Feasible and Stressful Trajectory Generation for Mobile Robots

Carl Hildebrandt
The University of Virginia, USA
hildebrandt.carl@virginia.edu

Sebastian Elbaum
The University of Virginia, USA
selbaum@virginia.edu

Nicola Bezzo
The University of Virginia, USA
nb6be@virginia.edu

Matthew B. Dwyer
The University of Virginia, USA
matthewbdwyer@virginia.edu

ABSTRACT

While executing nominal tests on mobile robots is required for their validation, such tests may overlook faults that arise under trajectories that accentuate certain aspects of the robot's behavior. Uncovering such stressful trajectories is challenging as the input space for these systems, as they move, is extremely large, and the relation between a planned trajectory and its potential to induce stress can be subtle. To address this challenge we propose a framework that 1) integrates kinematic and dynamic physical models of the robot into the automated trajectory generation in order to generate valid trajectories, and 2) incorporates a parameterizable scoring model to efficiently generate physically valid yet stressful trajectories for a broad range of mobile robots. We evaluate our approach on four variants of a state-of-the-art quadrotor in a racing simulator. We find that, for non-trivial length trajectories, the incorporation of the kinematic and dynamic model is crucial to generate any valid trajectory, and that the approach with the best hand-crafted scoring model and with a trained scoring model can cause on average a 55.9% and 41.3% more stress than a random selection among valid trajectories. A follow-up study shows that the approach was able to induce similar stress on a deployed commercial quadrotor, with trajectories that deviated up to 6m from the intended ones.

CCS CONCEPTS

• **Computer systems organization** → **Robotics**; • **Software and its engineering** → **Software testing and debugging**; **Dynamic analysis**.

KEYWORDS

Test Generation, Stress Testing, Robotics, Kinematic and Dynamic Models

ACM Reference Format:
Carl Hildebrandt, Sebastian Elbaum, Nicola Bezzo, and Matthew B. Dwyer. 2020. Feasible and Stressful Trajectory Generation for Mobile Robots. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3395363.3397387

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397387

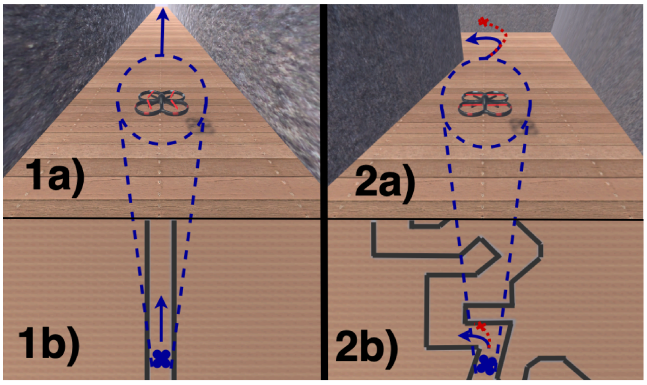


Figure 1: (1) shows a quadrotor flying in a straight corridor. (2) shows a more difficult trajectory through a winding corridor. (*a) show the quadrotor from behind, while (*b) show a bird's eye view of the quadrotor. The dashed lines convey location across views, solid arrows show optimal behavior, while dotted arrows show unforeseen behavior leading to a collision.

1 INTRODUCTION

Mobile robots are becoming more pervasive, ranging from autonomous cars [70] to micro-inspection drones [43], and this is raising awareness of the potential impact of faults in such systems, as evident in recent crashes [5, 26]. End-to-end system testing is a standard technique to detect such faults. System tests consist of executing a trajectory that resembles future deployment environments [9, 56]. For example, with autonomous cars, trajectories can be devised over existing road maps with typical traffic loads [61], over synthetic maps that meet road-design and traffic constraints [30], or over scenarios developed following certain even probability distribution [17]. A similar procedure is utilized for testing drones while accounting for the additional third spatial dimension [56].

