yet another area for expansion, which was discussed a couple of times throughout the semester, would be to incorporate a human-controlled rover that acts as an additional variable that the rovers would have to respond to. Rather than just focusing solely on their respective leaders, the rovers would (again, more realistically) have to pay attention to all of their immediate surroundings to avoid collision. Real-world mobile autonomous robots, such as the Khepera line of compact robots[4], come equipped with an array of sensors that sweep around them for immediately encroaching items. Designing a rover that makes use of such a sensor array would be an exciting extension to the project.

References

1. Astrom, K.J., Murray, R.M.Analysis and Design of Feedback Systems: An Introduction for Scientists and Engineers, *Chapter 8 - PID Control* (2003). https://www.cds.caltech.edu/~murray/courses/cds101/fa04/caltech/am04_ch8-3nov04.pdf
2. Kansas State University, Robotics Programming Study Guide. 13.1.3. The Unicycle Model. http://faculty.salina.k-state.edu/tim/robotics_sg/Control/kinematics/unicycle.html
3. Egerstedt, M. Georgia Institute of Technology. Control of Mobile Robots, Module 2 - Mobile Robots.
4. K-Team. Khepera IV. https://www.k-team.com/khepera-iv

Appendix

Source Code

For the sake of brevity, source code has not been included here as screenshots, nor is it included as a zip-file when emailed due to the size of the zip-files (>33 MB allowed via Queen's Outlook). All of the source code may be found on my GitHub repository, at:

https://github.com/MarshallRuse/Rover-Follower-Challenge

Notably, there are 2 branches, the master branch for the base project, and a 'convoy' branch for the project extension.