FAULT TOLERANCE SIMULATION REPORT

BY
ISHAN KALRA - 220626363

ABSTRACT:

The aim of this project is the implementation of fault injection and tolerance mechanisms in an autonomous navigation control robot tractor for farming purposes, the model called (Foldager, 2015). The fault injection techniques applied were GPS signal noise, loss of signal, and data corruption. A fault tolerance mechanism was then developed to implement a mechanism for a forward valid safe state. A route correction and backup value recovery functionality were introduced, thus allowing the system to recover after faults. The results of running this fault injection were quite extreme navigation deviations, but the fault tolerance system partly reduced them so that the system could enter a valid safe state. Conclusions on the effectiveness of the implemented approach and suggestions for future improvements are made for this report.

FAULT INJECTION:

Fault injection was applied to the GPS unit of the controller model to simulate realistic navigation challenges. Three types of faults were introduced:

1. **Random Noise:** Added small deviations to GPS coordinates.
2. **Signal Loss:** Simulated GPS signal failure with a 5% probability.
3. **Outliers:** Introduced extreme values into GPS data with a 1% probability.

CASE BY CASE COMPARISON:

Two approaches were made for simulating faults and to observe different methods of simplified fault injection and creating a redundant system for it.

Case 1: We Injected fault in all 3 sensors – Robotti 1(Folder name)

Case 2: Only SensorGNSS_H was Injected with faults – Robotti 2(Folder name)

CASE 1:

Fault injections-

SensorGNSS_E:

```
            --Simulating system fault (noise, signal loss, outliers - basially hallucination)
             let randomNoise = if MATH`rand(100) > 80 then (MATH`rand(100)/2000.0 - 0.025 ) else 0 in
             let signalLoss = MATH`rand(100) > 0.95 in
             let outliers = MATH`rand(100) > 0.99 in
            return if signalLoss then -1
            else if outliers then (port.getValue()* (MATH`rand(100)*10))
            else (port.getValue()+ randomNoise);
        );

end SensorGNSS_E
```

SensorGNSS_H:

```
    Launch | Debug
    public get: () ==> real
            get() ==
            --Simulating system fault (noise, signal loss, outliers - basially hallucination)
             let randomNoise = if MATH`rand(100) > 80 then (MATH`rand(100)/2000.0 - 0.025 ) else 0 in
             let signalLoss = MATH`rand(100) > 0.95 in
             let outliers = MATH`rand(100) > 0.99 in
            return if signalLoss then -1
            else if outliers then (port.getValue()* (MATH`rand(100)*10))
            else (port.getValue()+ randomNoise);

end SensorGNSS_H
```

SensorGNSS_N:

```
Launch | Debug
public get: () ==> real
        get() ==
        --Simulating system fault (noise, signal loss, outliers - basially hallucination)
         let randomNoise = if MATH`rand(100) > 80 then (MATH`rand(100)/2000.0 - 0.025 ) else 0 in
         let signalLoss = MATH`rand(100) > 0.95 in
         let outliers = MATH`rand(100) > 0.99 in
        return if signalLoss then -1
        else if outliers then (port.getValue()* (MATH`rand(100)*10))
        else (port.getValue()+ randomNoise);

end SensorGNSS_N
```

EXPLANATION OF FAULT INSERTION:

- Random noise (randomNoise) simulates minor inaccuracies.
- Signal loss (signalLoss) forces the function to return -1 for missing data.
- Outliers (outlier) produce extreme, unrealistic values.

SIGNAL LOSS

- *MATH`rand()* library has been used to generate a random integer between 0 and 100. Values greater than 95 will simulate signal loss.

RANDOM NOISE

- The noise is computed using *MATH`rand()* and scaled down for small variations in sensor data.

OUTLIER SIMULATION

- Extreme values are simulated by multiplying the port value with a large random factor.

RETURN LOGIC

- If signal loss occurs (rand > 95), return -1.
- If an outlier occurs (rand > 99), scale the port value abnormally.
- Otherwise, add the simulated noise.

CASE 2:

Fault Injection-

SensorGNSS_H:

```
class SensorGNSS_H

instance variables
    port: RealPort;
    isFaulty: bool := false; -- Indicates if the fault is active
    faultStartTime: nat := 10; -- Fault activation time in seconds
    elapsedTime: nat := 0; -- Tracks elapsed simulation time

operations
    -- Constructor
    public SensorGNSS_H: RealPort ==> SensorGNSS_H
    SensorGNSS_H(p) ==
    (
        port := p
    );

    -- Get sensor value with fault simulation
    Launch | Debug
    public get: () ==> real
    get() ==
    (
        if elapsedTime >= faultStartTime and isFaulty
        then
            -- Simulating system fault (noise, signal loss, outliers)
            let randomNoise = if MATH`rand(100) > 80 then (MATH`rand(100)/2000.0 - 0.025) else 0 in
            let signalLoss = MATH`rand(100) > 95 in
            let outliers = MATH`rand(100) > 99 in
            return if signalLoss then -1
                   else if outliers then (port.getValue() * (MATH`rand(100) * 10))
                   else (port.getValue() + randomNoise)
        else
            return port.getValue()
    );

    -- Activate fault
    Launch | Debug
    public activateFault: () ==> ()
    activateFault() ==
    (
        isFaulty := true
    );
```

```
-- Deactivate fault
Launch | Debug
public deactivateFault: () ==> ()
deactivateFault() ==
(
    isFaulty := false
);

-- Update elapsed time (simulated time progression)
Launch | Debug
public updateTime: nat ==> ()
updateTime(seconds) ==
(
    elapsedTime := elapsedTime + seconds
);

-- Reset elapsed time
Launch | Debug
public resetTime: () ==> ()
resetTime() ==
```

Detailed Analysis of codeword:

get() function-

- **Purpose**: Retrieves the sensor's value, optionally simulating faults if conditions are met.
- **Fault Types**:
  - **Random Noise**: Small deviations in the reading.
  - **Signal Loss**: Returns -1 to indicate a loss of signal.
  - **Outliers**: Extreme, unrealistic values generated to simulate hardware glitches.
- **Trigger Conditions**:
  - Faults are applied only if elapsedTime >= faultStartTime and isFaulty = true (failed to make it work, deleted from actual codework)
  - **Normal Behavior**: Returns the current value from port.getValue() if the sensor is not faulty or the timer hasn't reached the fault activation time.
  - **Fault Behavior**:
    - **Signal Loss**: Simulates a complete failure by returning -1.
    - **Outliers**: Multiplies the port value by a large random factor to create extreme values.
    - **Random Noise**: Adds a small random offset to the port value.

I aimed to integrate the following trigger functions which would fulfill the requirement of having a fault trigger and trigger the fault after 10 elapsed seconds:

-activateFault()

- **Purpose**: Activates the fault simulation mechanism by setting isFaulty := true.

