DEMO: A Model-Free Algorithm for Extremely Resilient Navigation

Christian J. Stromberger, Josefine B. Graebener, James F. Ragan, and Richard M. Murray

***Abstract*— Due to the increasing complexity of space missions and distance to exploration targets, future robotic systems used for space exploration call for more resilience and autonomy. Instead of minimizing the failure risk, we are focusing on missions that will inevitably encounter significant failures and are developing an algorithm that will autonomously reconfigure the system controller to continue to make progress towards the mission goal despite being in a reduced capacity state - we call this extreme resilience. In this paper, we develop a model-free framework to autonomously react to locomotion failures of robotic systems. This is done by the use of a neural network for path planning using the neuroevolution of augmenting topologies (NEAT) algorithm and a dynamic database of possible moves and their effect on the system's position and orientation. Two modes of failure detection and resolution are being introduced: (a) relative position failure detection, which is triggered by large, unexpected moves and results in a complete update of the database before a retraining of the neural network, and (b) absolute position failure detection, which triggers from large build-ups of position error from small failures and will induce a retraining of the neural network without an explicit database reset. We implement and validate this framework on a high-fidelity planetary rover simulation using Unreal Engine and on a hardware setup of a TurtleBot2 with a PhantomX Pincher robot arm.**

I. INTRODUCTION

Many interesting future space missions will require the ability to operate in the presence of failures [1, ch. 21, p. 13]. These might include failures of sensors, instruments, communication links, computing elements, actuators, and other components, without the possibility of human intervention, either due to communication delays or unknown or fast-changing environments. These missions call for increased resilience and more autonomy to achieve the mission objective.

For example, ocean worlds are communication-constrained, uncertain, and dynamic due to the presence of liquid water. Probes descending into the oceans of Europa or Enceladus would need to have the ability to learn from their interactions with the environment, react to imminent hazards, and make real-time decisions to respond to anomalies [2]. Similarly, ground missions to Venus are subjected to immense pressure and high temperatures on the planetary surface [3], and high altitude missions need to operate in the chemically reactive sulphuric-acid clouds [4]. After extended operation, spacecraft will eventually succumb to either of these conditions. For both these exploration targets, autonomous failure resolution is necessary due to constrained communications and time-critical decisions which prohibit successful human-in-the-loop control. Current approaches in spacecraft design address technical risk though prevention and redundancy [5], and environmental risk through large design margins [6]. Failure handling is heavily based on passive Fault Detection, Isolation and Recovery (FDIR) algorithms, and hard-coded reflexive behavior with deliberative decision making provided from the ground [7], [8]. We, however, are instead interested in sufficiently severe situations such that the mission can only continue by reconfiguring the system design on the fly and automatically, a capability we will call extreme resilience.

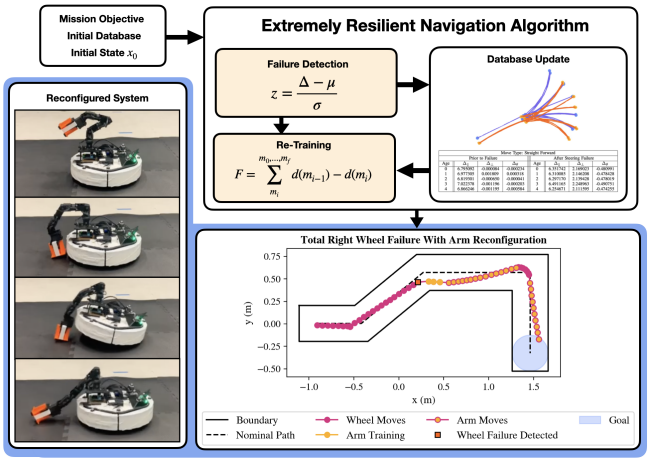

Fig. 1: The extremely resilient framework takes an objective, initial database, and initial sate as inputs. As failures are detected, it updates the database and retrains the path planning neural network as needed to reconfigure and reach the mission objective despite the system's reduced mobility. The output images in blue apply to the TurtleBot2 hardware demonstration.

Previous work has included on-orbit reconfiguration of multi-agent formations to optimally achieve changing mission objectives [9], [10] and avoid collisions [11]. Within a single agent, reconfiguration has been proposed for FPGAs to improve computational performance [12], or recover from permanent failures within a section of a FPGA [13]. For rover systems, previous work has examined reconfiguring the posture of the rover via redundant actuators to better traverse slippery sandy slopes [14] and other challenging terrain [15], as well as reconfiguring between wheeled and legged locomotion [16], [17]. This paper explores an

Research supported in part by the Foster and Coco Stanback Space Innovation Fund and a Caltech Summer Undergraduate Research Fellowship (SURF) sponsored by the Aerospace Corporation.

