Project Report

Abstract

This report documents the results and findings of the Co-modelling Exercise project. The aim of this project is to inject a fault into the GPS unit of an autonomous farming vehicle, and then use Fault Tolerance to take appropriate measures in order to deal with the fault. The vehicle is known as Robotti and was developed by AggroIntelli in Denmark. The model will be tested through co-simulation, a combination of the INTO-CPS tool, and the VDM-RT programming environment. Through co-simulation, faults can be injected into the model in a safe environment, allowing us to simulate how the model handles faults, reflecting its potential behaviour when it is constructed. By implementing dynamic redundancy into the system, the Robotti better handles disruptions to its intended route. Although my implementation of fault tolerance does not perfectly handle both types of GPS disruptions, it significantly reduces the effect of the injected fault. Through dynamic redundancy, the Robotti does not significantly stray from its intended path, avoiding unintentional complications such as driving into rivers, rough terrain, or another farmer's field. This project demonstrates the benefits of modelling and testing of fault tolerance before real world deployment.

Fault Injection

Intentional Spoofing

In order to implement GPS Spoofing into the controller model, I chose to inject the fault within the SensorGNSS class. In particular, I targeted the Sync method to alter the values that it receives for x, y, and theta. I implemented two types of GPS Spoofing, purposeful and natural spoofing. In order to recreate the effects of intentional, and malicious GPS Spoofing, I created the ‘IntentionalSpoofing’ method.

As seen in the code snippet below, this method uses the ‘MATH’ library to randomly generate a value between 0 and 100 and store it in the ‘spoofTheta’ value. I then assign it to the third element of the sequence ‘local_val’, which holds the Robotti’s ‘Theta’ value. Finally, I return this randomly generated value to use later in the ‘Sync’ method to manipulate the rotational movement of the Robotti.

```
30      public IntentionalSpoofing: () ==> real
31      IntentionalSpoofing() == (
32          spoofTheta := (MATH`rand(100));
33          local_val(3) := spoofTheta;
34          return spoofTheta;
35      );
```

I then modified the Sync method and set the fault to be injected if 10 simulated seconds have passed, continuously replacing the Theta value of the GPS with the new spoofed theta value.

```
42      public Sync: () ==> ()
43      Sync() == cycles(20)(
44
45          local_val := [i_x.get(), i_y.get(), i_theta.get(), 0.0];
46
47          -- Intentional Spoofing
48          if time/1e9 >= 10.0 then (
49              local_val(3) := spoofTheta;
50          );
51
```

I chose to continuously apply the intentional spoofing method for the rest of the simulation to mirror real-world incidents. During my research of GPS Spoofing, I found that whenever

intentional, malicious GPS spoofing occurs, it is usually continuous, rather than periodic or a one time occurrence. If it were periodic, it would change the values at regular intervals, causing obvious and sudden changes in the vehicle's intended path. On the other hand, if it were a one time occurrence, the Robotti's GPS would be altered, but would soon correct itself and return to normal functionality. In order to be prepared for the malicious spoofing that occurs in real life, it is important to test Robotti's ability to handle continuous GPS spoofing.

Natural Spoofing

In addition to intentional spoofing, I created the 'NaturalGPSSpoofing' method. When researching GPS spoofing in detail, I found that the probability of intentional, malicious GPS spoofing taking place, is very low. This is especially true in the case of Robotti. This is because intentional GPS spoofing requires high-grade technical skill to successfully deter it from its path. There is little value in maliciously targeting an autonomous farming vehicle like Robotti in comparison to what high-grade spoofing skill would be usually used for, such as military vehicles and weapons. Therefore, I decided to implement natural spoofing alongside it as it is a more realistic and common occurrence that affects GPS systems. Although this would not be ideal behaviour, small external disruptions would be considered Realistic behaviour, taking natural uncontrollable factors into account.

In order to simulate natural spoofing, I declared three instance variables at the top of the SensorGNSS class: offsetX, offsetY, and offsetTheta.

```
10      protected offsetX : real := 0.0;
11      protected offsetY : real := 0.0;
12      protected offsetTheta : real := 0.0;
```

Below, I then created the 'NaturalSpoofing' method. Similarly to my PurposefulGPSSpoofing method, I randomly generated a value, and assigned them to values offsetX, offsetY, and offsetTheta to use later. From my research, I found that offsets in real life can be both positive or negative. To better mirror real life scenarios, I adjusted the random value generation to generate numbers from 0 to 20, then subtracting 10 from the final result, creating the range of -10 to 10.

```
35      public AccidentalSpoofing: () ==> ()
36      AccidentalSpoofing() == (
37          offsetX := MATH`rand(20) - 10;
38          offsetY := MATH`rand(20) - 10;
39          offsetTheta := MATH`rand(20) - 10;
40      );
```

I selected to implement GPS Spoofing due to Robotti's heavy dependability on the GPS system. If the Robotti were to receive inaccurate GPS readings, the robot could greatly deter from its intended path, causing it to think that it is in the wrong place. This could lead to severe consequences such as Robotti over-correcting its path, straying into another farmer's land, driving into terrain it isn't built to handle like rivers and rocky land, crashing…etc. It is important that Robotii is built to handle natural disruptions as "One of the most critical issues in the development of autonomous vehicles and driver assistance systems is their poor performance under adverse weather conditions, such as rain, snow, fog, and hail."[1]

This potential damage to the farm could cost the farm owner significantly in both the agricultural aspect of having to replant, and care for damaged crops, but also for the fixing or replacement of the Robotti itself.