-deactivateFault()

- **Purpose**: Deactivates fault simulation by setting isFaulty := false.

-updateTime(seconds)

- **Purpose**: Simulates the progression of time by incrementing the elapsedTime variable.

-resetTime()

- **Purpose**: Resets the elapsed time counter to 0.

REASON FOR FAULT INSERTION SELECTION:

The Failure Mode Effects Analysis technique or FMEA used in this scenario was from SHARD from Collaborative designs (John Fitzgerald, 2010).

Following was based out of a shard analysis:

System Component	Description	Potential Issues	Validation/Testing Approach
Sensor Components	The individual components that measure positional data: - **SensorGNSS_E** (East direction) - **SensorGNSS_N** (North direction) - **SensorGNSS_H** (Heading or orientation).	- Sensor failure (returns constant value). - Signal noise (random deviations in values). - Incorrect calibration (systematic offsets). - Fault not activating or deactivating as expected. - Incorrect fault offsets applied.	Inject faults using random noise or offsets in SensorGNSS_E, SensorGNSS_N, and SensorGNSS_H. Log the effect on the
Data Aggregator	Aggregates data from the individual sensor components into a composite reading (read() in SensorGNSS).	- Data mismatch (components return incompatible values). - Faulty aggregation logic (improper computation).	Use unit tests to validate the logic of the read() method. Ensure results align with expected sensor inputs.
Interface with Controller	Passes aggregated GPS data to the main controller (e.g., for navigation).	- Data loss during transmission. - Latency in providing positional data.	Simulate scenarios where data updates are delayed or lost. Verify the controller's ability to handle stale or
Hardware Interface	Represents the connection to physical hardware (e.g., GPS antenna, processor).	- Connection failure (no data received). - Signal degradation (poor GPS signal quality).	Simulate disconnection or poor signal and check system response (e.g., switching to fallback modes).

Fault Injection System	The mechanism to introduce faults into the GPS readings, either randomly or systematically.	- Fault not activating or deactivating as expected	Validate with controlled inputs (e.g., manually activate/deactivate faults and check output consistency).

FAULT TOLERANCE:

For Case 1:

Code work

```
class SensorGNSS

instance variables
    public static i_x : [SensorGNSS_E] := nil;
    public static i_y : [SensorGNSS_N] := nil;
    public static i_theta : [SensorGNSS_H] := nil;
    protected local_val: seq of real := [0.0, 0.0, 0.0, 0.0];
    protected route : [Route] := nil; -- Reference to the Route class
    protected correction_threshold : real := 0.5; -- Deviation threshold

operations
    public SensorGNSS: SensorGNSS_E * SensorGNSS_N * SensorGNSS_H * Route ==> SensorGNSS
    SensorGNSS(x, y, theta, r) == (
        i_x := x;
        i_y := y;
        i_theta := theta;
        route := r;
    );

    Launch | Debug
    public read: () ==> seq of real
    read() == (
        Sync();
        return local_val;
    );

```

```
    Launch | Debug
    public Sync: () ==> ()
    Sync() == cycles(20)(
        -- Fetch GPS values
        let x = i_x.get(),
            y = i_y.get(),
            theta = i_theta.get()
        in
            -- Correct deviations and update local_val
            local_val := correctDeviation(x, y, theta)
    );

    private correctDeviation: real * real * real ==> seq of real
    correctDeviation(x, y, theta) ==
    (
        -- Fetch current and next waypoints from the route
        dcl current_wp : WayPoint := route.getCurrentRouteElement();
        dcl next_wp : WayPoint := route.getNextRouteElement();

        -- Get coordinates of the next waypoint
        dcl target_x : real := next_wp.PosX();
        dcl target_y : real := next_wp.PosY();

        -- Compute the deviation from the route
        dcl deviation : real := MATH`sqrt((x - target_x) ** 2 + (y - target_y) ** 2);
```

```
        -- Compute the deviation from the route
        dcl deviation : real := MATH`sqrt((x - target_x) ** 2 + (y - target_y) ** 2);

        if deviation > correction_threshold then
            -- Correct trajectory toward the next waypoint
            let corrected_x = x + (target_x - x) * 0.1, -- Gradual adjustment
                corrected_y = y + (target_y - y) * 0.1,
                corrected_theta = MATH`atan2(target_y - y, target_x - x)
            in
                return [corrected_x, corrected_y, corrected_theta, 0.0]
        else
            -- Stay on the current trajectory
            return [x, y, theta, 0.0]
    );
end SensorGNSS
```

Explanation

The **fault-tolerant system** developed for the autonomous navigation controller ensures reliable operation despite injected faults like random noise, signal loss, and outliers. A Forward Valid state mechanism was attempted here. Here is how the system works:

Mechanisms in Detail

1. Validation and Backup Recovery

This mechanism ensures the system can handle invalid or extreme sensor readings:

- **How it works:**

- Each sensor reading (e.g., GPS coordinates) is checked for validity.
- If a fault (e.g., signal loss or outlier) is detected, the value is replaced with a backup value stored from the last valid reading.

. Route Correction

This mechanism adjusts the tractor's trajectory based on the waypoints in the predefined route:

- **How it works:**
  - Calculates the deviation between the current position and the next waypoint.
  - If the deviation exceeds the threshold, the system adjusts the position and heading to steer back toward the waypoint.

For CASE 2:

Codework-

sens > SensorGNSS.vdmrt > SensorGNSS > Sync

```
class SensorGNSS

instance variables
    public static i_x: [SensorGNSS_E] := nil;
    public static i_y: [SensorGNSS_N] := nil;
    public static i_theta: [SensorGNSS_H] := nil;
    protected local_val: seq of real := [0.0, 0.0, 0.0, 0.0];
    protected history: seq of seq of real := []; -- History of valid readings
    protected maxHistory: nat1 := 10; -- Maximum size of history
operations
    public SensorGNSS: SensorGNSS_E * SensorGNSS_N * SensorGNSS_H ==> SensorGNSS
    SensorGNSS(x, y, theta) ==
    (
        i_x := x;
        i_y := y;
        i_theta := theta;
    );
     Launch | Debug
    public read: () ==> seq of real
    read() ==
    (
        Sync(); -- Synchronize the sensor values
```

```
         Launch | Debug
        public Sync: () ==> ()
        Sync() == cycles(20)(
            let new_x = validate(i_x.get(), 1),
                new_y = validate(i_y.get(), 2),
                new_theta = validate(i_theta.get(), 3)
            in
            (
                local_val := [new_x, new_y, new_theta, 0.0];

            )
        );
        -- Validate individual sensor values
    private validate: real * nat1 ==> real
    validate(value, index) ==
    (
        return if isFaulty(value)
```

```
-- Check if a value is faulty
private isFaulty: real ==> bool
isFaulty(value) ==
  (  return value = -1 or abs(value) > 1000.0; -- Example: outlier detection
  );

  -- Get fallback value from history
private fallbackValue: nat1 ==> real
fallbackValue(index) ==
(
    return if history = [] or index > len history(1)
           then 0.0 -- Default fallback if no history is available
           else history(len history)(index)
);
end SensorGNSS
```

Explanation: As previously stated for case1.

