Evo-ROS: Integrating Evolution and the Robot Operating System

Glen A. Simon*
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan, USA
simongle@msu.edu

Jared M. Moore
School of Computing and Information Systems
Grand Valley State University
Grand Rapids, Michigan, USA
moorejar@gvsu.edu

Anthony J. Clark
Computer Science Department
Missouri State University
Springfield, Missouri, USA
AnthonyClark@MissouriState.edu

Philip K. McKinley
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan, USA
mckinley@cse.msu.edu

ABSTRACT

In this paper, we describe the Evo-ROS framework, which is intended to help bridge the gap between the evolutionary and traditional robotics communities. Evo-ROS combines an evolutionary algorithm with individual physics-based evaluations conducted using the Robot Operating System (ROS) and the Gazebo simulation environment. Our goals in developing Evo-ROS are to (1) provide researchers in evolutionary robotics with access to the extensive support for real-world components and capabilities developed by the ROS community and (2) enable ROS developers, and more broadly robotics researchers, to take advantage of evolutionary search during design and testing. We describe the details of the Evo-ROS structure and operation, followed by presentation of a case study using Evo-ROS to optimize placement of sonar sensors on unmanned ground vehicles that can experience reduced sensing capability due to component failures and physical damage. The case study provides insights into the current capabilities and identifies areas for future enhancements.

CCS CONCEPTS

• **Computer systems organization → Evolutionary robotics**;

KEYWORDS

Evolutionary robotics, autonomous vehicle, sonar, sensor placement, fault tolerance, Robot Operating System, Gazebo, Ardupilot.

1 INTRODUCTION

Mobile robotic systems are increasingly being deployed in a wide variety of applications, such as public safety, agriculture, manufacturing, and supply chain. Traditionally such systems were operated by remote control, however, advances in machine learning technologies are making it possible to build robots that are either partially or fully autonomous. Such systems are often required to operate in the face of uncertainty: e.g., noisy communication, poor weather, unexpected human input, faulty or damaged sensors and actuators. To successfully complete tasks despite adverse events and conditions, systems must adapt to those situations. How can we design the physical robot and its control software so that it can operate effectively in such environments? The solution space for this problem is enormous, making exhaustive search impractical.

The field of evolutionary robotics (ER) [13] attempts to address this challenging problem by harnessing the open-ended search capabilities of evolutionary algorithms. An artificial genome specifies the robot's control system and possibly aspects of its morphology. Individuals in a population are evaluated with respect to one or more tasks, with the best performing individuals selected to pass their genes to the next generation. Evolutionary approaches have yielded effective controllers and physical designs for a variety of crawling, swimming, and flying robots [4, 16]. Our own research has applied evolutionary algorithms to optimize both morphology and control in aquatic and terrestrial robots [8, 17]. From an engineering perspective, a major advantage of evolutionary search is the possible discovery of solutions (as well as potential problems) that the engineer might not otherwise have considered.

Simulation is an essential component of ER, greatly reducing the time to evolve solutions while avoiding possible damage to physical robots. The ER community typically creates one-off simulation environments by selecting from a few different physics engines (e.g., ODE, Bullet, VoxCAD, Simulink, DART) to evaluate a candidate solution. Environments are sparse, generally featuring the robot and possibly a few obstacles. Tasks typically comprise locomotion, navigation, and basic problem solving. Robots themselves contain only a few sensors, most often custom developed for the specific experiment being conducted. Hence, ER tasks are often limited by the scope of the simulation environment and how much time a developer has to code obstacles, sensors, and the platform itself. Models are not necessarily shareable between developers due to a lack of standardization. While many research questions can, and have, been answered by simple simulations, it becomes difficult to address more complex questions in these environments.

*Glen Simon is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208269

In contrast, the broader robotics community can address very complicated tasks, utilizing multiple sensors and actuators to interpret and navigate complex and often adverse environments. The Robot Operating System (ROS) [20] was developed to facilitate reuse of control software across projects, and it has gained a large following in recent years. Together with the Gazebo physics simulator [14], these tools provide tested models of commercially available hardware, enabling study of high-level behaviors while saving developer time during the design phase. Our early experiments [7] demonstrated the potential benefits of integrating ROS/Gazebo with evolutionary search, but the customized nature of the simulation environment illustrated the need for a more general platform.

The main contribution of this paper is to describe Evo-ROS, a software framework integrating evolutionary search, ROS, and Gazebo. The current design provides a genetic-algorithm front-end and distributes evaluations across many ROS/Gazebo worker instances. To our knowledge Evo-ROS is the first ER system that includes simulation tools regularly employed by the broader robotics community. Evo-ROS internals are described in Section 3. Evo-ROS is open-source; a github link is available at the end of the paper. To demonstrate the operation of Evo-ROS, we conducted a case study to optimize the placement and configuration of sonar sensors on unmanned ground vehicles (UGVs) that may experience random sensor failures and loss of multiple sensors due to physical damage. The target platform is the Erle-Rover [11], a commercial terrestrial robot which we have tasked with waypoint following under sensor uncertainty. Figure 1 shows the Erle-Rover, its simulation model, and a physical rover equipped with additional sensors. Sections 4 and 5, respectively, describe the experiments and results of the case study. Finally, in Section 6, we identify areas for improvement by discussing issues that arose during the case study.