While exploring typical trajectories is necessary to validate the behavior of mobile robots, it may overlook faults that arise in the presence of stressful trajectories, trajectories that accentuate a particular behavior of the robot. This is analogous to the motivation for stress testing software with harsh inputs as a complementary way to judge its robustness [4]. Consider, for example, the physical maneuvers that are required for a micro-drone to race through a tight tunnel. Intuitively, a trajectory as the one on the left of Figure1 will not subject the drone to the same level of stress as the one on the right side, which is full of tight turns that put pressure on the whole system, from the perception subsystem to the rotors.

The goal of this work is to provide an automated approach for the systematic generation of stressful trajectories for mobile robots.

Such an approach must overcome two fundamental challenges: 1) determining whether a trajectory is physically valid as a precondition to be feasible, and 2) efficiently identifying stress-inducing trajectories. A third and cross-cutting challenge is to remain general enough to support a broad range of mobile robots, stress measures, and ease of execution in both simulated and real environments.

To address the first challenge, we build on the insight that mobile robots' physical capabilities are usually approximated with kinematic and dynamic (KD) models. A KD model is a mathematical description of how the physical state of the robot is affected by any given input. For example, a KD model for a quadrotor may compute position, linear and angular velocity, and attitude based on the torque applied to the rotors. The proposed approach uses readily available KD models to compute the set of physically achievable future states of a robot[27]. Any state outside that set is physically infeasible.

To address the second challenge, we build on reachability algorithms using KD models to efficiently sample the valid physical space and incorporate a parameterizable scoring model that assigns a score representing stress to each trajectory as it is generated. Furthermore, the approach uses a beam search to incrementally explore the space of trajectories in order to identify and select the ones with the most potential to add stress to the system.

Finally, the challenge of remaining general is addressed by virtue of building on models that are available or easily approximated for most common mobile robot types, a high-degree of trajectory search abstraction and parameterization, and the use of the Robot Operating System (ROS) to standardize the message formats[59] and of open-source simulators to explore the trajectories as part of the implementation.

The primary contributions of this work are:

1) An automated approach for the efficient generation of physically valid and stressful trajectories for mobile robots, through the novel integration of kinematics and dynamics (system's physical aspects), the development of a parameterizable scoring model, and a configurable trajectory search process.

2) A tool pipeline[1] that implements the approach and is applicable to a broad range of mobile robots. In order to facilitate the evaluation, the implementation includes instances of an open-source quadrotor with four different software controllers and an existing commercial quadrotor.

3) An evaluation of the approach that demonstrates its benefits. A controlled evaluation shows that a KD model is crucial for generating trajectories of non-trivial length, while the introduction of handcrafted and learned scoring models increased stress by 56% and 41% on average over a random baseline. A case study on a commercial quadrotor demonstrated similar levels of induced stress, with the drone deviating up to $6m$ from its nominal trajectory.

2 BACKGROUND AND RELATED WORK

We begin with an overview of how mobile robots are currently being tested in §2.1, followed by an introduction to KD models in §2.2.

[1] Artifact available at: https://hildebrandt-carl.github.io/RobotTestGenerationArtifact/

2.1 Testing of Mobile Robots

The rise of autonomous cars and the impact of their potential failures have led to a resurgence of techniques for testing mobile robots. We point the reader to work by Stellet et al. [60] and Huang et al. [24] for overviews on techniques from the intelligent vehicle community. We now summarize efforts related to ours, organized by whether a technique is meant to operate at the system or component level, then we briefly discuss the closest work for generating stress-like tests for mobile robots, and finalize with a special mention for simulation, a key tool in the validation of mobile robots.