C.J. Stromberger is an Undergraduate Student at the California Institute of Technology, Pasadena, CA 91125, USA cjs@caltech.edu

J.B. Graebener and J.F. Ragan are with the Graduate Aerospace Laboratories, California Institute of Technology, Pasadena, CA 91125, USA jgraeben@caltech.edu, jragan@caltech.edu

R.M. Murray is with the Department of Control and Dynamical Systems, and the Department of Biology and Biological Engineering, California Institute of Technology, Pasadena, CA 91125, USA murray@cds.caltech.edu

approach to design reconfigurable systems that continue to provide valuable scientific data even as they undergo significant system failures. The main contributions are:

(i) Developing a model-free algorithm that autonomously adjusts for unexpected behavior of the system by retraining a neural network for path planning.
(ii) Introducing two modes of failure detection and resolution: relative position failure detection, which results in a retraining of the network after updating the database during a training cycle; and absolute position failure detection, which results in a retraining of the network from the current database.
(iii) Demonstrating and evaluating the developed approach in a high-fidelity simulation and with a hardware implementation on a low-cost TurtleBot2 mobile robotic platform with a PhantomX Pincher robotic arm.

II. RELATED WORK

There is a history of work on resilient systems. Bongard et al. have shown that via continuous self-modeling, four legged robots can learn new gaits after the loss of a limb [18]. We are interested in these large failures, but we want to create an approach that is not reliant on any particular model.

Quality Diversity algorithms allow for robots to learn new skills under normal operation, which can then be used to improve the robot's performance during tasks. Lim et al. have shown how a dynamics-aware quality diversity algorithm can perform the same as a standard quality diversity algorithm while only needing 20 times fewer samples [19]. Additionally, this algorithm can be used to recover from small failures such as the gap between simulation and reality. While these quality diversity algorithms may have a place in future work on the reconfiguration aspect of extreme resilience, we are focused on the larger, potentially mission critical, failures which may occur during operation. For this reason, we use a genetic algorithm (GA), which is better at responding to large unexpected changes.

Genetic algorithms have previously been applied to path planning and failure recovery, such as to find a route through a grid in [20]. We adopt a similar fitness function, which rewards shorter paths and penalizes paths which run into obstacles. GAs have also been used for guiding manipulators around environments with obstacles [21].

Additionally, GAs have been proven to be successful at reacting to the unknown. In [22], GAs are used to path plan through random, unknown, and time varying environments. Rather than focusing on unexpected surroundings, we are focusing on large and unexpected failures. Learning has also been used to deal with fault tolerance before. The authors of [23] use a no reset reinforcement learning approach for failure recovery. By sampling new gaits and leg movements while rewarding forward progress, a four legged robot is able to recover from failures and move to a goal, without human intervention or resetting.

When our robot moves, we store data about the move to be used for future path planning. Using a database to store robotic movements is also seen in [19], however, this was with the goal of correcting for failures arising from the simulation to reality gap. By limiting the size of the database, we are able to only store the most recent moves, which allows for quick reactions and recoveries from failures.

By combining and building on this previous work, we create a simple, model-free, algorithm for failure detection and recovery which can handle otherwise mission critical failures.

III. BACKGROUND

We achieve system reconfiguration by combining techniques from genetic algorithms with a model-free approach for navigation.

A. Evolutionary Algorithms

Evolutionary algorithms (EAs) are meta-heuristics that take inspiration from the biological process of evolution to generate solutions to complex optimization and search problems. A very popular EA is the genetic algorithm (GA) described in detail by Goldberg in [24]. GAs work by evolving an initially random candidate solution towards improved outcomes, which is represented by a fitness score. GAs use mutation and crossover as a selection strategy to combine genes from the individuals in a population to improve the fitness of the the offspring for the next generation. This selection strategy is biased towards higher fitness scores. GAs have been widely used and demonstrated to be efficient in many real-world applications, such as network load balancing [25], [26], [27], scheduling [28], [29], forecasting [30], [31], [32], encryption [33], and in game settings [34], [35].

1) NEAT: The genetic algorithm used in the paper is the neuroevolution of augmented topologies (NEAT) algorithm [36], which evolves the topology and the weights of Artificial Neural Networks (ANNs).

Three techniques are the basis of the NEAT approach: tracking genes using historical markings to allow crossover without the need for extensive topological analysis, speciation of the population to protect topological innovation, and incrementally growing from minimal structure — resulting in NEAT being able to simultaneously increase complexity and optimize solutions. Each of these techniques is crucial for NEAT's performance [36]. Historical markings are used to determine the origin of each gene and then assign an *innovation number* to this gene. This innovation number allows the algorithm to match up genes and perform crossover by randomly selecting one gene from each innovation number to create the offspring genome. This ensures that genomes stay compatible.

Speciation allows for the survival of species with smaller genomes, protecting them from elimination for a specific time to allow them to optimize their structure. The networks first compete within their species before they compete with the entire population. The process is possible due to the use of historical markings, which allows NEAT to compute the

degree of similarity between networks. Speciation ensures a diverse set of topologies of neural networks in a population.

While the initial population consists of very simple networks without any hidden nodes, over the generations the networks gradually expand and become more complex by adding or disabling nodes and altering their weights. As new structures evolve, only beneficial structures will survive. This process is called *complexification* and it ensures that networks do not become more complex than necessary [37]. NEAT has been applied to multiple domains such as game settings, including the game Go [38], the NERO video game [39], Atari [40], and Sonic the Hedgehog [41], as well as robotic applications such as the robust control of a quadrotor [42].