2 BACKGROUND AND RELATED WORK

As with natural evolution, results of many ER studies exhibit a tight coupling between aspects of morphology, such as sensor positioning, and the controller [5]. Moreover, the resilience of natural organisms has led researchers to apply evolutionary search in order to enhance engineered systems. Bongard et al. [4] demonstrated the potential of evolution in *self-modeling* terrestrial robots, where the system maintains an internal "mental image" of itself and can evolve compensatory behaviors to mitigate damage; their estimation-exploration algorithm has also been applied to aquatic robots [18]. Cully et al. [9] improved and extended this general approach to enable robots to adapt locomotion strategies in real time, based on sensory feedback.

Despite impressive results from the ER community, however, there remains a disconnect with the mainstream robotics community, which has also seen major advances in recent years. As noted above, ER simulations are typically developed in-house and are simple relative to commercial robots. Silva et al. [19] recently pointed out these shortcomings and suggested possible advantages of adopting tools from mainstream robotics for ER simulations. In particular, ROS and Gazebo provide tested models of commercially available actuators and sensors, saving the ER researcher time in constructing target platforms for evolutionary runs. Additionally, results have been shown to transfer to real robots, helping to address the "reality-gap" often encountered in ER [15]. The primary drawback of using high-fidelity simulation in an evolutionary algorithm is the overhead needed for evaluations. Our view is that this issue can be partially addressed through relatively small-scale parallelization and can eventually be marginalized with continuing advances in processing capability and larger-scale parallelization.

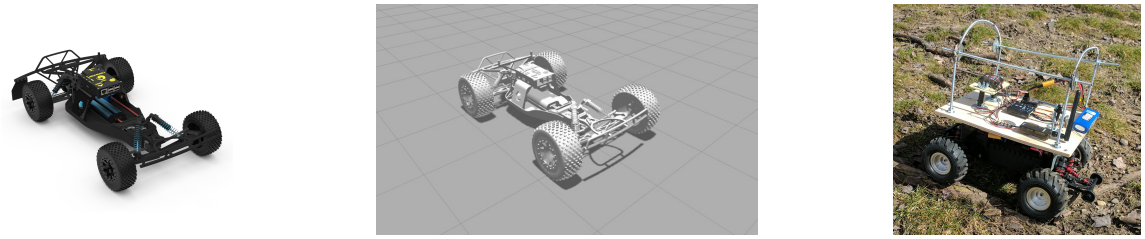
We have realized this approach with the Evo-ROS platform. The particular problem we address in the subsequent case study is optimal placement of sonar sensors while accounting for possible failures. Although these issues are of considerable interest to both the ER and mainstream robotics communities [3, 10, 21], most studies have focused on fault tolerance and not on sensor placement [22]. Evolutionary algorithms are particularly well suited to such problems, as they can search large solution spaces, unbiased by human preconception. Moreover, evolution can discover "unlikely-but-possible" situations that might otherwise result in system failure.

3 EVO-ROS FRAMEWORK

The Evo-ROS framework is intended as a bridge between the evolutionary robotics community and the broader field of robotics. Evo-ROS integrates evolutionary algorithms with evaluations constructed using popular software tools: ROS, Gazebo and, for the case study here, Ardupilot. ROS [20] is a publisher/subscriber framework for writing robot control software and includes a large collection of libraries realizing complex interactions among communicating components. Here, ROS is used to implement an obstacle avoidance algorithm for the UGV. The Gazebo simulator [14] includes models for a wide variety of commercial devices and enables the same control code to be used in both simulated and physical robots. Ardupilot [1] is an open-source autopilot stack capable of controlling terrestrial, aquatic and aerial vehicles. Ardupilot has built-in algorithms for the waypoint following task addressed in this study.

Together, ROS, Gazebo, and Ardupilot can produce a simulation of commercial robots operating in complex physical environments. However, these software packages are traditionally used during robot development to test new designs manually configured by the experimenter. Moreover, they typically simulate the target platform at real-time speed, often interacting with a user through a remote control interface identical to that used with the corresponding physical platform. Such a process is unsuitable for evolutionary search, where large numbers of simulations need to be conducted in an automated manner and, typically, much faster than real time.

Evo-ROS Structure. Evo-ROS comprises a set of ROS processes capable of spawning, managing, and changing simulations carried out by the software tools described above. Specifically, Evo-ROS defines the interface between an external evolutionary algorithm (a GA in the present study) and the simulation environment, enabling individuals from the GA population to be evaluated within the ROS/Gazebo/Ardupilot stack. A user would need to construct the genome for their specific problem, and integrate with the provided GA code. In a typical evolutionary run, multiple Evo-ROS instances are executed in parallel across virtual machines (VMs). This configuration enables the large number of simulations required for a typical evolutionary run.



(a) Physical Erle-Rover frame (b) Gazebo simulated Erle-Rover (c) Physical rover frame equipped with sensor board, sensors, and a roll cage

Figure 1: The unmanned ground vehicle platform used in this study.