[1] IEEE Vehicular Technology Magazine, The impact of adverse weather conditions on autonomous vehicles: how rain, snow, fog, and hail affect the performance of a self-driving car

Fault Tolerance

In order to help overcome the faults that I implemented, I created a method that adds Dynamic Redundancy. In the method ‘isValid’, I check to see if the coordinates received are within a certain range. To determine these values, I ran the model without any faults injected, and analysed the graphs to see the minimum and maximum values for each coordinate, as that establishes the correct range that the coordinates should be in between. If the data received from the GPS is valid, return true.

```
35      public isValid: seq of real ==> bool
36      isValid(coordinates) == (
37          if (coordinates(1) > -13 and coordinates(1) < 8) and
38             (coordinates(2) > -4 and coordinates(2) < 18) and
39             (coordinates(3) > -5 and coordinates(3) < 0.2) then
40             return true
41          else
42              return false
43      );
```

I use ‘isValid’ in the Sync method. If the coordinates from local_val are valid, assign those coordinates to backup_val. This provides a series of ‘correct’ coordinates to fall back to in case future coordinates are invalid. I then update valid_data to hold the current valid data.

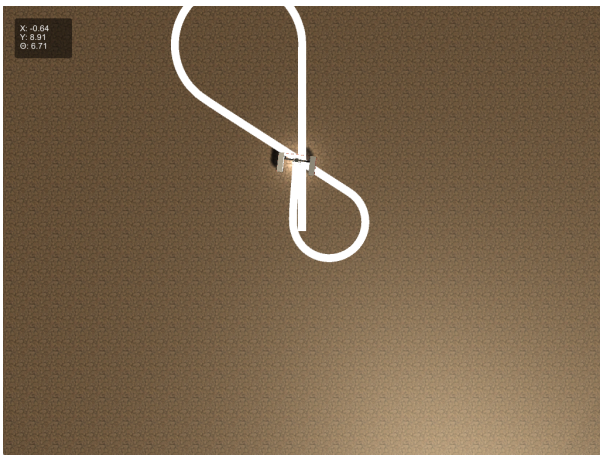
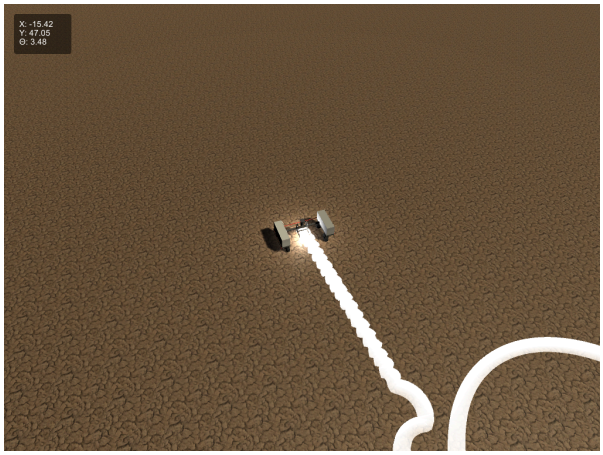
```
59      public Sync: () ==> ()
60      Sync() == cycles(20)(
61
62          local_val := [i_x.get(), i_y.get(), i_theta.get(), 0.0];
63
64          if isValid(local_val) then (
65              backup_val := local_val;
66              valid_data := local_val;
67          ) else (
68              valid_data := local_val;
69          );
```

I chose to implement Dynamic Redundancy specifically as I deemed it best suited for the type of fault I injected. By consistently checking the coordinates of the Robotti and seeing if it was in a reasonable range, the system can either store the coordinates and mark them as ‘valid’, and if the current coordinates aren’t within a reasonable range, it can dynamically switch to the last

valid coordinates. This allows the Robotti to detect faults in real-time, and still stay somewhat on its intended course.

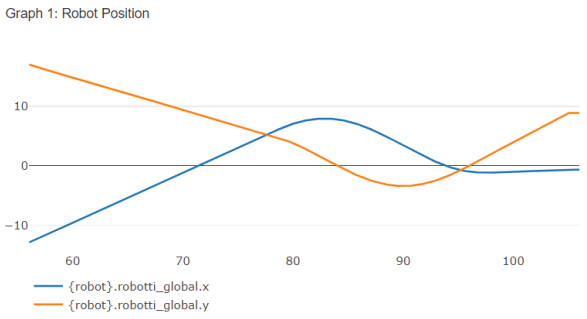
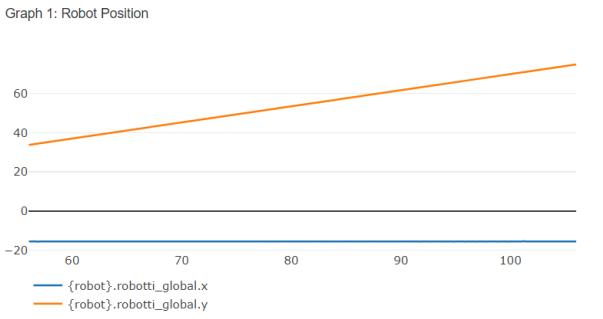
Results and Evaluation