At the system level, the state of practice relies extensively on simulation, execution of predefined scenarios [14], and field deployment for testing [65]. For instance, in 2018, Waymo announced that it had completed 10 million miles of driving on public roads and over 7 billion miles in simulation [71]. State-of-the-art has concentrated on enabling the generation of test scenarios that account for a rich set of factors that appear in realistic contexts [57]. Many approaches have focused on the generation of images, a challenging input type that is fundamental to most mobile systems [28], and in the alteration of those images to expose faults [10, 63]. A natural progression of these efforts had led to domain-specific languages that can express images for realistic contexts[17]. Other emerging approaches target different types of inputs, such as control commands to manage acceleration, velocities, or positions [31]. All these approaches recognize that generating a full test environment is more complex and thus attempts to leverage existing information to guide test generation. For example, importing existing road maps instead of synthesizing ones [61], distilling police reports as models to guide the generation of the environment such that they resemble contexts associated with car crashes [18, 35], using an approximation to the system control model to guide the command generation [31], or incorporating traffic models [54].

Independent of the chosen approach to generate tests for mobile robots, a recognized challenge is the management of the enormous input dimensions and state-space[32]. To address these challenges existing approaches either reduce the input space, reduce the represented state space, or define a set of constraints to work within. For example, Loiacono et al. [37] use their domain knowledge to focus on race-tracks as the input space of autonomous cars. Althoff et al. [1] reduce the state space by setting the test scenario as constant and only optimizing the initial conditions of the vehicle under test. BaerkGu et al. [30] use the power of SMT solvers to generate a sequence of road segments that meet user design criteria. O'Kelly et al. [47] focused just on particular highway settings.

At the component testing level, we find many specialized input-generation techniques. For example, there are techniques targetting sensor and actuation components [8], control components [3, 40], image processing components [15, 36, 67], and reactive layers that include machine learning models. Among the latest, the software engineering community has generated an increasing body of knowledge on testing DNN's [11, 62, 77, 78].

The closest efforts to our work in terms of the integration of the system physical elements, aim to force robots to either operate along performance boundaries [45], or at maximizing exposure to unsafe behavior [1, 64]. These efforts are different from ours in two ways. First, they only aim to generate the initial set of

robot conditions instead of a whole trajectory. In theory, one could expose all valid stressful trajectories by just setting the initial robot conditions. In practice, however, identifying initial conditions that are representative of how the system operates in the real-world is challenging as they must avoid both unreachable conditions as well as conditions that are impossible to reproduce in real deployments. Second, none of them connects the KD model to generating tests that are guaranteed to be physically valid for the given robot. The work by Althoff et al. [1] makes the connection to KD models but uses them to favor tests with small reachability sets that represent tight operating spaces, which may not necessarily be stressful.

It is worth noting that simulation plays a crucial role in the validation of mobile robots driven by factors such as the time required to build a complete physical system prototype, the interactions with the physical world that require to mock at least part of that world, and the cost of field failures. Indeed, most of the approaches listed rely on various types of simulation support. Gambi et al. [19] used BeamNg[49] a vehicle simulator to find vehicle bugs using genetic algorithms. Dosovitskiy et al. [9] use a simulator, Carla, to prototype three types of autonomous vehicle. Among the many simulators available [53], in this work, we leverage recent advances in high-fidelity ones that provide not just accurate modeling of the world through sophisticated physics engines (which model, for example, gravity, friction, inertia), but also emulate a robot's sensors as it moves through the world (the atmospheric pressure, the distance to an object as measured by a laser scan, the images captured by a camera). Simulators like Carla, [9], Airsim, [56], and FlightGoggles [23] are increasingly providing such capabilities. In one of the studies in this work, we execute the generated trajectories by extending the FlightGoggles[23] framework with four additional controllers, and redesigning FlightGoggles rendering software using the Unity game engine[13] to allow the development of scenarios that do not rely on proprietary resources.

2.2 Kinematic and Dynamic Models

Kinematic models describe the motion of an object through measures such as position, velocity, and acceleration [2, 27, 72, 73]. Dynamic models describe the forces associated with the motion of an object [20]. Given an object's current state and a given input, these models can predict the object's future state. Such predictions are used in many fields including robotics[7], astrophysics[74], mechanical engineering[12], biomechanics[58], and game physics simulations[41]. Within the field of robotics, most systems are likely to base their development on a KD model, or can at least be approximated by an existing model.