Figure 2 depicts the main components of Evo-ROS and their interaction for an individual evaluation. The Evo-ROS "core" consists of three main components: the transporter, software manager, and the simulation manager. The transporter is responsible for maintaining communication between an instance of Evo-ROS and the external GA via TCP sockets. The software manager is responsible for spawning and managing the processes within Evo-ROS, including control (ROS) and simulation (Gazebo) software. A user only has to start the software manager process which then instantiates all the necessary software (ROS, Gazebo, Ardupilot, etc.). The simulation manager monitors various aspects of an individual Gazebo simulation, as discussed below. The rover is equipped with a ROS-based navigation controller that includes waypoint following and obstacle avoidance modes as well as a mechanism for triggering transitions between these modes. These modes are covered in detail in Section 4. For the case study, commands generated by ROS-based controllers are sent via MAVROS to the Ardupilot software running on the rover. They are then transmitted over a TCP link to the vehicle using the MAVProxy protocol. Note: The Ardupilot and MAVROS components can be easily swapped out for other low-level control libraries.

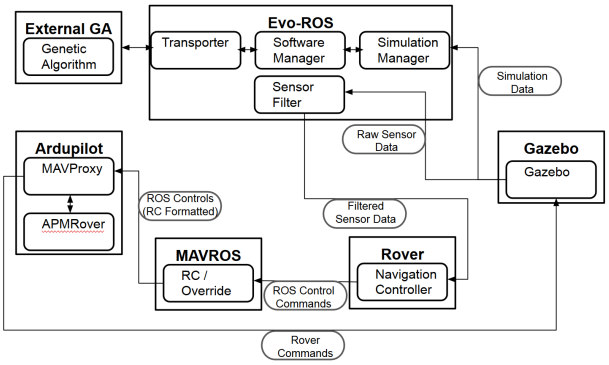


Figure 2: Main processes and communication channels Evo-ROS management of an individual simulation environment.

As shown in Figure 2, an additional component, the sensor filter, has been added to Evo-ROS for this study. One of our design goals with Evo-ROS is to allow simple integration of specific components. The sensor filter is a lightweight process that intercepts raw sensor data from Gazebo and applies filters to the data before forwarding it to the UGV controller. This filtering could involve injecting noise into sensor readings, introducing delay between when the data is read in Gazebo and delivered to the controller, or selectively neglecting to forward data so that the controller does not receive the readings from specified sensors. Filtering enables simulating sensor failures in the case study described later.

Evo-ROS Workflow. Figure 3 depicts the workflow of evaluating an individual. First, the GA encodes the attributes of the individual in a genome. Attributes of an individual could be either behavioral, such as parameter values for various controllers, or physical, such as the number, type, or location of sensors with which this individual is equipped. The genome is transferred via a TCP connection to the transporter process within a single Evo-ROS instance. Once received, controller parameters and physical traits from the genome are transformed into corresponding ROS parameters. The transporter then sends a ready flag, via the appropriate ROS topic interface, to the software manager process. The software manager determines whether a new simulation environment needs to be spawned or if the previous one can simply be reset. (To reduce overhead, when evolving controller parameters with unchanging physical traits, the simulation process is persistent and only needs to be reset before starting a new evaluation. However, when physical traits of the robot are being changed, such as evolving sensor placements, the software manager must tear down the simulation environment and modify the robot's unified robot description format (URDF) model file to reflect the changes.) The software manager then spawns the various simulation components, as described above, as well as a simulation manager process, and waits for each to initialize. It then hands control to the simulation manager and waits until the simulation session completes.

The simulation manager handles an individual evaluation in the physics simulation. It monitors information within both ROS and Gazebo, including several metrics describing the state of the simulation. In this study, such metrics include the speed of travel, progression through the mission, distance to each waypoint, collisions with any objects, termination conditions (successful completion of the mission, reaching the end of the allotted evaluation time). At the conclusion of an evaluation, the simulation manager reports the performance of the individual to the software manager. The software manager forwards the performance report to the transporter, which relays it to the GA for calculating fitness. The software manager then cleans up the simulation environment. This process is repeated for each individual sent to the Evo-ROS instance.

Current Limitations. A limiting factor of the Evo-ROS framework using the Ardupilot control framework is the fact that simulations must be capped at near real-time due to the integral 400Hz update

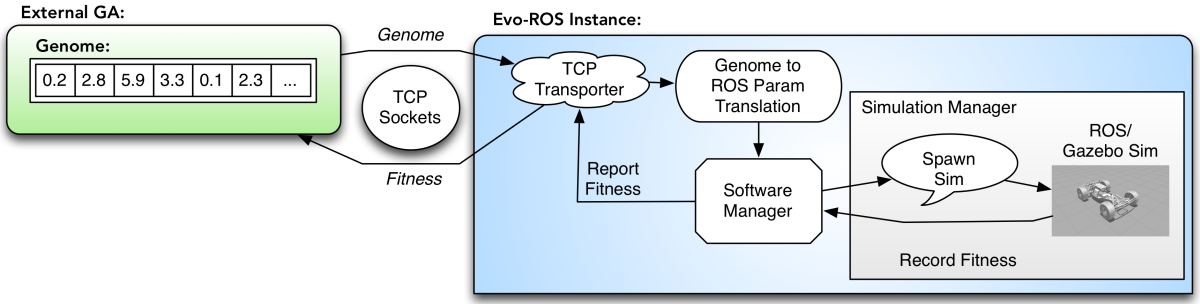


Figure 3: The workflow of evaluating an individual genome using an Evo-ROS instance.

loop in the Ardupilot stack. Furthermore, due to the distributed nature of the ROS system, individual processes within a single ROS instance communicate through a socket-based architecture. Typically, this means that only a single ROS/Gazebo instance can be run on one physical machine. We address this issue by creating an individual virtual machine (VM) for each ROS/Gazebo instance and extend the communication "fabric" to include a pool of VMs running on a cluster of physical nodes.