Special functions for this one-

-Sync()

- **Purpose**: Synchronizes and validates individual sensor readings.
- **Details**:
    - Retrieves raw readings using get() from i_x, i_y, and i_theta.
    - Calls validate() to check each reading for faults and replace faulty values with fallback data.
    - Updates local_val with the validated values.
- **Fault Handling**:
    - Detects faults in individual sensor values and replaces them with fallback values using validate().

- Validate(value, index)

- **Purpose**: Validates individual sensor readings and handles faulty data.
- **Details**:
    - Checks if the sensor value is faulty using isFaulty(value).
    - If faulty, retrieves a fallback value from history using fallbackValue(index).
    - Otherwise, returns the original sensor value.

- isFaulty(value)

- **Purpose**: Identifies whether a sensor reading is faulty.
- **Details**:
    - Considers a reading faulty if:
        - **Signal Loss**: Value is -1.

    - **Outlier**: Absolute value exceeds 1000.0.
- **Example**:
    - value = -1 → Faulty (signal loss).
    - value = 1200.0 → Faulty (outlier).

-FallbackValue(index)

- **Purpose**: Provides a fallback value for faulty sensor readings.
- **Details**:
    - If history is empty or the index is invalid, returns 0.0 as the default.
    - Otherwise, retrieves the most recent valid reading for the specified index from history

System Resilience mechanisms: I tried putting the same mechanisms in this codework to do the following-

- Fault Detection

- Implemented in isFaulty(value).
- Detects:
    - **Signal Loss**: Returns -1.
    - **Outliers**: Absolute value exceeding 1000.0.

- Fallback Mechanism

- Implemented in fallbackValue(index) and validate(value, index).
- Uses historical data (history) to replace faulty readings.
- Ensures system stability even during transient faults or sensor failures.

- History Maintenance

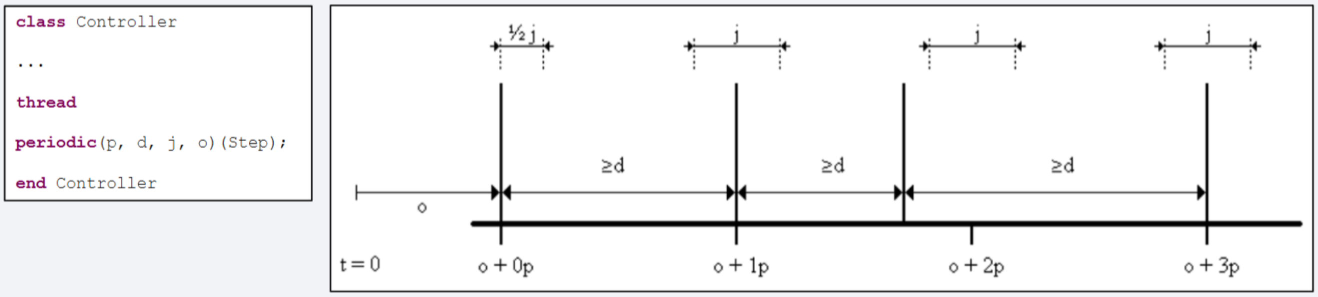
- Stores valid readings in history (not fully implemented in provided code but should be part of Sync()).
- Limits history size to maxHistory (10 readings).

-Data Aggregation

- Combines validated readings from i_x, i_y, and i_theta into local_val.
- Provides a single source of truth for the entire sensor system.

I also implemented a modelling drift into the codework to establish a time trigger-

Using the following design system:



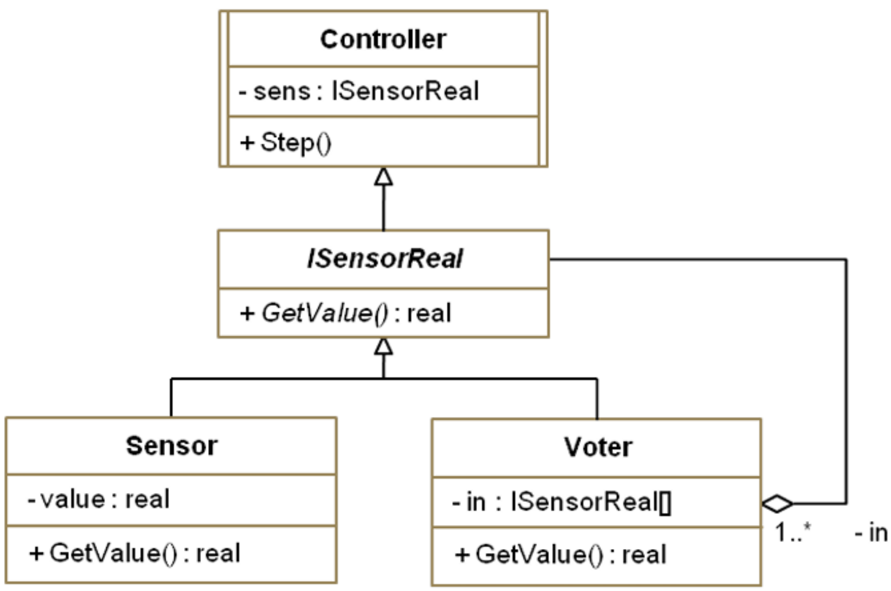
8

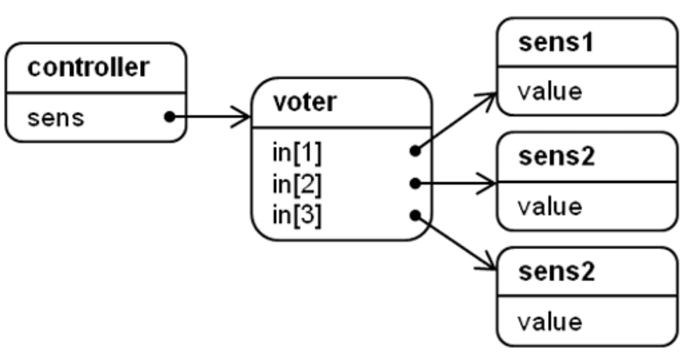
I used the following method to select my fault tolerant mechanisms:

(Error detection)--(establish recovery point)--(primary module)--(acceptance test)