IV. ALGORITHM DESIGN

The goal of this algorithm is to move a robotic system from from a starting point to a goal point through a known safe corridor while recognizing and recovering from any failures in locomotion that occur along the way. Because the focus of this algorithm is on locomotion failure detection and recovery, we assume apriori knowledge of this safe corridor and assume no sensor or computational failures. This algorithm works by breaking up the robot's motions into discrete moves, each of which correspond to a set of driving inputs to be performed for a specific amount of time. Through experiments in the results section, we will demonstrate that due to the failure detection and replanning done by the algorithm, the robot will always reach its goal barring a complete failure in locomotion.

A. Neural Network Design

Because our algorithm uses NEAT to build the neural network used during path planning, only the number of input and output nodes need to be specified. Each network starts with five input nodes: the first two represent the robot's spacial x and y coordinates, the third represents the robot's orientation about the z axis, and the final two represent the x and y coordinates of the next point in the list of points that specify the safety corridor. We call this next point in the safety corridor a "subgoal point," because it may or may not be the final point the robot is trying to reach. Each of the output nodes corresponds to a move type that the robot can perform. The NEAT algorithm evolves all hidden nodes, connections, and weights.

Structuring our network in this way allows us to give the network the robot's position and the goal position and receive out the next move the robot should preform (the move which corresponds with the most activated output node). In order to complete the whole path through the safety corridor, a new network is created for every subgoal point of the journey until the goal has been reached.

B. Neural Network Fitness

Each time we need to generate a network for the next subgoal point, or plan a new path after a failure, NEAT is used to generate a population of potential networks, which must each be tested and assigned a fitness score to inform NEAT's creation of the next generation of networks. To test each of these networks, we input the starting robot position and subgoal position into the network. The network will select a move via output node activation, which will then be simulated in the environment, assessed, and the resulting simulated position is fed back into the network to choose the next move.

This process continues until either the network succeeds in providing a viable path for the robot that will result in it reaching the subgoal, or the network fails by either suggesting a move that brings the robot out of the safety corridor or suggesting a path that contains more than a predetermined number of moves which bring the robot away from the goal. These fail conditions exist both to minimize risk to the robot and encourage efficient paths. When the test is over, the following equation is used to assign the network a fitness score.

$$F = \sum_{m_i}^{m_2,\ldots,m_f} d(m_{i-1}) - d(m_i) \tag{1}$$

Algorithm 1: Extremely Resilient Navigation

```
def resilient_travel(p_i, p_g, pretrained_database):
    moves ← replan(p_i, p_g);
    resilient_move(moves, pretrained_database, p_g);
def resilient_move(moves, database, p_g):
    for m = m_0, ..., m_f ∈ moves do
        p_i ← get_current_position();
        drive(m);
        p_f ← get_current_position();
        Δ ← get_position_change(p_i, p_f);
        if reached_goal(p_f, p_g) then
            return;
        if relative_position_failure(Δ, database) then
            new_database ← retrain();
            new_moves ← replan(p_i, p_g);
            resilient_move(new_moves, new_database);
            return;
        database ← add_to_database(database, m, Δ);
        if absolute_position_failure(p_f, moves) then
            new_moves ← replan(p_i, p_g);
            resilient_move(new_moves, database);
            return;
        if out_of_safe_zone(p_f) then
            inverse_drive(m);
            new_moves ← replan(p_i, p_g);
            resilient_move(new_moves, database);
            return;
    if not reached_goal(p_f) then
        new_moves ← replan(p_i, p_g);
        resilient_move(new_moves, database);
    return;
```

While meeting one of the failure conditions stops the network's test, its fitness score will remain the same so it can still be used for creating the next generation.

For each move m_i proposed by the network, the distance function $d(m_i)$ returns the distance remaining to the subgoal point after performing that move. Thus, (1) rewards the network each time it suggests a move that brings it closer to the subgoal and punishes moves that bring it further away from the subgoal. While this fitness function can tailored to better apply to specific systems, this model-free function has proven strong enough to work in both the simulation and hardware demonstrations.

C. Training the Robot

For the algorithm to accurately simulate and score neural networks as they are provided by NEAT, having an idea of how each move will effect the robot's position is crucial. To satisfy this need, the algorithm builds up a database of what moves the robot is capable of and how each of these moves effects the robot's position and orientation during a process called *training*. During training — which happens prior to the robot's deployment in the field — the robot completes each move multiple times. By recording initial and final positions and orientations as each move is performed, the changes in position and orientation can be calculated and inputted into the database. This method of training is entirely model-free, and the moves taught to the robot are user defined.

To calculate changes in position consistently, the database keeps track of the change in position parallel and perpendicular to the direction the robot is facing prior to the move, and how much the robot rotates during the move, as given by:

$$\Delta_{\parallel} = (x_f - x_i) \cdot \cos\theta_i + (y_f - y_i) \cdot \sin\theta_i \tag{2}$$

$$\Delta_{\perp} = -(x_f - x_i) \cdot \sin\theta_i + (y_f - y_i) \cdot \cos\theta_i \tag{3}$$

$$\Delta_{\theta} = \theta_f - \theta_i \tag{4}$$

where x_i, x_f refer to the initial and final positions respectively, and similarly for y_i, y_f, and θ_i, θ_f. Using (2), (3), and (4), the initial and final coordinates and orientation can be translated from the global coordinate frame to the robot's coordinate frame before the move occurred. This common frame allows each move to be compared to the other moves in the database, thus allowing the algorithm to identify when a move gives an unexpected result.