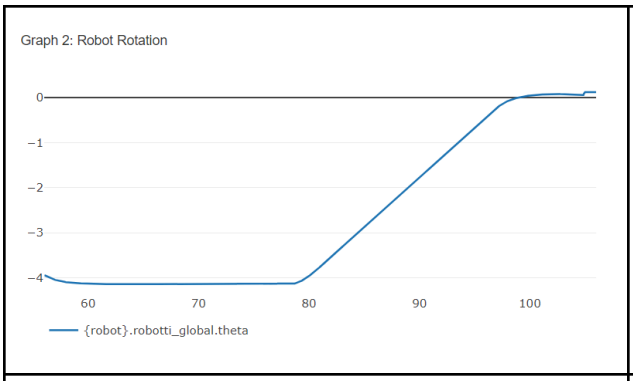
Fault Injection (Intentional Spoofing)

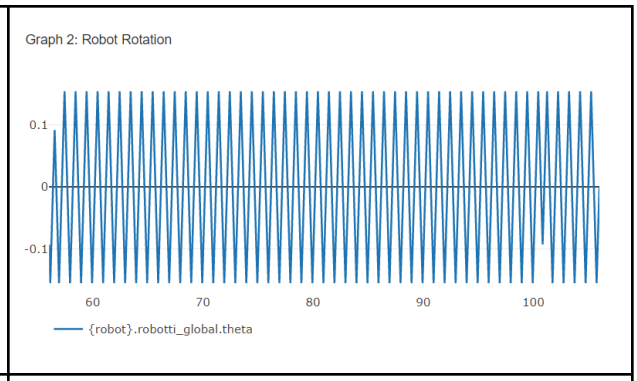
Route before Fault Injection (Intentional)	Route after Fault Injection (Intentional)
	

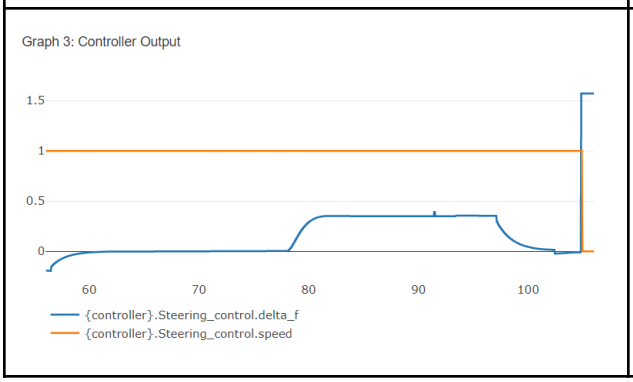
As seen from the screenshots above, before the intentional spoofing fault was injected, the Robotti performed a clean figure of 8, as expected behaviour. After the fault was injected, it drives as expected for 10 simulated seconds, before driving in two small curves in the opposite direction, before driving in small and sharp zig-zags for the rest of the simulation. This accurately reflects the fault that I injected into the model, as it simulates incorrect theta values, rotating and deterring the vehicle away from its intended route.

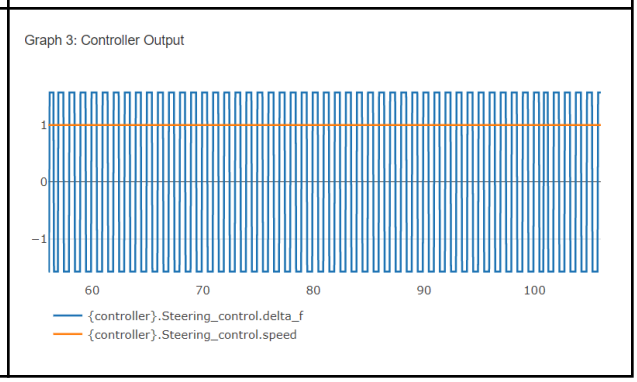
The change in the model post fault injection are further supported by the final graph results after the simulation had ended:

Graphs before Fault Injection (Intentional)	Graphs after Fault Injection (Intentional)
	

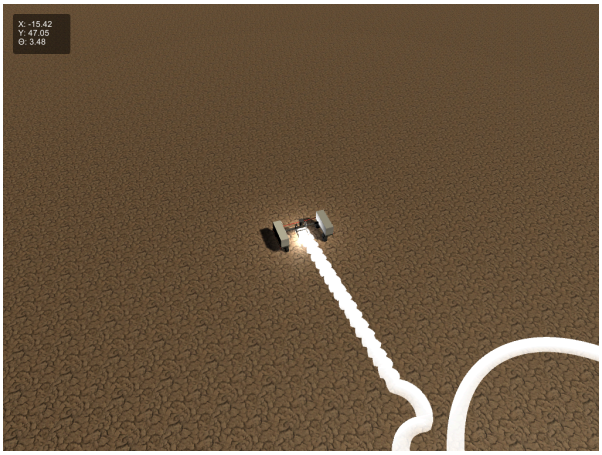
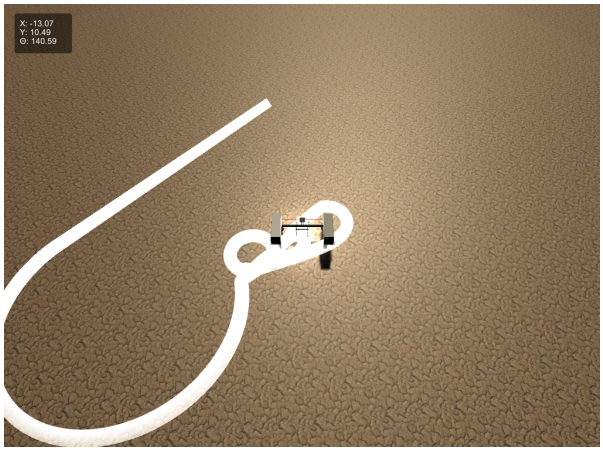