Following design was used as reference to use SHARD-

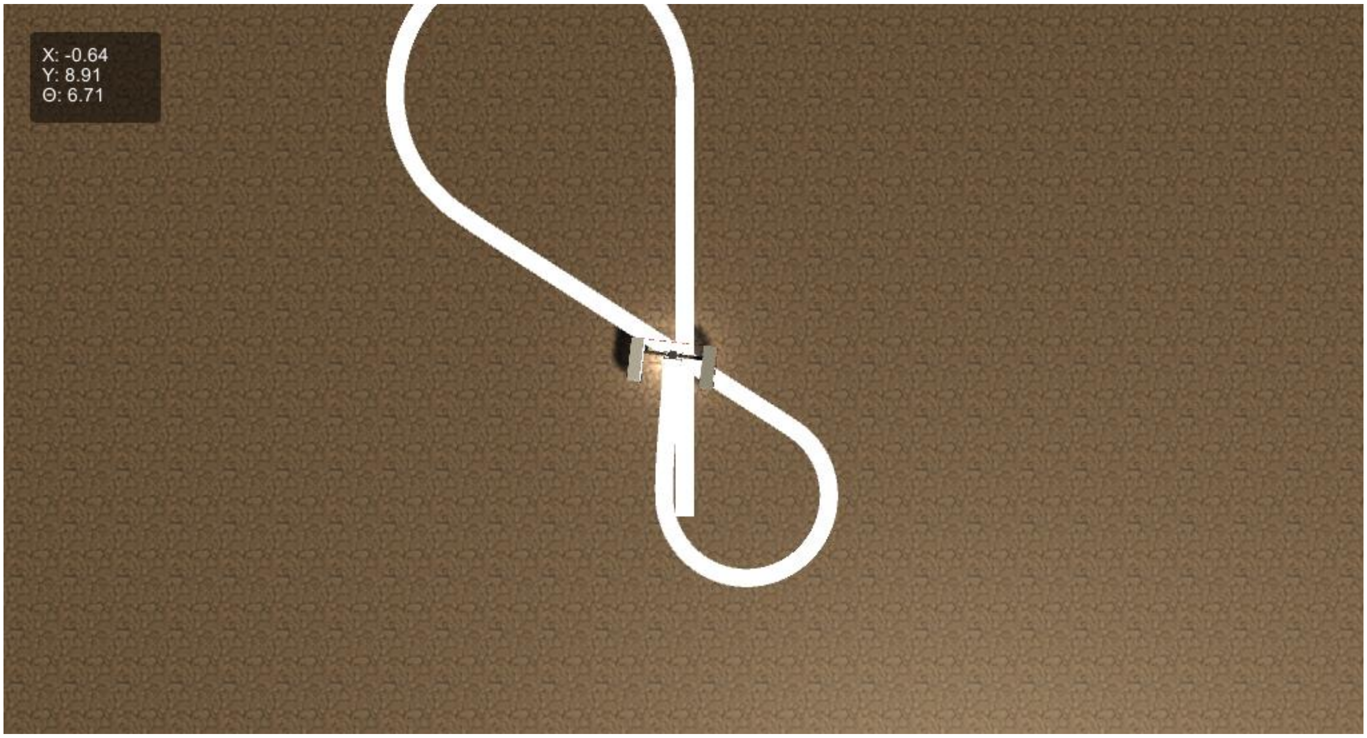
9 Co-modelling of Faults and Fault Tolerance Mechanisms



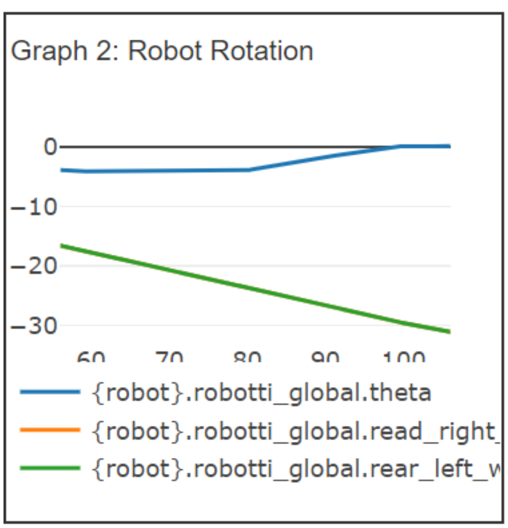


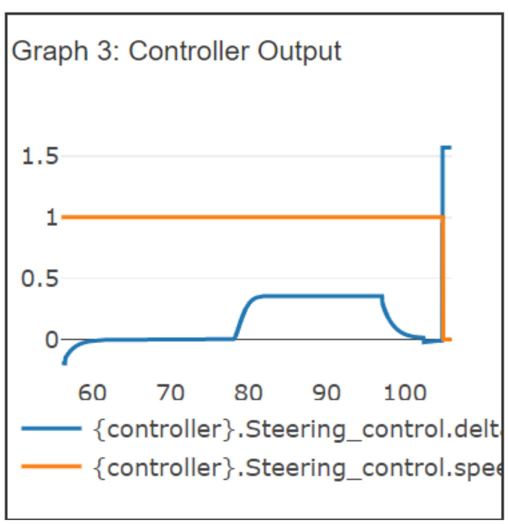
RESULTS

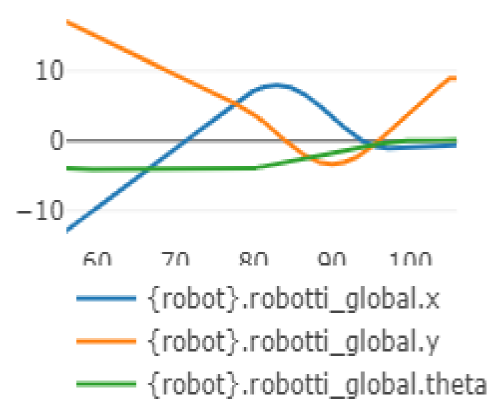
PRE-FAULT INJECTION BEHAVIOUR OF ROBOTTI:



PLOT READINGS:

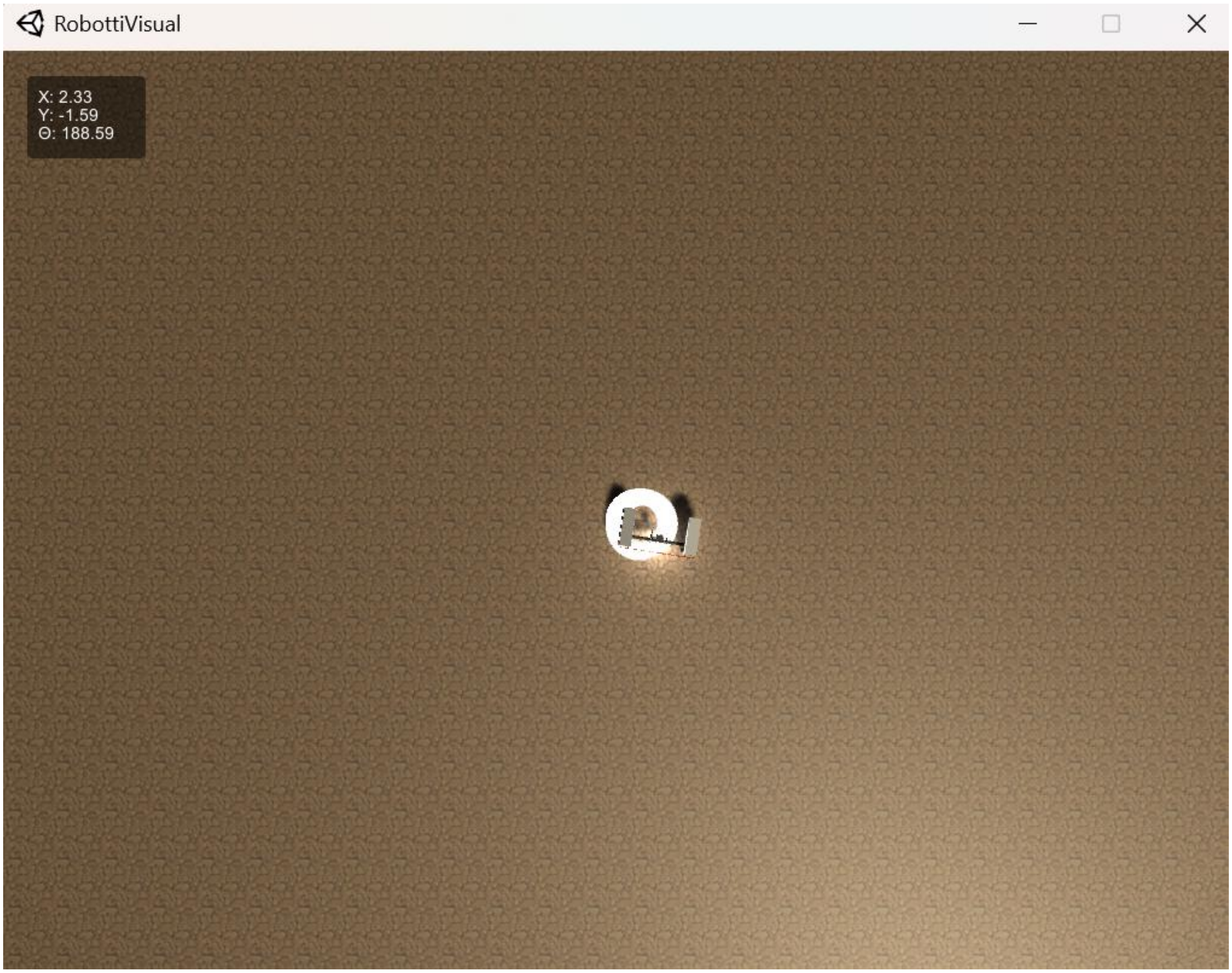




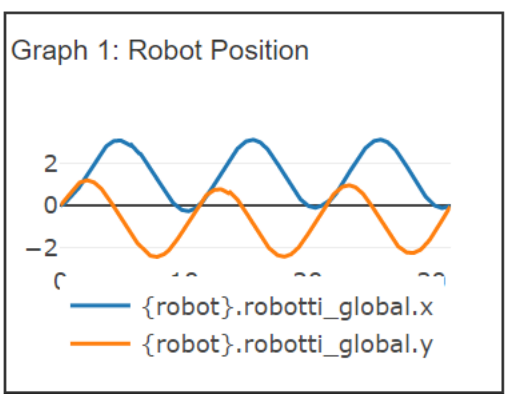


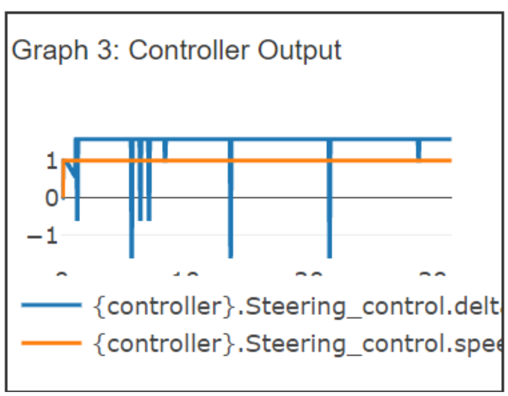
CASE 1:

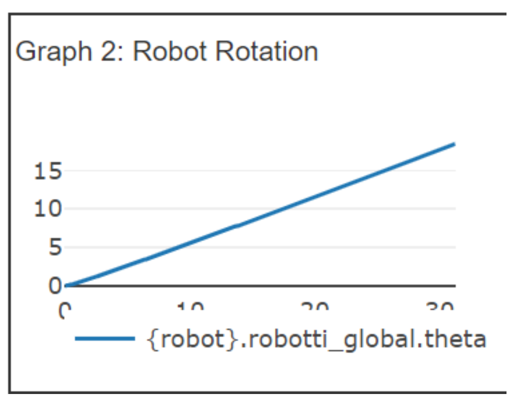
ROBOTTI BEHAVIOUR WITH FAULT INJECTION:



Plot readings:

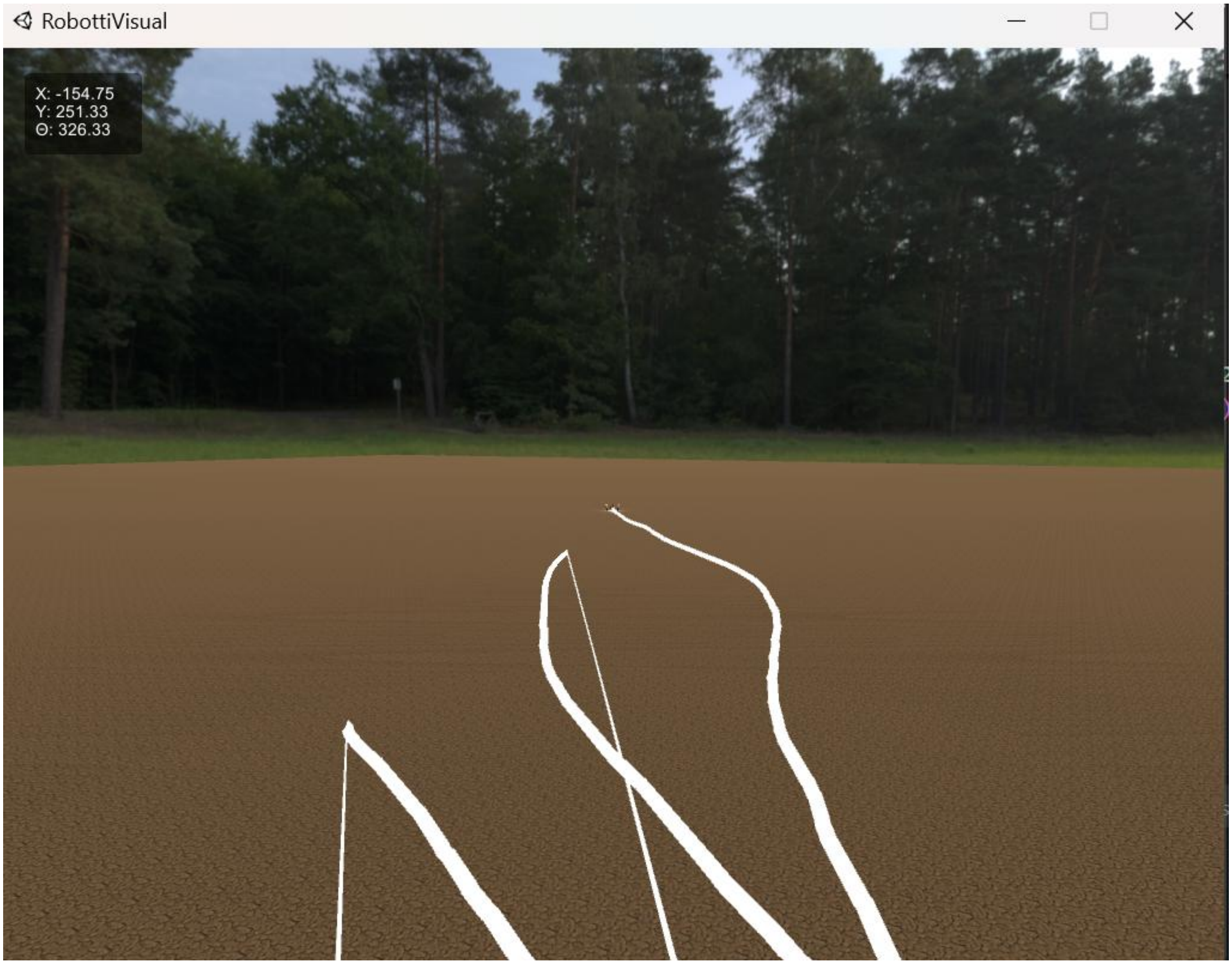




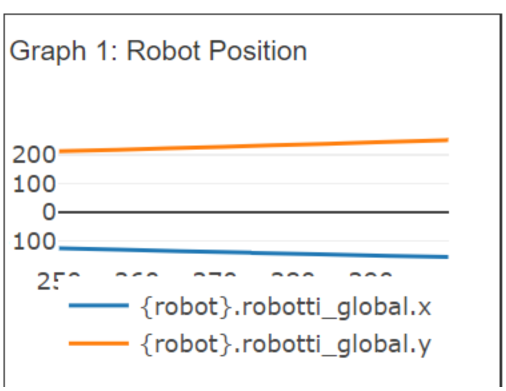


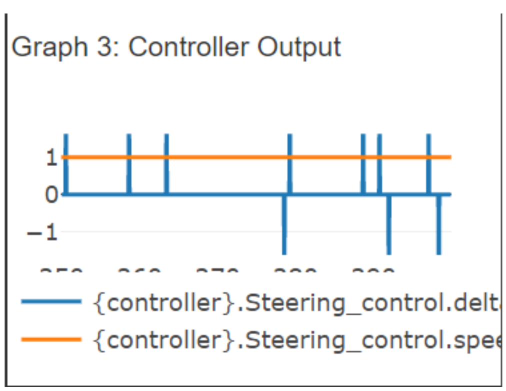
POST – FAULT TOLERANCE BEHAVIOUR OF ROBOTTI:

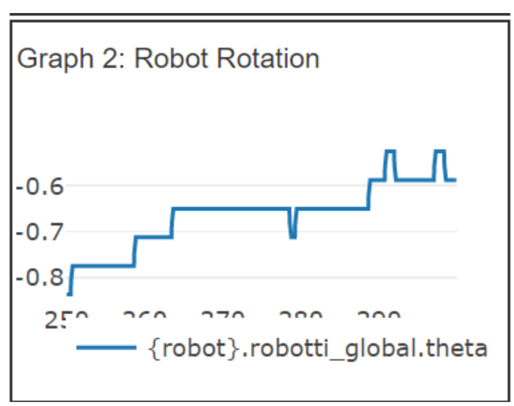
After 5 simultaneous simulations-



Plot Readings (1 simulation):