When evaluating neural networks, the algorithm uses the average $\Delta_{\parallel}$, $\Delta_{\perp}$, and Δ_{θ} of the entries for a specific move to get an accurate estimate of how that move will effect the robot's pose.

While this database is created during the initial training, it continues to be updated as the robot performs moves in the field. The database is designed to only hold a predetermined number of entries per move, so when a move is performed in the field, the change in position from this move is added to the database and the oldest recorded position change is removed. Thus, if any slight variations in the environment were to arise, or a small failure were to occur that only slightly altered the robot's movements, the database would soon fill up with accurate data while removing the outdated entries.

In the event of a large and sudden failure, the database being used becomes obsolete, since the effect each move has on the robot's pose will change dramatically. When this type of failure is detected, the old database is cleared and a round of training is completed in the field. This *retraining* is different from the initial training in two ways. First, every time a move is performed and recorded, the reverse of that move (if the move is reversible) is performed so the robot remains in the same spot after the retraining is complete. Second, the retraining may perform each move less times than the initial training in an effort to reduce time spent traveling or to reduce the risk that comes from moving with a reconfigured part, as seen when we retrain an arm for locomotion during our hardware demonstration.

D. Failure Detection

There are two methods of failure detection used by the algorithm. The first, called relative position failure detection, attempts to recognize large failures as soon as they occur by considering the relative position changes of the robot after each move and determines whether the move result is unexpected (meaning a failure has likely occurred) or not. The second type of failure detection is absolute failure detection, which looks at the absolute position of the robot in the global coordinate frame to determine whether a failure has occurred. This method recognizes small failures that are not big enough to trigger the relative position failure detection but still cause the robot to stray significantly from its planned path over the course of many moves.

1) Relative Position Failure Detection: After the position and orientation changes have been calculated with (2), (3), and (4), each of $\Delta_{\parallel}$, $\Delta_{\perp}$, and Δ_{θ} are compared to the other entries for that move in the database, and the move is assigned a set of z-scores $(z_{\parallel}, z_{\perp}, z_{\theta})$ that correspond to each of the changes in position or orientation, calculated as:

$$z = \frac{\Delta - \mu}{\sigma} \tag{5}$$

The z-score, z, simply provides the number of standard deviations, σ, from the mean, μ, a particular change, Δ, lies from the reference population held in the database. While the specific threshold of z-score that results from a failure can be tuned depending on the size of the database and the consistency of a system, this method of failure detection has proven effective in both the hardware and simulation demonstrations [43].

Because the failures detected by this method greatly effect the robot's movements, when this method detects a failure the database is retrained and a new path is planned.

2) Absolute Position Failure Detection: If a failure is too subtle to result in a large z-score, it will not trigger the relative position failure detection. Absolute position failure detection combats this by checking what the algorithm

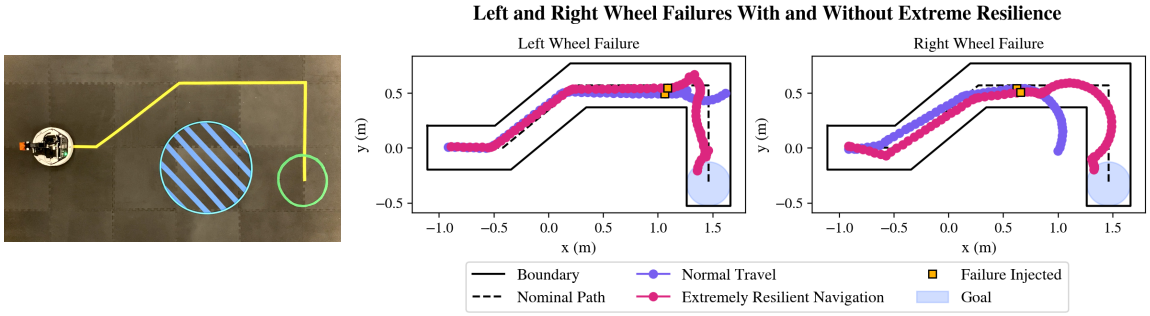

Fig. 2: The TurtleBot must navigate around the blue striped hazard by following the yellow nominal path to reach the green goal circle. On the right, the TurtleBot's travel during left and right wheel failures can be seen. These failures were injected suddenly at similar locations and brought the respective wheel down to 50% of its nominal speed. Retraining has been omitted for clarity.

expects the robot's global position and orientation to be after each move has been completed and comparing it to the robot's actual pose. Because it takes several moves for small failures to build up and become noticeable, by the time this failure detection is triggered, the database already been repopulated by accurate representations of the effects of each move. Thus, the robot skips retraining and only replans. Any residual database entries from before the subtle failure will be discarded as the robot continues to move and new entries are added to the database.

E. Reconfiguration

In some instances, a robot loses a critical amount of mobility and becomes stuck. In the extremely resilient system we envision, the robot will *reconfigure* itself by taking something not initially designed to aid in the robot's mobility and repurposing it. This is shown in the TurtleBot2 hardware demonstration where, upon losing total function in one or both of its wheels, the algorithm will use the robotic arm initially designed for manipulating the environment around the robot and reconfigure it into a pushing device. While doing so risks damage to the arm, the alternative is total mission failure.