Graph 2: Robot Rotation
0
−1
−2
−3
−4
60
70
80
90
100
{robot}.robotti_global.theta


Graph 2: Robot Rotation
0.1
0
-0.1
60
70
80
90
100
{robot}.robotti_global.theta


Graph 3: Controller Output
1.5
1
0.5
0
60
70
80
90
100
{controller}.Steering_control.delta_f
{controller}.Steering_control.speed

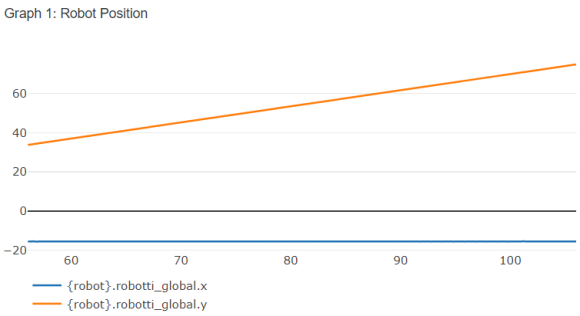
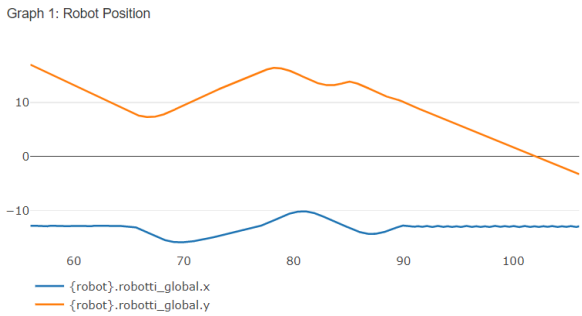

Graph 3: Controller Output
1
0
−1
60
70
80
90
100
{controller}.Steering_control.delta_f
{controller}.Steering_control.speed

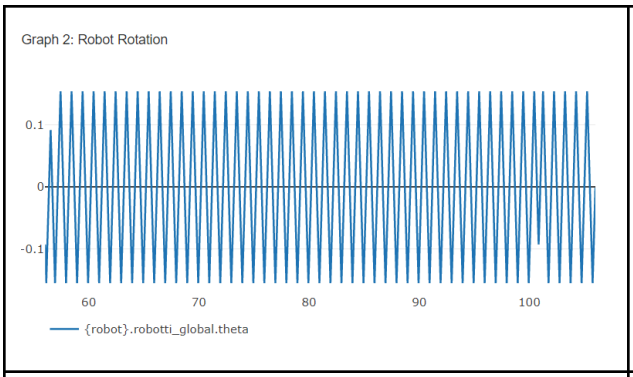
Fault Tolerance (Intentional Spoofing)

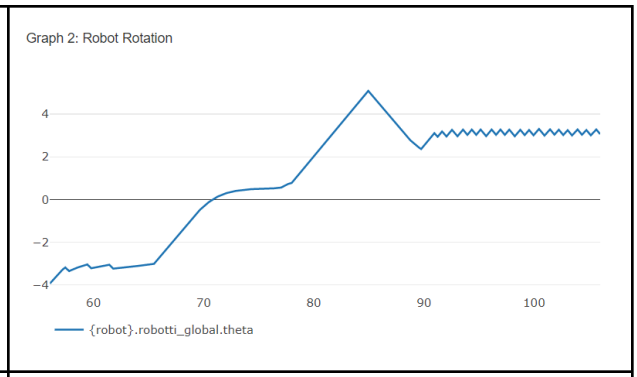
Route before Fault Tolerance (Intentional)	Route after Fault Tolerance (Intentional)
	

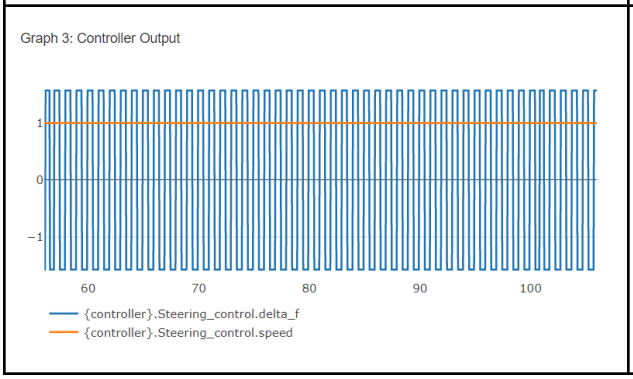
After implementing dynamic redundancy, the Robotti now finishes the first half of the figure of 8, before becoming driving in small loops, attempting to find the next waypoint.