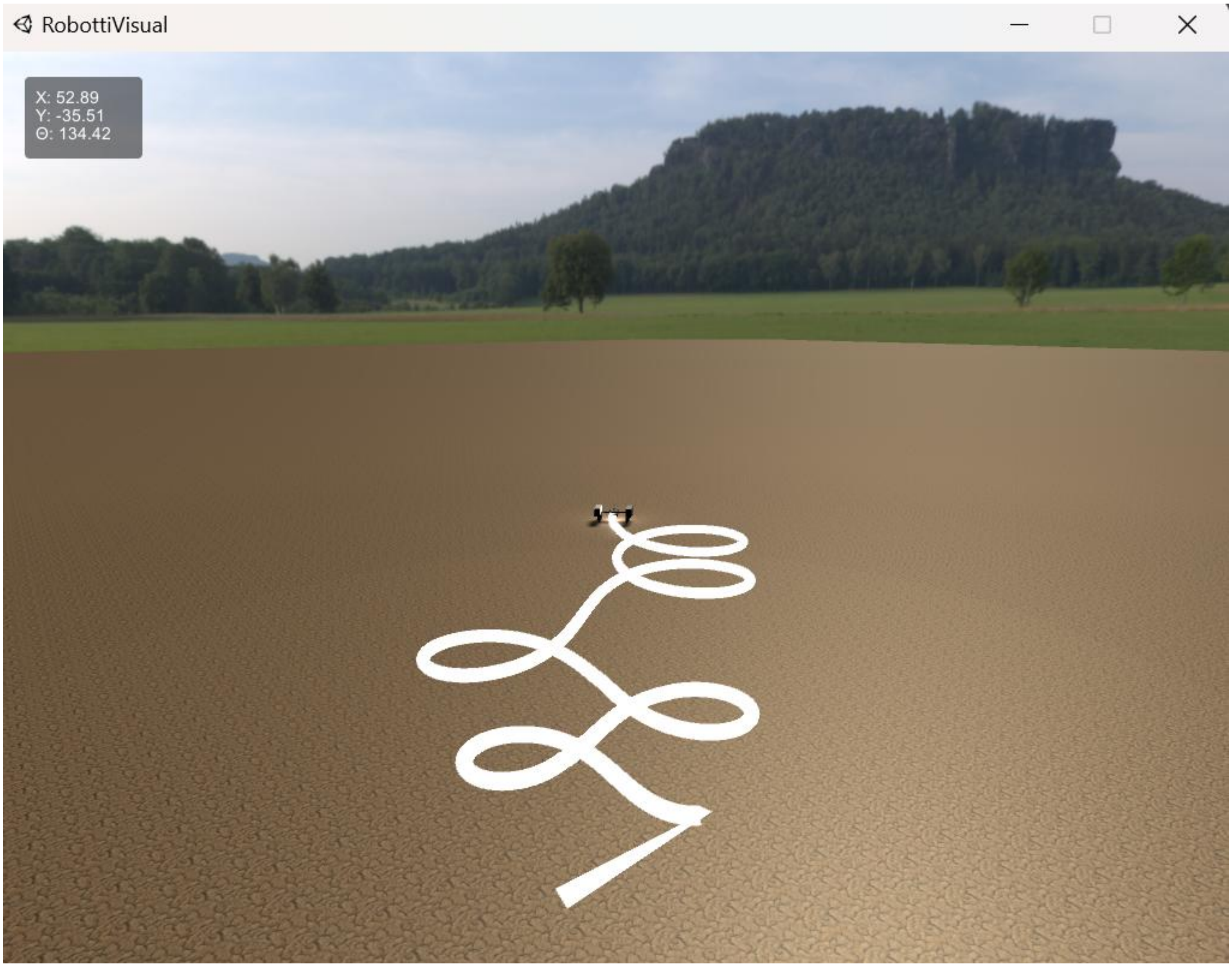




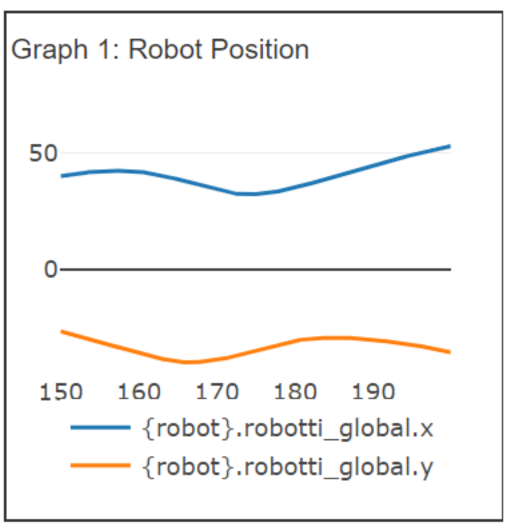


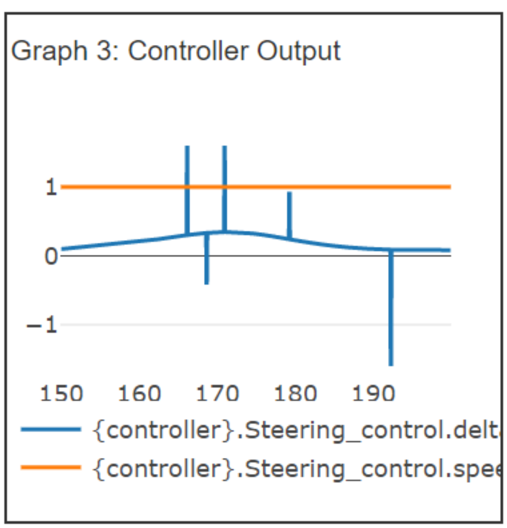
CASE 2:

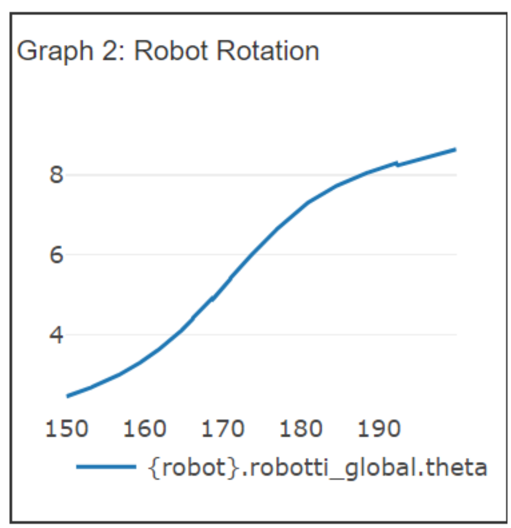
ROBOTTI3 BEHAVIOUR WITH FAULT INJECTION:



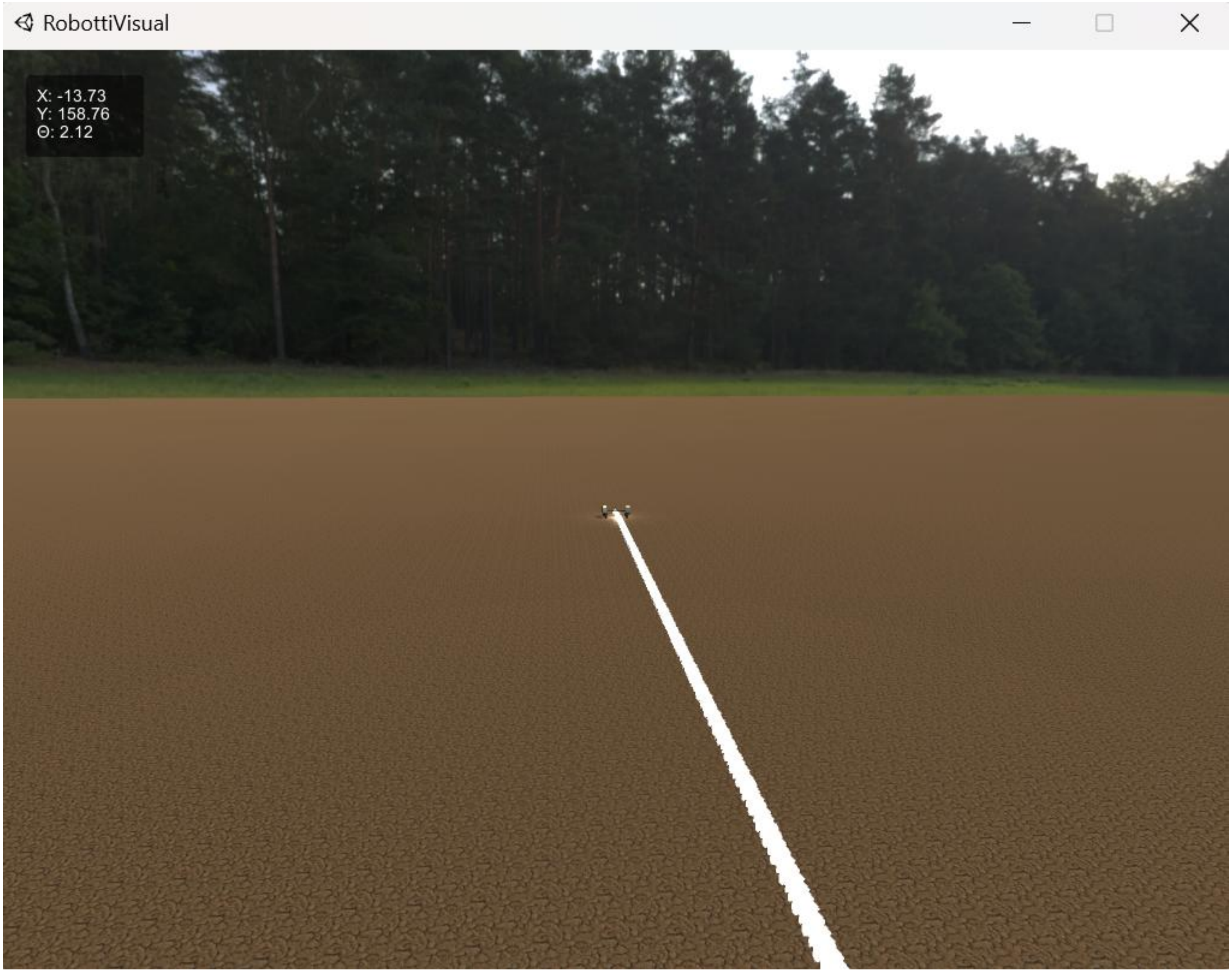
Plot Readings:



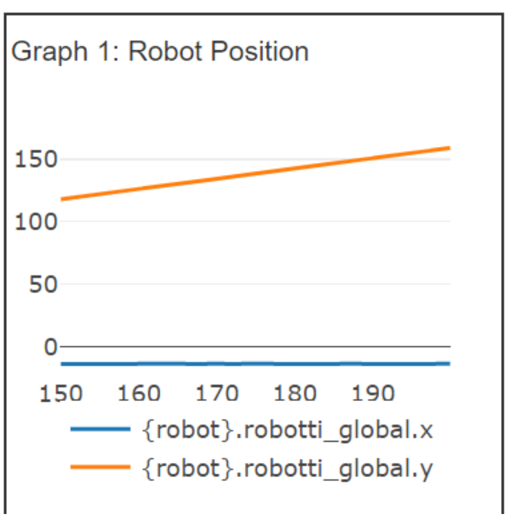


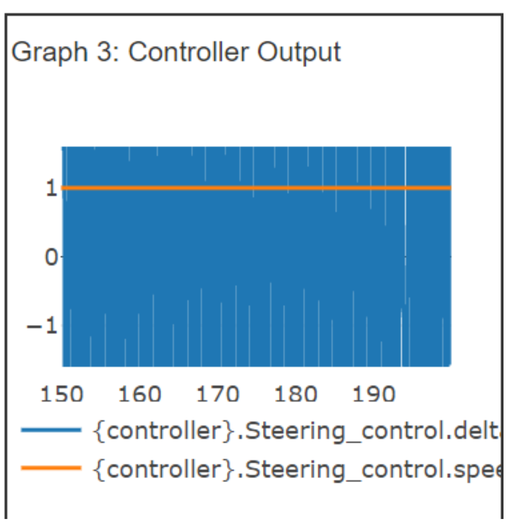


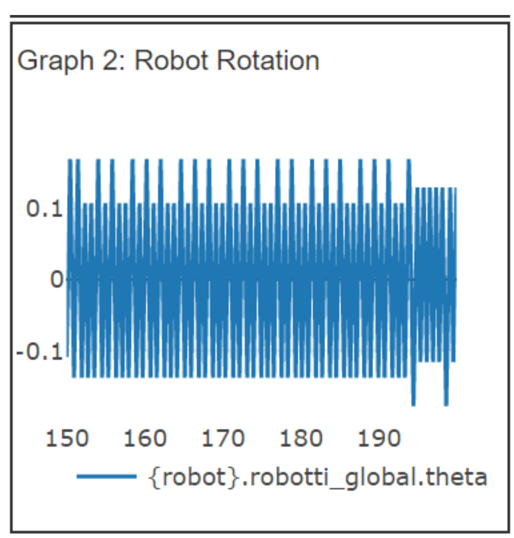
POST-FAULT TOLERANCE BEHAVIOUR OF ROBOTTI3:



Plot Readings:







OBSERVATIONS BASED ON RESULTS:

CASE1:

- Having faults in all GPS units caused the robotti to loop around, possible reason could be picking up route readings (small_curve).

-For Robotti1, I tried to integrate forward valid Safe state as a personal challenge.

-While I did not achieve the desired result from my attempt, a safe state was still established where the robotti kept moving straight.

- Another key difficulty was to mitigate the intensity of faults integrated, since all the sensors were simulating faults a backup state could not be generated.

CASE 2:

- Having only 1 GNSS indicating a fault, robotti3 was able to use the remaining ones to keep going, while figure_8 could not be established, it continued a loop in (small curve, straight_line) repeatedly.
- For Robotti3, I tried a Backward Error recovery mechanism to compare how that will work against the same issues.
- An interesting observation was made that the redundancy system did manage to keep the robotti3 on the correct route for an approx. 40%of the simulation until a half-8 figure was made and then probably due to logic error, strong threshold restrictions in steering.vdmrt or failure to call get_nextway_point(), the robotti3 proceeded to enter a safe state and kept moving straight after.
- A key benefit found in this model was the usage of other GNSS sensors, since only GNSS_H was injected with fault other sensors could be used for backup.

CONCLUSION

SUMMARY OF WORK DONE:

For this project, I developed two Robotti models- Robotti1 and Robotti3 both with the same types of erroneous intended fault injections but different fault-tolerant mechanisms. There were three degrees of comparisons made-

1) Comparing simulations of Robotti1 and Robotti3 before and after injecting faults and evaluating the system behaviour.
2) Comparing the different patterns of attempts made to create a redundant system and evaluating its effectiveness.
3) Comparing behaviours of Robotti1 and Robotti3 based on the coding methods used and evaluating the results.

- For future work, I would like to try out corrupting controller units and injecting packet losses, route corruptions or logic errors and moving over to creating hardware faults and backup units for it.
- I want to try manipulating the battery unit and manipulating its resistance meters and energy outputs.

In conclusion, This coursework taught me a key component of engineering which I can use for microchip manufacturing or building any other electronic component I want. My only drawback was finding the VDM language model very restrictive and pressing. In my view, it does require being modernized to match the current technological ease. Migrating from simple developer languages like Python and Java to learning the VDM manual was challenging but it did help me realize that as a software developer, we must continue adapting and learning rather than being limited and stagnant with what we already know.

REFERENCES

Foldager, F., 2015. *ROBOTTI.* Denmark.

John Fitzgerald, P. G. L. M. V., 2010. *Collaborative Design for Embedded Systems.* s.l.:s.n.