In an event where reconfiguration is required, new move types will be added to the database and an in-field training will occur to populate the database entries. Additionally, new output nodes will be added to the neural network structure, so these new moves can be included in future navigation.

Reconfiguration is at the core of extreme resilience, as it allows a robot to continue a mission when an otherwise mission-critical failure occurs.

V. EXPERIMENTAL SETUP

To demonstrate the effectiveness of the extremely resilient navigation algorithm, two experimental setups are considered. The first is a hardware demonstration of a mobile base with a robotic arm on top, and the second is a software simulation of a lunar rover. These demonstrations also show the model-free nature of this method, since these robots differ significantly in many ways, including wheel configuration, controller inputs, and size. Despite these differences, both have the goal of reaching a specific location through a safe corridor and both experience failures along the way.

A. TurtleBot Hardware Demonstration

This hardware demonstration uses a two-wheeled TurtleBot2 Kobuki mobile base [1] with a PhantomX Pincher Robot Arm [2] attached to the top. The laptop that acts as the TurtleBot's processor has been replaced by an NVIDIA Jetson Nano to reduce the overall system's weight and increase mobility. The Jetson Nano is powered by an onboard rechargeable 5V power supply, making the whole system untethered and therefore entirely mobile. Sensing failures are outside the scope of this work, so an OptiTrack motion capture system is used to measure the TurtleBot's position in real-time.

During training, the TurtleBot is taught to move forward, pivot both clockwise and counterclockwise around each wheel, and spin both clockwise and counterclockwise in place.

At any point during the test, the operator can inject a failure in one or both of the wheels of the robotic base which will limit their speed. If the algorithm determines that at least one wheel has completely failed, it will begin to retrain the robot's original moves in addition to reconfiguring the arm and training three arm moves: a pivot clockwise, a pivot counterclockwise, and a push. Because complete reconfiguration is outside of the scope of this paper, these three arm movements have been preprogrammed, but not trained for in the initial training. Once the TurtleBot reconfigures itself and thus regains the ability to move, it plans a new path to its goal.

B. Moon Rover Simulation

To demonstrate how this approach applies to planetary exploration missions, a simulation in a realistic environment is required. The setup is built using AirSim, a high-fidelity simulator developed by Microsoft [44]. AirSim is based on Unreal Engine[3], a real-time game engine that is free for research and non-commercial use and has become widely used for simulation and visualization purposes. The environment chosen for the demonstration is a simulated a Moon

[1]https://www.turtlebot.com/turtlebot2/
[2]https://www.trossenrobotics.com/p/PhantomX-Pincher-Robot-Arm.aspx
[3]https://www.unrealengine.com/

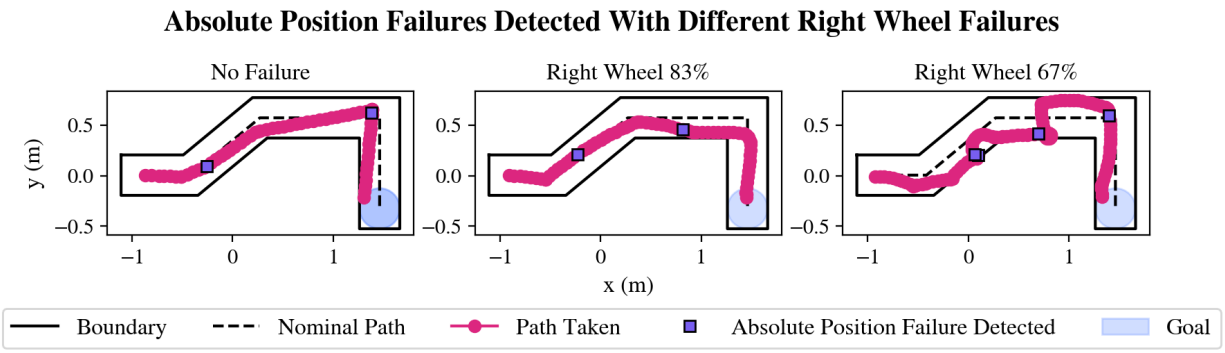

Fig. 3: The TurtleBot's navigation under only absolute position failure detection. The database used for each of these runs holds five entries per move type. There are two failures detected in both the no injected failure and 83% failure tests and four in the 67% failure test.

landscape [4] on which the default AirSim car, acting as a rover, needs to reach predetermined target locations while being subjected to failures.

The rover is taught to move straight forward, straight backwards, and how to make hard and soft right and left turns. Each move is executed for three seconds after which the brakes are applied until the rover comes to a stop. During the mission, a steering drift may be injected, causing the rover to always steer slightly more towards a particular direction than expected. All the information required to use our algorithm, such as rover position and orientation, is gathered from the AirSim API.

VI. RESULTS

Using the TurtleBot hardware demonstration and the Moon rover simulation, in this section we show the trajectories taken by the robots under different failure modes and conditions. Extremely resilient navigation is compared to a system which is trained before deployment and uses the NEAT algorithm to plan a path, but does not update the database, replan, retrain, or detect failures.