In this work, we are specifically interested in using a KD model to describe how a robot's state $\mathbf{s}$ (e.g., position, velocity, acceleration) will change due to some input u. For instance a quadrotor KD model can be described using a 12^{th} order state system $\mathbf{s} = [x\ y\ z\ \phi\ \theta\ \psi\ v_x\ v_y\ v_z\ \omega_x\ \omega_y\ \omega_z]^T$, which describes the position, attitude, velocity, and angular velocity respectively [66, 75]. The input to the model is four motor speeds w_{1-4}. The speeds are used to calculate the values u_1 to u_4 as shown in equation 1. u_1 represents the thrust force upwards F generated by the four rotors. u_2 and u_3 represent the difference in thrust for both roll M_x and pitch M_y respectively. u_4 is the difference in torque between the two clockwise turning rotors and the two counterclockwise turning rotors which result in yaw M_z. d is the drone's arm length, and k_f and k_m are the proportionality constants for thrust and moments respectively.

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} F \\ M_x \\ M_y \\ M_z \end{bmatrix} = \begin{bmatrix} k_f & k_f & k_f & k_f \\ 0 & dk_f & 0 & -dk_f \\ -dk_f & 0 & dk_f & 0 \\ k_m & -k_m & k_m & -k_m \end{bmatrix} \begin{bmatrix} w_1^2 \\ w_2^2 \\ w_3^2 \\ w_4^2 \end{bmatrix} \quad (1)$$

The values of u_2, u_3, and u_4 are used to compute the change in quadrotors angular velocity ω using Equations 2, where the I terms correspond to the inertial properties unique to each quadrotor.

$$\begin{bmatrix} \dot{\omega_x} \\ \dot{\omega_y} \\ \dot{\omega_z} \end{bmatrix} = \begin{bmatrix} \frac{I_{yy}-I_{zz}}{I_{xx}}\omega_y\omega_z \\ \frac{I_{zz}-I_{xx}}{I_{yy}}\omega_x\omega_z \\ \frac{I_{xx}-I_{yy}}{I_{zz}}\omega_x\omega_y \end{bmatrix} + \begin{bmatrix} \frac{1}{I_{xx}} & 0 & 0 \\ 0 & \frac{1}{I_{yy}} & 0 \\ 0 & 0 & \frac{1}{I_{zz}} \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} \quad (2)$$

The quadrotor's angular velocity ω is then used to compute the change in the attitude of the quadrotor using Equations 3.

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & sin(\phi)tan(\theta) & cos(\phi)tan(\theta) \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi)sec(\theta) & cos(\phi)sec(\theta) \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \quad (3)$$

Finally, the change in velocity is computed using Equations 4. The new velocity is used to update the position of the quadrotor.

$$\begin{bmatrix} \dot{v_x} \\ \dot{v_y} \\ \dot{v_z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{1}{m} \begin{bmatrix} cos(\phi)cos(\psi)sin(\theta) + sin(\phi)sin(\psi) \\ cos(\phi)sin(\theta)sin(\psi) + cos(\psi)sin(\phi) \\ sin(\theta)sin(\phi) \end{bmatrix} u_1 \quad (4)$$

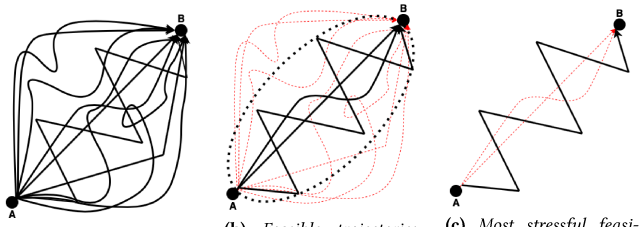
A KD model like the one introduced can be used to compute a robot reachable set, that is, the area or volume a robot can reach in a given amount of time. Efficiently calculating the reachable sets is its own research area [6, 22, 25, 34, 44, 68, 76]. For our work, a rough approximation is calculated using a sample of inputs to the KD model (w_{1-4} for the drone), and then computing the convex hull over the set of generated outputs. To the best of our knowledge, the only work that combines reachable sets with testing, minimizes the reachable sets in a given test scenario to maximize collision[1]. In contrast, we use reachable sets to create physically feasible tests.