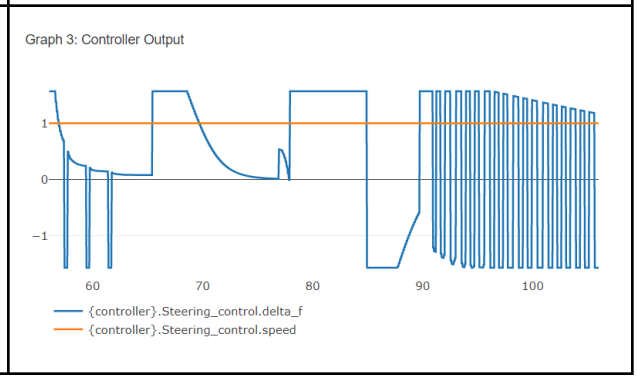
Although it did not fully correct the path, the impact of dynamic redundancy is shown through the graphs below:

Graphs before Fault Tolerance (Intentional)	Graphs after Fault Tolerance (Intentional)
	

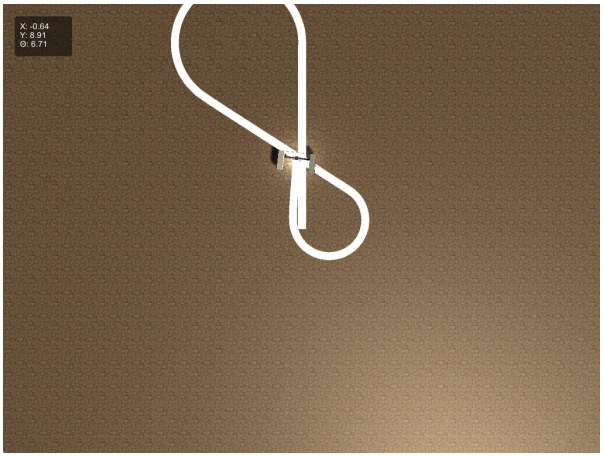
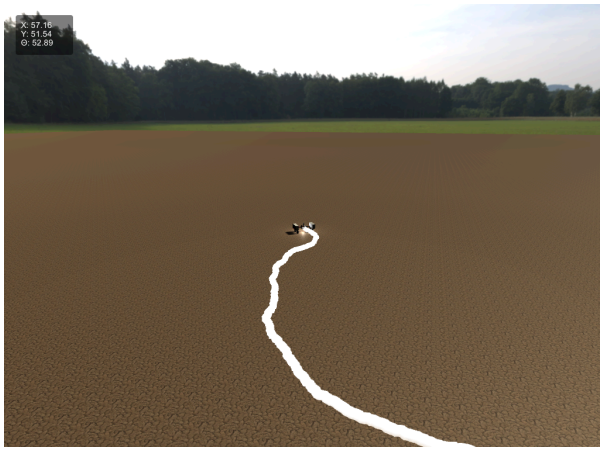

Graph 2: Robot Rotation
0.1
0
-0.1
60
70
80
90
100
{robot}.robotti_global.theta


Graph 2: Robot Rotation
4
2
0
−2
−4
60
70
80
90
100
{robot}.robotti_global.theta


Graph 3: Controller Output
1
0
−1
60
70
80
90
100
{controller}.Steering_control.delta_f
{controller}.Steering_control.speed

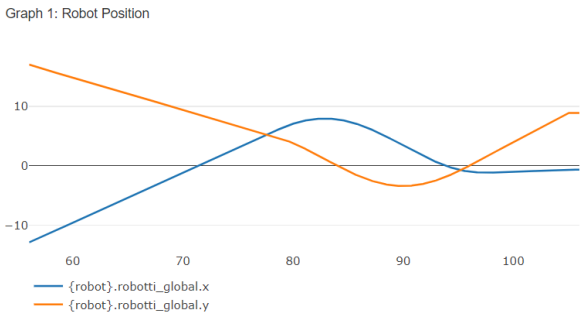
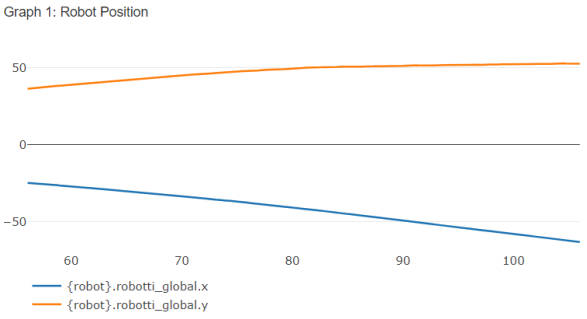

Graph 3: Controller Output
1
0
−1
60
70
80
90
100
{controller}.Steering_control.delta_f
{controller}.Steering_control.speed

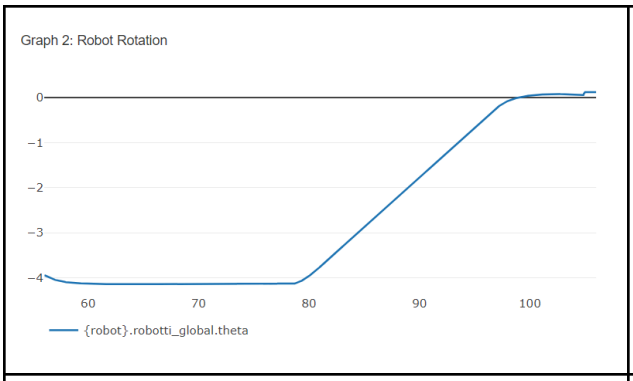
Fault Injection (Natural Spoofing)