Figure 4 illustrates our parallelization strategy. Each Evo-ROS instance communicates with the external GA through the transporter process. Instances of Evo-ROS are spawned on multiple VMs spread across several physical machines. This configuration allows the population of the GA to be distributed across many identical simulation environments, greatly reducing the overall evaluation time per generation. The limiting factor is the number of individual VMs that can be spawned across a compute cluster.

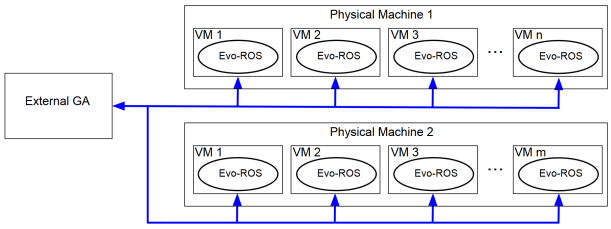


Figure 4: Parallelization of evaluations in Evo-ROS.

4 CASE STUDY: UGV SENSOR RESILIENCE

To demonstrate Evo-ROS on a problem of current research interest, we conducted a case study to optimize sensor placement on a commercially available UGV using prebuilt control/sensor algorithms from the ROS/Gazebo community. Figure 1a shows the target UGV, the Erle-Rover, a car-like robot controlled by ROS [11]. Erle Robotics has made available a simulated model of this platform, shown in Figure 1b. The simulated model matches the dimensions and mechanical capabilities of the physical rover, which is is 32.5 X 46.5 X 14.5cm with a wheel base of 33.4cm. As shown in Figure 1c, we have augmented our rover with a mounting board to hold sensors, instruments, and battery packs. To protect the on-board electronics from impact, a roll cage has also been installed. This platform enables the investigation of several questions related to resiliency, including optimal sensor placement, discovery of execution modes for different conditions, and unwanted feature interaction.

Sensor Placement. We conducted several preliminary experiments where we allowed the number of sensors and their locations to evolve; initially, placement and orientation were not required to be symmetric. However, we observed little convergence in those runs, and so we enforced symmetry in all subsequent runs. Given space limitations, we present a subset of those results here. Specifically, all individuals are equipped with six sonar sensors and symmetry is enforced on the their evolved locations and orientations, that is, they evolve as three symmetric pairs.

Figure 5 shows the valid regions of the vehicle where sensors can be placed, limited by the physical configuration of the existing Erle-Rover. Sensors are constrained to the outer 5cm on the front half of the rover. For this study, the controller does not drive in reverse so we do not consider placement on the rear half of the rover. Sensor orientation is also constrained to orient sensing regions to the front or sides of the rover. Failure models for sensors are described below.

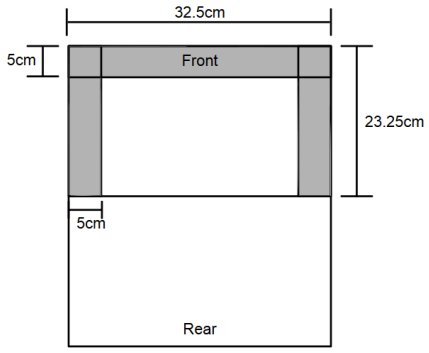


Figure 5: Sensors can be placed on the outer 5cm on the front half of the robot in the shaded region.

Control Hardware. Rover control is handled through a combination of Ardupilot waypoint navigation and an override when obstacle avoidance is required. Incorporating Ardupilot requires adding the `arupilot_sitl_gazebo_plugin` [6] to the simulation environment. Doing so ensures proper interaction between a Gazebo simulation and the Ardupilot autopilot, but forces simulations to run at Ardupilot's fixed 400Hz update rate. Evo-ROS implements a step-lock mechanism at each simulation timestep, synchronizing the Gazebo simulation and Ardupilot by pausing the simulation until a new movement command is received. Evo-ROS then steps the simulation by 2.5ms and returns new sensor measurements to

Ardupilot. Unlike typical Gazebo simulations where the simulator runs without waiting for commands, Ardupilot is the master of the simulation clock [12]. The availability of the simulated Erle-Rover model and the Ardupilot plugin provides an accurate representation of the physical rover and portability of evolved controller code between simulated and physical rovers. We emphasize, however, that Ardupilot is not integral to Evo-ROS. As discussed later, we are currently conducting studies that replace Ardupilot with other control software.

Evaluation. Figure 6 depicts one of the three environments used to evaluate individuals. Environments are specified in a Gazebo specific file format, facilitating reuse between experiments. Each rover carries out a pre-defined waypoint following mission, in two phases. First, a rover navigates through the waypoints within the environment, eventually returning to the starting location. If the rover successfully completes the first phase, it then must navigate the waypoints in reverse order, finally returning home again. Having two phases is intended to avoid sensor configurations from becoming overfit to missions favoring turns in one direction. In addition, the direction of the first phase is selected randomly to deter sensor configurations from "memorizing" the route.