A. TurtleBot Hardware Demonstration

The main result of extremely resilient navigation is the ability for a system to recover from a locomotion failure and still reach a goal. The method's effectiveness is shown in Fig. 2, where the TurtleBot is subjected to a right or left single wheel partial failure but still reaches its goal. Because this failure removes the ability for the TurtleBot to move directly forward, the NEAT algorithm finds patterns of moves which, when done together, result in an overall forward motion. In the example of the left wheel failure, the last leg of the journey consists of "forward" motions (which now drift the robot towards the left due to the failure) followed by clockwise pivots, which reset the TurtleBot's orientation so it can perform more forward moves. Similar behavior is seen in the case of the right wheel failure. After the failure is recognized, the robot pivots left, which allows it to curve to the next subgoal near $(1.5, 0.5)$ and continue curving until it performs one more orientation reset just before reaching the goal.

Fig. 1 shows how reconfiguration in extremely resilient navigation can allow a system to recover from a critical mobility failure. Around halfway through the TurtleBot's journey, it experiences a total right wheel failure, restricting its motion to pivots around the broken right wheel. However, the algorithm reconfigures the arm and trains the robot to use it to push itself forward and turn itself clockwise and counterclockwise. Because the arm movements have not been trained before, the TurtleBot is unable to train them in place, and the resulting movements from their training can be seen in yellow. The algorithm plans a path that still takes advantage of the working wheel to make a turn around the final corner, showcasing how reconfiguration allows all aspects of the robot to work together.

To demonstrate the effectiveness of a dynamic database and absolute position failure detection, we consider an experiment in which the relative position detection is disabled and the TurtleBot begins with a database of moves from a training with no failures. Before the tests are run, the right wheel speed is reduced to either 83% or 67% its nominal value, or left unchanged as a control. The results are sown in Fig. 3. Despite there being no injected failures in the control trail, absolute failure detection is still triggered twice. This is due to the natural inconsistencies between moves that arise due to noise during the TurtleBot's normal operation, and is one of the things absolute failure protection helps protect against. Under the smaller 83% right wheel failure, the absolute position failure detection is also triggered twice, which demonstrates the importance of a dynamic database. By the time the absolute failure detection is triggered, the database has already been updated enough to not require retraining, and thus only replanning is enough. The 67% failure plot shows how the TurtleBot reacts to a failure that usually would be caught by relative position failure detection when only absolute failure detection is present. While absolute failure detection is triggered in this trail more than the other two trails, the robot is still able to recover and complete the entire last leg of the path without any replanning.

[4] https://www.unrealengine.com/marketplace/en-US/product/moon-landscape-01

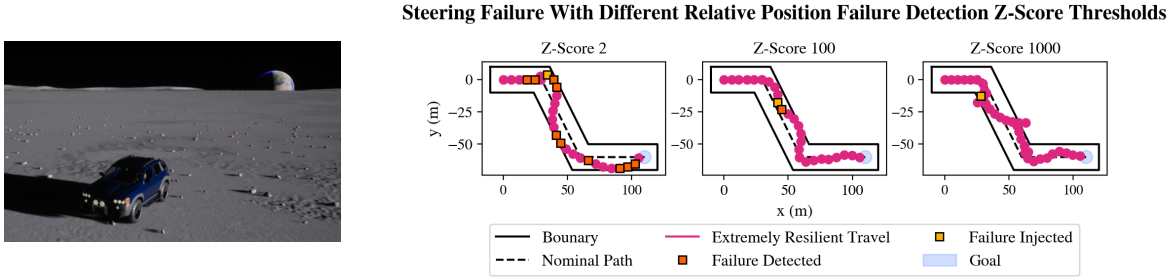

Fig. 4: The rover on the moon is pictured on the left. On the right, the rover's failure detection is shown when reacting to a rightward steering failure of 10% of the maximum steering value at different z-score thresholds. Too low of a threshold results in false positives, while too high of a z-score threshold results in missing the failure completely. In the 1000 z-score example, the rover leaves the safety threshold twice. Retraining is omitted for clarity.

B. Moon Rover Simulation

Because relative failure detection is driven by calculating the z-scores of each move compared to the moves in the database, it is important to tune the z-score to the particular system. This is demonstrated by the experiment summarized in Fig. 4, where the rover experiences a steering drift that is 10% of the maximum steering value more to the right. When too small of a z-score is chosen, relative position failure detection is triggered too often, resulting in many unnecessary retraining rounds that waste time and, in the case of a real life rover, energy.

On the other hand, picking too large of a z-score can prove detrimental. In the plot on the right of Fig. 4, the rover drives out of the safety corridor twice due to its inability to accurately path plan with its outdated database. By picking a proper z-score, both of these extremes can be avoided.

VII. CONCLUSIONS & FUTURE WORK

In this paper we have outlined a model-free method for navigating through a safe corridor in the face of failures. We introduce two failure detection methods, relative position failure and absolute position failure, which are designed to catch big and small failures respectively. The use of a dynamic database to capture moves and adapt to small changes in the environment as they arise makes both of these failure detection methods possible, and allows for the NEAT algorithm to evolve neural networks for path planning. To demonstrate the effectiveness and the model-free aspect of this method, it has been applied to both a robot with a two-wheeled mobile base with a robotic arm and a rover simulation with four wheels and a different set of control inputs.