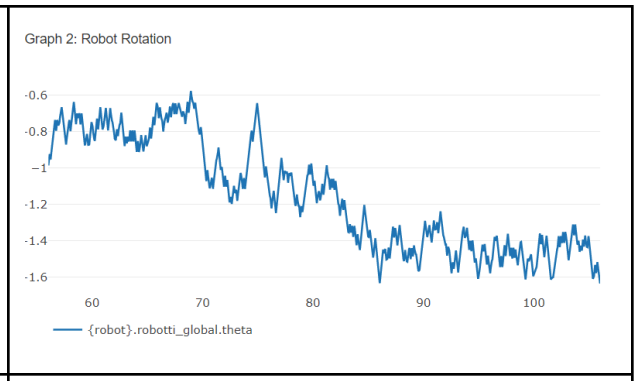
Route before fault injection (Natural)	Route after fault injection (Natural)
	

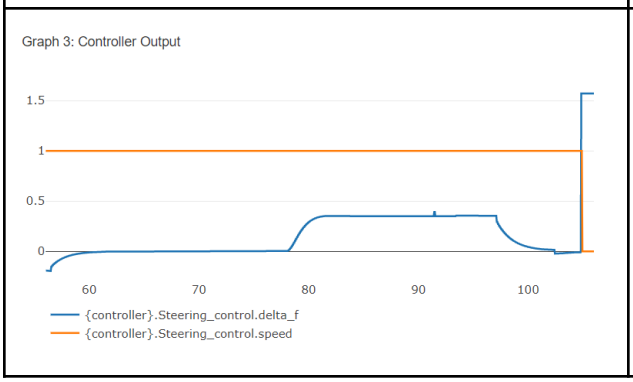
Similarly to intentional spoofing, the Robotti drives in a clean figure of eight before the fault is injected. After the natural spoofing fault is injected, it begins to drift off in a random direction using the natural offsets I implemented between the range of -10 and 10. The screenshot above reflects its real-world behaviour under harsh weather conditions or rough terrain.

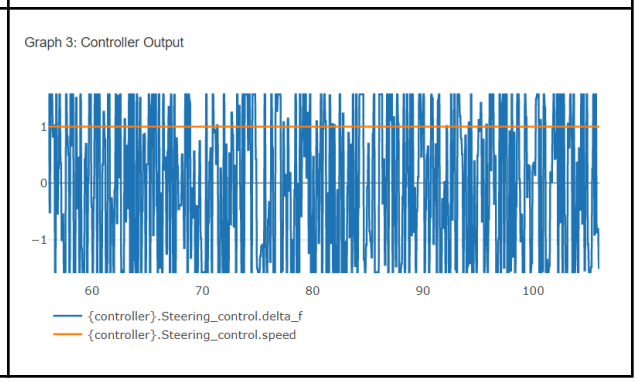
This is further supported by the graphs below:

Graphs before Fault Injection (Natural)	Graphs after Fault Injection (Natural)
	

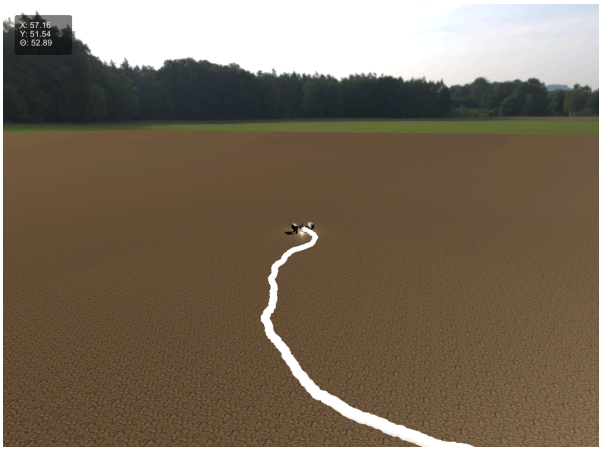
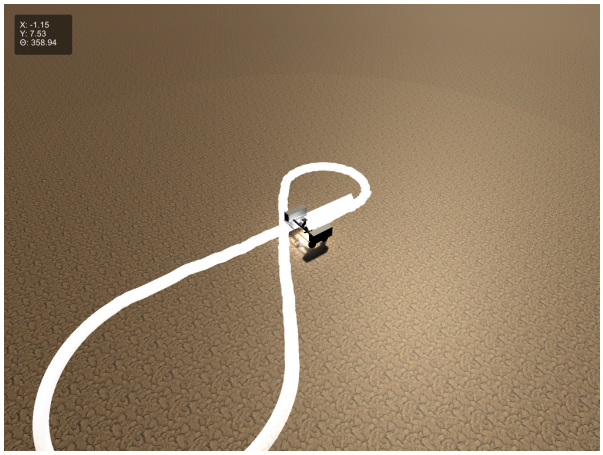

Graph 2: Robot Rotation
0
−1
−2
−3
−4
60
70
80
90
100
{robot}.robotti_global.theta


Graph 2: Robot Rotation
-0.6
-0.8
−1
-1.2
-1.4
-1.6
60
70
80
90
100
{robot}.robotti_global.theta


Graph 3: Controller Output
1.5
1
0.5
0
60
70
80
90
100
{controller}.Steering_control.delta_f
{controller}.Steering_control.speed