3 PROBLEM STATEMENT

A physical space, W, is defined by a set of waypoints, $wy \in W$, with a designated origin, $o \in W$. A robot r, is capable of moving between a subset of waypoint pairs during a given time step, $valid(r) \subseteq W \times W$; a pair $(wy, wy') \notin valid(r)$ is said to be infeasible. A robot traversing to a waypoint, wy_i, will arrive in a state, s_i, that depends on both previous waypoint, wy_{i-1}, and previous state, s_{i-1}. For example, for a ground vehicle, a state may consist of the position wy, a velocity v, and a heading θ.

A robot traversing through a series of waypoints is following a trajectory $traj$. A trajectory of length N_{traj} is a sequence of N_{traj} states, $traj = \langle s_0, s_1, \ldots, s_{Ntraj} \rangle$, where each state is recorded at a given wy. The i^{th} wy in a trajectory is written as $traj[i]$.

The set of trajectories of length N_{traj}, $Traj$, is exponential in size $|Traj| = |W|^{N_{traj}}$. Many of these trajectories are **infeasible** – they cannot be realized by the implemented system. The feasible



(a) *Potential trajectories from A to B.*

(b) *Feasible trajectories within dashed oval.*

(c) *Most stressful feasible trajectory based on* $score(\cdot)$.

Figure 2: From the set of potential trajectories, the physically feasible, and most stressful ones are selected.

trajectories are those comprised of only valid steps, $Traj_f = \{traj \mid \forall 0 \leq i < n : (traj[i], traj[i+1]) \in valid(r)\}$.

Given a function, $score : W \times W \mapsto \mathbb{R}$, that defines the stress placed on r for a pair of waypoints according to a stress scoring criterion (e.g., maximum deviation from an intended trajectory), the stress for a trajectory is $score(traj) = \sum_{i=0}^{n} score(traj[i], traj[i+1])$. A key objective of this research is to compute the most stress-inducing feasible trajectories, $traj_s \in Traj_f$ such that $\forall traj \in Traj_f : score(traj) \leq score(traj_s)$.

This problem is illustrated in Figure 2 where Figure 2(a) depicts a set of trajectories $Traj$, the feasible trajectories $Traj_f \subseteq Traj$ are shown in Figure 2(b), and given a $score(\cdot)$ function the most stressful of the feasible trajectories, $Traj_s \subset Traj_f$, is found as shown in Figure 2(c).

4 APPROACH

The goal of the approach is the systematic and efficient generation of stressful trajectories for mobile robots. In this section we provide an overview of the approach describing the search for trajectories, a detailed description of how our approach identifies both feasible and stressful trajectories, a running example of the approach, and a brief description of the implementation.

4.1 Overview

The approach consists of two main algorithms. Algorithm 1 manages the search for trajectories through the use of an exploration frontier. The frontier consists of all the trajectories which are currently under consideration. Algorithm 1 expands the frontier through calls to *exploreFrontier* which is described in Algorithm 2. Algorithm 2 controls how the frontier is explored by only checking trajectories that are both feasible and stressful. In this section we take a detailed look at the workings of Algorithm 1.