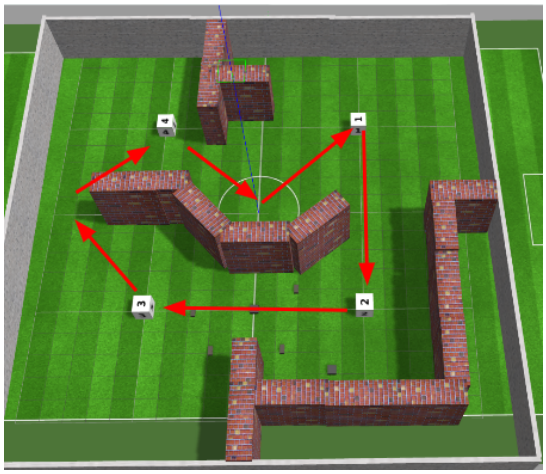


Figure 6: A sample first phase of a mission path consisting of four waypoints in oriented around a home location.

Fitness. Fitness reflects individual performance in three different environments, specifically:

$$fit = \sum_{n=1}^{3}(1+perc\_complete)^2+C*(1+perc\_time\_remaining)^2 \quad (1)$$

where *perc_compelete* represents the percentage of mission completed, C is either 1 or 0 indicating whether or not the mission was completed, and *perc_time_remaining* indicates how much of the allotted time remained at the completion of the mission.

Control. The control strategy is split into two different modes, (1) a waypoint following mode encoded in Ardupilot and (2) a ROS-based obstacle avoidance mode. Both modes pass commands to the simulated rover via UDP sockets maintained by Ardupilot. By default, the waypoint following algorithm governs navigation when no obstacles are detected by sensors [2]. Ardupilot navigates between waypoints using a serpentine driving behavior, sweeping the front of the rover through a 60 degree arc as it moves forward, rather than following a straight line. When at least one sensor detects an obstacle within a threshold distance from the vehicle, the obstacle avoidance mode preempts the waypoint following algorithm. Obstacle avoidance is implemented by switching Ardupilot into "Manual" mode and sending movement commands from a ROS script. Once the obstacle is cleared, waypoint navigation resumes.

The command flow for operating in these two modes can be seen in the bottom half of Figure 2. In the waypoint following mode, commands are generated by the APMRover process within Ardupilot. In the obstacle avoidance mode, commands are generated by the navigation controller on the rover. These commands are passed through the MAVROS process so that they can be translated to follow the MAVLink protocol. Regardless of the source, the MAVProxy process is responsible for the communication of the commands to either a physical rover or, in our case, a simulated one.

Obstacle avoidance commands are generated using a weighted voting algorithm. Each sensor that detects an obstacle casts a vote to turn either left or right, depending on the orientation of the sensor. For example, if a sensor on the front of the vehicle and angled slightly left detects an obstacle, it would vote to turn right. Each vote is weighted based on the proximity of the detected object. After each sensor has cast its vote, the right and left totals are calculated. A small difference, with both left and right votes above a certain threshold, indicates that an object is directly in front of the rover. The default response is to navigate left around the object. Otherwise, turning direction is determined by the sign of the difference between the left/right votes, negative indicating a left turn and positive indicating right. The strength, or sharpness, of the turn is determined by the following equation:

$$turn\_strength = 1 - \frac{(max\_range - \left|\sum_{i=0}^{6} w_i * d_i\right|)}{max\_range} \quad (2)$$

where *max_range* is the maximum detection range for the sensor, w_i is the weight and d_i is the turn direction vote for the i^{th} sensor. The summation takes into account all voting sensors. If *turn_strength* is 1.0 the rover will turn as sharply as possible. As *turn_strength* approaches 0 the turn becomes more gradual. After an obstacle has been avoided and no sensors report collision threats, the autopilot returns to the waypoint following behavior.

5 EXPERIMENTS AND RESULTS

As noted above, we present only a subset of the results of our experiments, where the UGV was equipped with six sonar sensors, and left-right symmetry enforced. We conducted three treatments. First, a greenfield (idealized) scenario establishes a baseline of performance. In this treatment, no sensors fail during the mission and the vehicle is able to operate under ideal conditions. In the next treatment, a single sensor randomly fails during the mission, simulating a situation where an electrical or mechanical failure arises in the robot. The final treatment simulates physical damage to the vehicle, wherein multiple sensors can fail based on their physical proximity to each other. We next describe the details of the evolutionary runs, followed by results for each treatment.

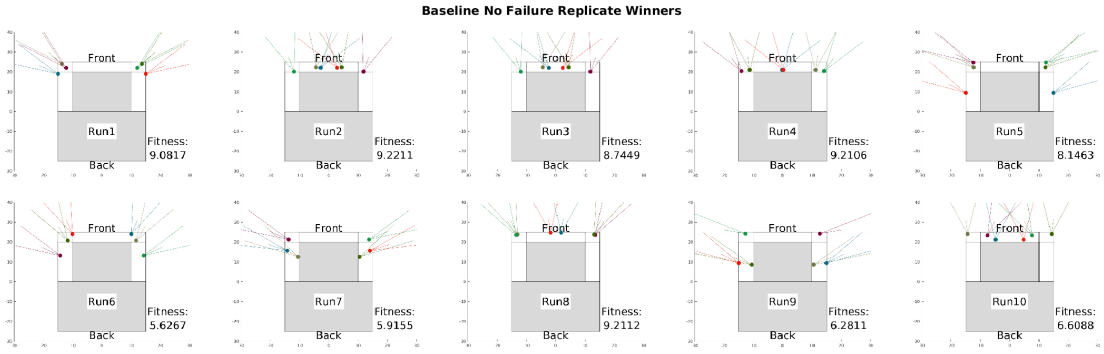


Figure 7: A panel image showing the winning sensor configuration for each replicate in the baseline treatment experiment.