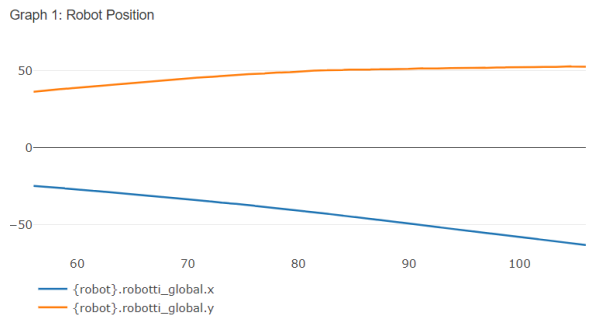
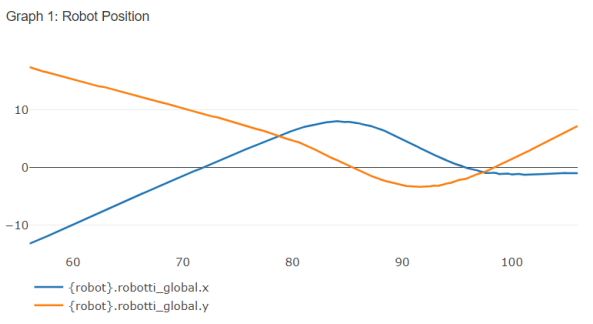

Graph 3: Controller Output
1
0
−1
60
70
80
90
100
{controller}.Steering_control.delta_f
{controller}.Steering_control.speed

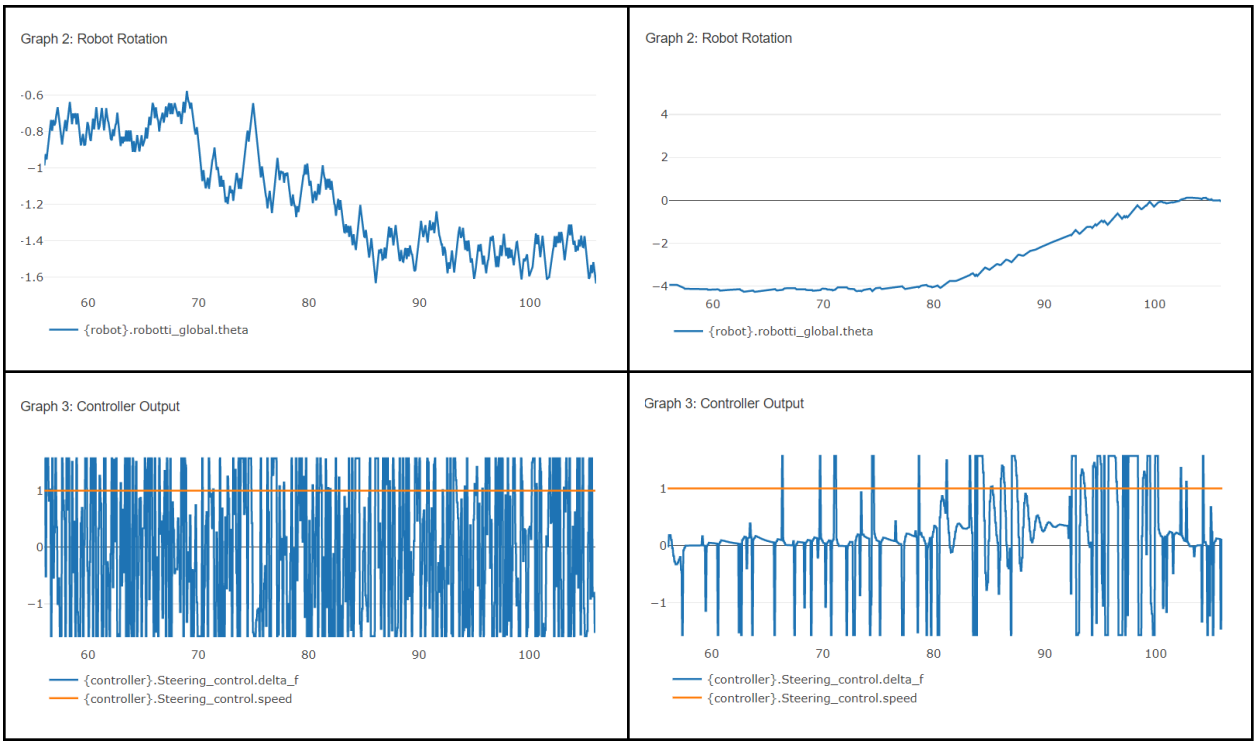
Fault Tolerance (Natural Spoofing)

Route before Fault Tolerance (Natural)	Route after Fault Tolerance (Natural)
	

Before I added dynamic redundancy, the Robotti would randomly drift off due to the added offsets to its original coordinates. However, after adding fault tolerance, the Robotti forms a figure of eight close to the original, with slight wobbles and shakes throughout.

The graphs below support the fault tolerance added and somewhat closely resembles the original graphs before the faults were injected.

Graphs before fault tolerance (Natural)	Graphs after fault tolerance (Natural)
	



I was satisfied with the final results of both the fault injection and fault tolerance mechanisms. The implementation of both intentional spoofing and natural disruptions accurately reflected the impacted behaviour of these faults, were they to occur in the real-world. I was especially pleased with the added dynamic redundancy. It significantly corrected the natural disruption fault, its route being very close to the original figure of eight. Although it did not completely correct the intentional spoofing fault, it did keep the Robotti within reasonable bounds.