In the future, the method outlined in this paper could be applied to other aspects of the robot beyond recovering from locomotion failures, such as sensor failures or power system failures. Automatic reconfiguration is the key next step in designing extremely resilient systems. When implemented, the robot would be able to reconfigure itself without the need for pre-designed fall back moves by learning the database on-the-fly. In such a system, the need to preprogram the arm movements into the TurtleBot would disappear and the robot would be able to figure out how to use the arm to move itself on its own.

REFERENCES

[1] National Academies of Sciences, Engineering, and Medicine, "Origins, worlds, and life: A decadal strategy for planetary science and astrobiology 2023-2032," 2022.

[2] B. Sherwood, "Strategic map for exploring the ocean-world enceladus," *Acta Astronautica*, vol. 126, pp. 52–58, 2016.

[3] J. Sauder, E. Hilgemann, M. Johnson, A. Parness, B. Bienstock, J. Hall, J. Kawata, and K. Stack, "Automation rover for extreme environments: Nasa innovative advanced concepts (niac) phase 1 final report," Tech. Rep., 2017.

[4] J. Hall, D. Fairbrother, T. Frederickson, V. Kerzhanovich, M. Said, C. Sandy, J. Ware, C. Willey, and A. Yavrouian, "Prototype design and testing of a venus long duration, high altitude balloon," *Advances in Space Research*, vol. 42, no. 10, pp. 1648–1655, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0273117707002360

[5] M. E. Paté-Cornell, R. L. Dillon, and S. D. Guikema, "On the limitations of redundancies in the improvement of system reliability," *Risk Analysis: An International Journal*, vol. 24, no. 6, pp. 1423–1436, 2004.

[6] C. Ercol, E. Abel, G. A. Holtzman, and E. Wallis, "Thermal design verification testing of the solar array cooling system for parker solar probe." 48th International Conference on Environmental Systems, 2018.

[7] M. Tipaldi and B. Bruenjes, "Survey on fault detection, isolation, and recovery strategies in the space domain," *Journal of Aerospace Information Systems*, vol. 12, no. 2, pp. 235–256, 2015.

[8] D. Atkinson, M. James, D. Lawson, R. Martin, and H. Porta, "Automated spacecraft monitoring," in *1990 IEEE International Conference on Systems, Man, and Cybernetics Conference Proceedings*, 1990, pp. 756–761.

[9] G. T. Huntington and A. V. Rao, "Optimal reconfiguration of spacecraft formations using the gauss pseudospectral method," *Journal of Guidance, Control, and Dynamics*, vol. 31, no. 3, pp. 689–698, 2008.

[10] O. Junge and S. Ober-Blobaum, "Optimal reconfiguration of formation flying satellites," in *Proceedings of the 44th IEEE Conference on Decision and Control*. IEEE, 2005, pp. 66–71.

[11] Q. Hu, H. Dong, Y. Zhang, and G. Ma, "Tracking control of spacecraft formation flying with collision avoidance," *Aerospace Science and Technology*, vol. 42, pp. 353–364, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1270963815000334

[12] B. Osterloh, H. Michalik, S. A. Habinc, and B. Fiethe, "Dynamic partial reconfiguration in space applications," in *2009 NASA/ESA Conference on Adaptive Hardware and Systems*, 2009, pp. 336–343.

[13] J. Zhang, Y. Guan, and C. Mao, "Optimal partial reconfiguration for permanent fault recovery on sram-based fpgas in space mission," *Advances in Mechanical Engineering*, vol. 5, p. 783673, 2013.

[14] H. Inotsume, M. Sutoh, K. Nagaoka, K. Nagatani, and K. Yoshida, "Modeling, analysis, and control of an actively reconfigurable planetary rover for traversing slopes covered with loose soil," *Journal of Field Robotics*, vol. 30, no. 6, pp. 875–896, 2013. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21479

[15] P. S. Schenker, P. Pirjanian, J. Balaram, K. S. Ali, A. Trebi-Ollennu, T. L. Huntsberger, H. Aghazarian, B. A. Kennedy, E. T. Baumgartner, K. D. Iagnemma, A. Rzepniewski, S. Dubowsky, P. C. Leger, D. Apostolopoulos, and G. T. McKee, "Reconfigurable robots for all-terrain exploration," in *Sensor Fusion and Decentralized Control in Robotic Systems III*, G. T. McKee and P. S. Schenker, Eds., vol. 4196, International Society for Optics and Photonics. SPIE, 2000, pp. 454 – 468. [Online]. Available: https://doi.org/10.1117/12.403744

[16] W. Reid, R. Fitch, A. H. Göktoğan, and S. Sukkarieh, "Sampling-based hierarchical motion planning for a reconfigurable wheel-on-leg planetary analogue exploration rover," *Journal of Field Robotics*, vol. 37, no. 5, pp. 786–811, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21894

[17] E. Rohmer, G. Reina, and K. Yoshida, "Dynamic simulation-based action planner for a reconfigurable hybrid leg–wheel planetary exploration rover," *Advanced Robotics*, vol. 24, no. 8-9, pp. 1219–1238, 2010. [Online]. Available: https://doi.org/10.1163/016918610X501499