Evolutionary Parameters. Table 1 lists the parameters of the GA used in this study. Each individual of the population represents a viable sensor configuration. Specifically, the genome of an individual consists of six behavioral traits relating to six individual sonar sensors. Each of these traits contain four values: a string describing the type of sensor such as 'sonar', the x and y position values of the sensor on the rover as well as a z value that determines the angle at which this sensor should face. After the population has been evaluated in the simulation environment, tournament selection, with a tournament size of two, creates a parent pool. Members of the parent pool produce the children pool using crossover with a probability of 0.25. If crossover does not occur, a child is cloned from a parent. The original population and the new individuals present in the children pool are combined into the candidate pool, which is larger than the original population size. A fraction of the members of the candidate pool will randomly undergo mutation, where the location and/or orientation of one or more sensors can be changed. If mutation or crossover create a new individual, it is marked as unevaluated. All such individuals are then evaluated via Evo-ROS before the next generation's population is selected. Elitism is used to maintain the highest performing individual with the rest of the population filled by performing tournament selection.

Table 1: Genetic Algorithm Parameters.

Population Size	30
Generations	25
Mutation Probability	0.15
Crossover Probability	0.25

A controller/sensor configuration is evaluated by measuring the performance of the UGV on waypoint following and obstacle avoidance tasks in three different environments, discussed in Section 4. Obstacles in the environments include tall walls, one meter wide cubes, and cinder blocks that are shorter than the height of sensors on the rover. Cinder blocks thus pose a potential challenge as they are visible to the rover only at distances greater than 0.5 meters, due to the cone shaped detection area of a sonar. We suspect that the cinder blocks will force the vehicle to evolve navigation strategies that maintain a distance threshold from obstacles in order to avoid losing the ability to detect very close objects.

Ten replicates are conducted per experiment. While this number is considered low for ER experiments, the simulation of each mission is computationally expensive and is limited to real time due to the use of Ardupilot. An individual replicate takes between 16 and 24 hours of wall clock time, depending on the average performance of individuals in the replicate. Attempting to run simulations faster than real-time results in random behavior, as we cannot assure the control signals are consistent across multiple simulations of the same controller/sensor configuration. We plan to address this in future work by replacing the Ardupilot control stack with a custom autopilot stack, enabling faster than real time simulation. That said, a goal of this initial case study was to apply evolution to a target system *exactly* matching one from the robotics community, including the use of Ardupilot.

Baseline Experiments. The UGV is equipped with six sensors and symmetry is enforced across experiments. In the first treatment, all sensors perform nominally, reporting on time with accurate data. Figure 7 shows the sensor configuration of best performing individual in each replicate. Two main configurations emerge. First is an array of sensors spread across the front of the vehicle, with slightly varying angles. This configuration gives the UGV full sonar coverage directly in front, and sightly to its sides. Second, the sensors "drift" toward the sides of the vehicle. Neglecting frontal coverage may seem like a weakness, but it appears that this configuration exploits an artifact of Ardupilot's waypoint navigation behavior. Specifically, the UGV drives in a serpentine pattern towards each waypoint, causing the orientation of the rover to swing in an arc of up to 60 degrees. Sensors mounted facing the side of the rover thus sweep through the area in front of the rover, while also providing information to the controller about the peripheral environment.

Figure 8 overlays the results of the 10 replicates on a single rover. Common sensing cones and placement are indicated by darker shaded areas. As seen in Figure 7, the sensors primarily sweep the forward viewing cone or cover the sides.

A scatter plot of the evolutionary progress of a sample run is shown in Figure 9. Here each individual is plotted according to its fitness score and generation. Individuals are color-coded based on their generation, thus individuals from the same generation share the same color. The fitness of the best fit individual and the average fitness for the generation are also plotted. It can be seen that most increases in fitness take place during the first 15 generations, after

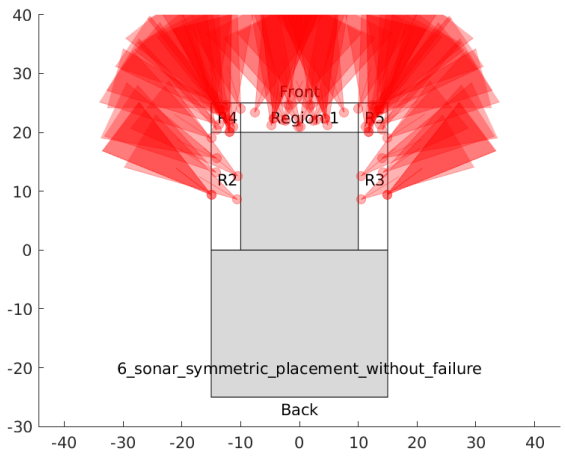


Figure 8: Overlay of sensor placements and viewing areas from 10 replicates of the baseline treatment experiment.

which fitness plateaus, and the average fitness for each generation approaches that of the most fit individual.