Conclusions

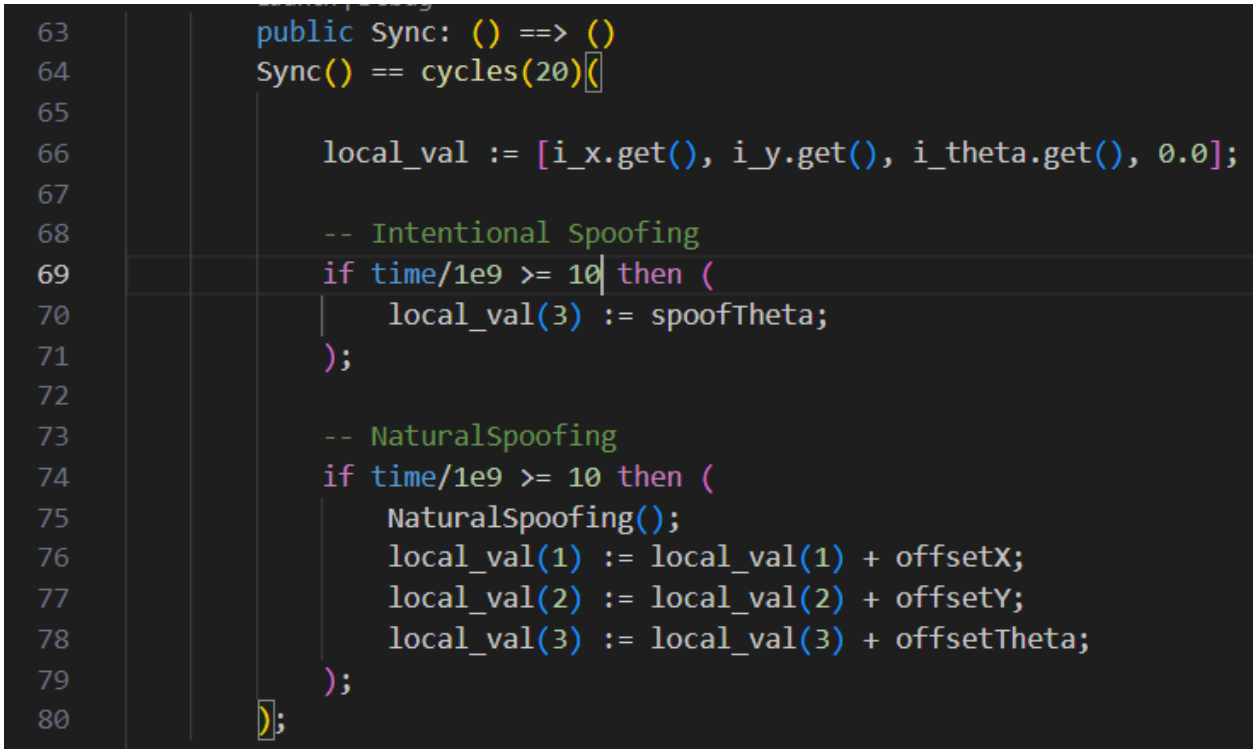
Overall, I am satisfied with the implementation of my chosen fault injection and fault tolerance. The GPS fault that I implemented and modelled for the Robotti closely resembled the effect of the real-world counterpart, and I was pleased to see my fault tolerance method counter the GPS spoofing that I injected against it. Through this project, I have significantly broadened my understanding of fault tolerance and how faults can be modelled and tested to improve the reliability of the final product.

Although my fault tolerance implementation does not fully counter the injection, I am satisfied that it works most well with the natural spoofing over the intentional one, as the natural disruption of the GPS signal is more realistic and much more probable to occur to an autonomous farming vehicle like Robotti.

Despite this, my implementation has many areas of improvement. If I were to attempt a problem through co-simulation in the future, I would spend more time developing a complete understanding of the syntax and environment, which in turn would mean that I would spend less time attempting to implement features and fix simple errors. Additionally, I would like to improve the ‘NaturalGPSSpoofing’ method to better simulate natural GPS Spoofing in the real world. If the Robotti were to experience harsh weather conditions, it may slide on wet land, or a wheel may become lodged and stuck in mud. These unaccounted for conditions could cause issues where some coordinates might stay on the same value, while the others increase and decrease at different rates. By incorporating these factors into my future projects, I can more realistically model and simulate the effect of GPS faults.

Appendix

To change the time that the fault occurs in my model, edit line 69 or line 74, commenting out the type of spoofing you don't want to execute.



```
63          public Sync: () ==> ()
64          Sync() == cycles(20)(
65
66              local_val := [i_x.get(), i_y.get(), i_theta.get(), 0.0];
67
68              -- Intentional Spoofing
69              if time/1e9 >= 10 then (
70                  local_val(3) := spoofTheta;
71              );
72
73              -- NaturalSpoofing
74              if time/1e9 >= 10 then (
75                  NaturalSpoofing();
76                  local_val(1) := local_val(1) + offsetX;
77                  local_val(2) := local_val(2) + offsetY;
78                  local_val(3) := local_val(3) + offsetTheta;
79              );
80          );
```