[18] J. Bongard, V. Zykov, and H. Lipson, "Resilient machines through continuous self-modeling," *Science*, vol. 314, no. 5802, pp. 1118–1121, 2006. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.1133687

[19] B. Lim, L. Grillotti, L. Bernasconi, and A. Cully, "Dynamics-aware quality-diversity for efficient learning of skill repertoires," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE Press, 2022, p. 5360–5366. [Online]. Available: https://doi.org/10.1109/ICRA46639.2022.9811559

[20] T. W. Manikas, K. Ashenayi, and R. L. Wainwright, "Genetic algorithms for autonomous robot navigation," *IEEE Instrumentation & Measurement Magazine*, vol. 10, no. 6, pp. 26–31, 2007.

[21] L. Tian and C. Collins, "An effective robot trajectory planning method using a genetic algorithm," *Mechatronics*, vol. 14, no. 5, pp. 455–470, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957415803000990

[22] K. Sugihara and J. Smith, "Genetic algorithms for adaptive motion planning of an autonomous mobile robot," in *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'*, 1997, pp. 138–143.

[23] K. Chatzilygeroudis, V. Vassiliades, and J.-B. Mouret, "Reset-free trial-and-error learning for robot damage recovery," *Robotics and Autonomous Systems*, vol. 100, pp. 236–250, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889017302440

[24] D. E. Goldberg, "Genetic algorithms in search, optimization and machine learning," *Reading: Addison-Wesley*, 1989.

[25] H. Cheng, S. Yang, and J. Cao, "Dynamic genetic algorithms for the dynamic load balanced clustering problem in mobile ad hoc networks," *Expert Systems with Applications*, vol. 40, no. 4, pp. 1381–1392, 2013.

[26] T. Scully and K. N. Brown, "Wireless lan load-balancing with genetic algorithms," in *International Conference on Innovative Techniques and Applications of Artificial Intelligence*. Springer, 2008, pp. 3–16.

[27] J. He, S. Ji, M. Yan, Y. Pan, and Y. Li, "Load–balanced cds construction in wireless sensor networks via genetic algorithm," *International Journal of Sensor Networks*, vol. 11, no. 3, pp. 166–178, 2012.

[28] R. Guido and D. Conforti, "A hybrid genetic approach for solving an integrated multi-objective operating room planning and scheduling problem," *Computers & Operations Research*, vol. 87, pp. 270–282, 2017.

[29] R. Zhang, S. Ong, and A. Y. Nee, "A simulation-based genetic algorithm approach for remanufacturing process planning and scheduling," *Applied Soft Computing*, vol. 37, pp. 521–532, 2015.

[30] G. Sermpinis, C. Stasinakis, K. Theofilatos, and A. Karathanasopoulos, "Modeling, forecasting and trading the eur exchange rates with hybrid rolling genetic algorithms—support vector regression forecast combinations," *European Journal of Operational Research*, vol. 247, no. 3, pp. 831–846, 2015.

[31] S. Mahfoud and G. Mani, "Financial forecasting using genetic algorithms," *Applied artificial intelligence*, vol. 10, no. 6, pp. 543–566, 1996.

[32] L. Nunez-Letamendia, "Fitting the control parameters of a genetic algorithm: An application to technical trading systems design," *European journal of operational research*, vol. 179, no. 3, pp. 847–868, 2007.

[33] M. Kaur and V. Kumar, "Beta chaotic map based image encryption using genetic algorithm," *International Journal of Bifurcation and Chaos*, vol. 28, no. 11, p. 1850132, 2018.

[34] J. Wang and L. Huang, "Evolving gomoku solver by genetic algorithm," in *2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*. IEEE, 2014, pp. 1064–1067.

[35] N. Cole, S. J. Louis, and C. Miles, "Using a genetic algorithm to tune first-person shooter bots," in *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, vol. 1. IEEE, 2004, pp. 139–145.

[36] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[37] ——, "Competitive coevolution through evolutionary complexification," *Journal of artificial intelligence research*, vol. 21, pp. 63–100, 2004.

[38] ——, "Evolving a roving eye for go," in *Genetic and Evolutionary Computation Conference*. Springer, 2004, pp. 1226–1238.

[39] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Evolving neural network agents in the nero video game," *Proceedings of the IEEE*, pp. 182–189, 2005.

[40] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, "A neuroevolution approach to general atari game playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 4, pp. 355–366, 2014.

[41] G. Johnson, V. Argyriou, and C. Politis, "Similarity model using gradient images to compare human and ai agents," in *2021 17th International Conference on Distributed Computing in Sensor Systems (DCOSS)*. IEEE, 2021, pp. 309–313.

[42] J. F. Shepherd III and K. Tumer, "Robust neuro-control for a micro quadrotor," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, 2010, pp. 1131–1138.

[43] Z. Wang, J. Hong, P. Liu, and L. Zhang, "Voltage fault diagnosis and prognosis of battery systems based on entropy and z-score for electric vehicles," *Applied energy*, vol. 196, pp. 289–302, 2017.

[44] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics*, 2017. [Online]. Available: https://arxiv.org/abs/1705.05065