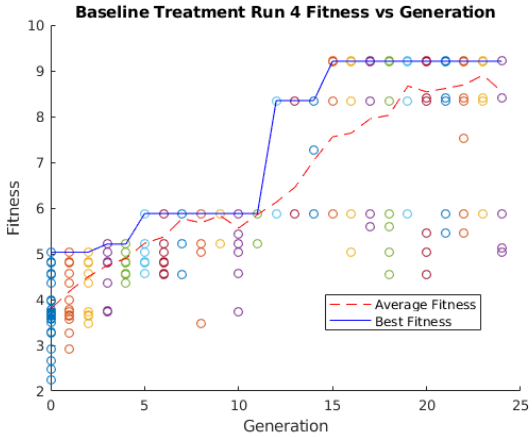


Figure 9: Plots of the average and best fitnesses in the population over 25 generations for a sample replicate of the baseline treatment.

Random Sensor Failures. In the second treatment, a randomly selected sensor fails between 40 and 80 seconds into an individual simulation. The UGV is allowed 540 seconds to complete a mission, with the optimal time being between 360 and 400 seconds, depending on the distribution of obstacles.

Sensor placement for the random failure treatment was similar to the baseline treatment. Again two configurations emerge, the array of sensors across the front of the vehicle and also spread along both sides. However, the results of this treatment evolve configurations with redundant sensors placed next to each other. In six of the ten replicates, redundancy is built into the configuration by having two sensors with nearly identical placement and orientation. Even when two sensors are not nearly identical in placement, evolved solutions tend to have multiple sensors dedicated to covering the same area. It appears that, in the case of random failures, evolution favors redundancy over a wider sensing capability.

Spatially Correlated Sensor Failure. Spatially correlated sensor failure simulates physical damage to a robot. Figure 10 depicts the spatial failure model centered on a sensor selected at random. Sensors within the affected area are also damaged and stop reporting. We hypothesize that this will pressure the sensors to be more evenly distributed across the robot and perhaps also demonstrate redundancy of coverage among the six sensors.

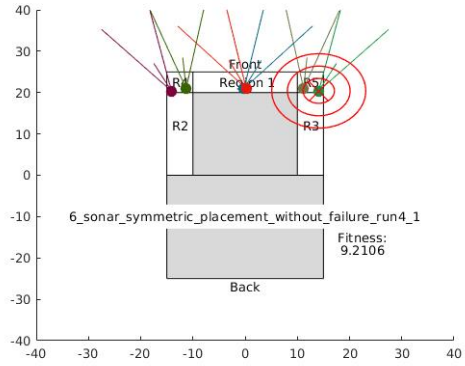


Figure 10: The areas impacted by the spatially correlated failure model on a sample sensor configuration.

Figure 11 plots the sensor placement of the highest performing individual in each of the ten replicates. As in the baseline treatment, two primary configurations evolve. The first remains an even spread of sensors across the front face of the rover. The second is focused on the sides of the platform. Surprisingly, there are clusters of sensors that would fall within the failure radius of the model. Apparently, the position of these sensors is more important than the risk of damage from sensor failures.

6 CONCLUSIONS AND FUTURE DIRECTIONS

Evo-ROS is intended as a bridge between the ER and traditional robotics communities. Leveraging the ROS/Gazebo simulation stack, evolutionary optimization can be applied to simulated robotic systems with pre-built and tested models of commercially available hardware. This approach should reduce developer time requirements in building an accurate simulation, and also increase the scope of available environments/platforms for the ER community. To address the execution time needed for high-fidelity simulations, Evo-ROS provides an interface to parallelize evolutionary runs across multiple VMs.

In a case study, we investigated optimization of sensor placement on the Erle-Rover, a commercially available UGV, equipped with with pre-built sensor libraries provided by ROS/Gazebo. Evolved solutions exhibit both expected (front facing) and unexpected (side facing) sensor deployments. Resilience to damage is an important need in robotic systems, but can be challenging to design into a system. Even in the presence of sensor failure, evolved solutions are able to complete the waypoint following tasks. Evolutionary

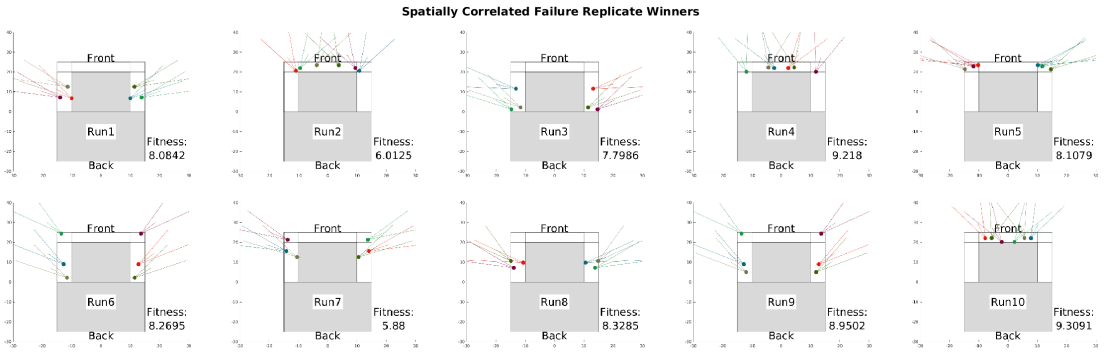


Figure 11: A panel image showing winning sensor configuration for each replicate with spatially correlated failures.

search explores the possible space and can potentially suggest novel solutions to address this issue, as observed in this study.

While conducting the case study, we identified three areas for improvement/ongoing development. First the Ardupilot control software employed on the rover limits the speed of simulations to real time. While Ardupilot-controlled commercial platforms are readily available, this controller is intended primarily for the hobbyist market. In ongoing work, we are removing Ardupilot from the control stack of the rover, and will instead utilize a purely ROS-based controller. Doing so enables faster simulations and should not limit the available software stacks for control and sensing, as many other ROS/Gazebo software libraries are available.

Second, the use of Ardupilot and its associated MAVProxy component limited our initial experiment to a single robot per environment. Large-scale robotics problems might involve many robots per simulation. Our ongoing work eliminates the Ardupilot/MAVProxy dependency, allowing Evo-ROS to be used with simulations containing multiple robotic agents.

Finally, the ROS/Gazebo software stack can have a high initial investment in terms of start-up and configuration. We are currently assembling a virtual machine, preconfigured with Evo-ROS and corresponding tutorials, greatly reducing the time for new users to install and use Evo-ROS. Rather than requiring individual installation of the many needed software packages, the VM will contain all necessary software, ready to run. A user can download and deploy a VM on their own system, or distribute many instances across a computing cluster and begin parallelized runs immediately. By eliminating many of the time-consuming challenges we encountered in developing Evo-ROS, we hope to facilitate use of Evo-ROS by both the ER and mainstream robotics communities.

ACKNOWLEDGMENTS

This work was supported in part by grants from the U.S. National Science Foundation and the Air Force Research Laboratory. Additional support was provided by Grand Valley State University.

REFERENCES

[1] ArduPilot. Developer website. http://ardupilot.org/about, 2018. Online; accessed 31 January 2018.
[2] ArduPilot Dev Team. Auto Mode. http://ardupilot.org/copter/docs/auto-mode.html, 2016. Online; accessed 30 January 2018.
[3] K. Balakrishnan and V. Honavar. On sensor evolution in robotics. In *Proceedings of the 1st Annual Conference on Genetic Programming*, pages 455–460, Stanford, California, 1996. MIT Press.
[4] J. Bongard, V. Zykov, and H. Lipson. Resilient machines through continuous self-modeling. *Science*, 17, November 2006.
[5] J. C. Bongard, A. Bernatskiy, K. Livingston, N. Livingston, J. Long, and M. Smith. Evolving robot morphology facilitates the evolution of neural modularity and evolvability. In *Proceedings of the 2015 Genetic and Evolutionary Computation Conference*, pages 129–136, Madrid, Spain, 2015. ACM.
[6] Buyval, Alex. Aurélien, Roy. Maxime, Lafleur. Ardupilot SITL Gazebo Plugin. https://github.com/AurelienRoy/ardupilot_sitl_gazebo_plugin/tree/master/ardupilot_sitl_gazebo_plugin, 2015. Online; accessed 31 January 2018.
[7] A. J. Clark. Evolving Adabot: A mobile robot with adjustable wheel extensions. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, Honolulu, HI, USA, 2017.
[8] A. J. Clark, X. Tan, and P. K. McKinley. Evolutionary multiobjective design of a flexible caudal fin for robotic fish. *Bioinspiration & Biomimetics, special issue on Bioinspired Soft Robotics*, 10(6), November 2015.
[9] A. Cully, J. Clune, and J. Mouret. Robots that can adapt like natural animals. *ArXiv Preprint*, 2014.
[10] D. Duckworth, B. Shrewsbury, and R. Murphy. Run the robot backward. In *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 1–6. IEEE, 2013.
[11] Erle Robotics. Erle-Rover. http://erlerobotics.com/blog/erle-rover/, 2018. Online; accessed 31 January 2018.
[12] Erle Robotics. Simulation Introduction. http://docs.erlerobotics.com/simulation/intro, 2018. Online; accessed 31 January 2018.
[13] D. Floreano, P. Husbands, and S. Nolfi. Evolutionary Robotics. In *Handbook of Robotics*. Springer Verlag, Berlin, 2008.
[14] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator, 04 2004.
[15] S. Koos, J. B. Mouret, and S. Doncieux. Crossing the reality gap in evolutionary robotics by promoting transferable controllers. In *Proceedings of the 2010 ACM Genetic and Evolutionary Computation Conference*, pages 119–126, Portland, Oregon, USA, 2010. ACM.
[16] H. Lipson and J. B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406(6799):974–978, August 2000.
[17] J. M. Moore and P. K. McKinley. Evolution of joint-level control for quadrupedal locomotion. *Artificial Life*, 23(1):58–79, January 2017.
[18] M. J. Rose, A. J. Clark, J. M. Moore, and P. K. McKinley. Just keep swimming: Accounting for uncertainty in self-modeling aquatic robots. In *Proceedings of the 6th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems*, Taormina, Italy, September 2013.
[19] F. Silva, M. Duarte, L. Correia, S. M. Oliveira, and . A. L. Christensen. Open issues in evolutionary robotics. *Evolutionary Computation*, 24(2):205–236, 2016.
[20] Tim Smith. About ROS. http://www.ros.org/about-ros/t, 2018. Online; accessed 31 January 2018.
[21] X. Wang, S. X. Yang, W. Shi, and M. Q. H. Meng. A co-evolution approach to sensor placement and control design for robot obstacle avoidance. In *2004 International Conference on Information Acquisition*, pages 107–112, Hefei, China, 2004.
[22] Y. Zhang and J. Jiang. Bibliographical review on reconfigurable fault-tolerant control systems. *Annual reviews in control*, 32(2):229–252, 2008.