Algorithm 1 manages the search for feasible stressful trajectories inside W. To keep the approach general, Algorithm 1 takes in ten parameters: (1) W: a world defining the physical volume in which trajectories will be executed, (2) N_{wy}: The number of waypoints to be explored in W, (3-4) wy_{start} and wy_{end}: a start and ending waypoints for the returned trajectories, (5) N_{traj}: the required length of a trajectory, (6) *Limit*: the total computation time allowed, (7) *KD*: the KD model of the robot, (8) *Res*: resolution of samples used to compute the reachable set, (9) *Width*: the number of trajectories explored and expanded during each loop of the algorithm, and

Algorithm 1: Trajectory Generation Manager

Input : W, N_{wy}, wy_{start}, wy_{end}, N_{traj}, Limit, KD, Res, Width, ScoringModel
Output: Traj_s

1 $\text{Traj}_s = \emptyset$
2 **while** *time* < *Limit* **do**
3 $\quad \text{Wy} = \text{randomWpSelect}(\text{W}, \text{N}_{wy})$
4 $\quad \text{G}_W = \text{graph}(\text{wy}_{\text{start}}, \text{wy}_{\text{end}}, \text{Wy})$
5 $\quad \text{s}_{\text{start}} = \text{estimateRobotState}(\text{Null}, \text{Start})$
6 $\quad \text{traj}_{\text{init}} = \{\text{s}_{\text{start}}\}$
7 $\quad \text{Frontier} = \{(\text{traj}_{\text{init}}; 0)\}$
8 $\quad \text{Traj}_c = \emptyset$
9 $\quad$ **while** $Traj_c == \emptyset$ *and* $|Frontier| > 0$ **do**
10 $\quad\quad \text{Frontier}', \text{Traj}_c = \text{exploreFrontier}(\text{G}_W, \text{wy}_{\text{end}}, \text{KD}, \text{Frontier}, \text{Res}, \text{Width}, \text{N}_{\text{traj}}, \text{ScoringModel}))$
11 $\quad\quad \text{Frontier} = \text{Frontier} \cup \text{Frontier}'$
12 $\quad$ **end**
13 $\quad \text{Traj}_s = \text{Traj}_s \cup \text{Traj}_c$
14 **end**
15 return Traj_s

(10) *ScoringModel*: a function used to select the most promising trajectories to be further explored.

The goal of Algorithm 1 is to find trajectories of length N_{traj}, that start and end at user-defined waypoints wy_{start} and wy_{end}. It represents W as a graph G_W, where the vertices are waypoints. Each vertex is connected to all other vertices by the shortest straight line between them, creating a complete graph. An edge represents the optimal path a robot should follow to traverse between waypoints. A path is created by combining sequences of vertices and following the edges between them. Paths through the graph represent all the possible trajectories in the world.

It starts by initializing the set of stressful trajectories $Traj_s$ to an empty set in line 1. Algorithm 1 repeatedly computes trajectories until it exceeds a computation time of *Limit* and then returns the generated $Traj_s$ in line 15. The $Traj_s$ are computed in lines 3-13 as follows. First, in lines 2-3, a graph is built through a process used by probabilistic roadmap planners (PRM)[29]. The graph's vertices consist of N_{wy} randomly sampled waypoints as well as the user-defined wy_{start} and wy_{end} waypoints. Lines 5-7, initialize a search *Frontier*. The *Frontier* is a set of trajectories and trajectory score pairs. A trajectory score represents an estimation of the trajectory induced stress on the robot. The stress is estimated using a scoringModel described later in §4.2.3. The *Frontier* keeps track of the explored trajectories and is incrementally expanded in *exploreFrontier*. The frontier is initialized with an $traj_{\text{init}}$ containing a single state, s_{start}, that is estimated by assuming no prior information, *Null*, and the starting waypoint wy_{start}.

The algorithm repeatedly invokes *exploreFrontier* in line 10 to incrementally expand the *Frontier* and search for complete trajectories; trajectories which start at wy_{start}, end at wy_{end}, and are length N_{traj}. Algorithm 2 describes *exploreFrontier*, which returns the newly explored frontier *Frontier'* and complete trajectories $Traj_c$ from each iteration.

Algorithm 2: Explore Frontier

```
1  Function exploreFrontier(G_W, wy_end, KD, Frontier, Res,
     Width, N_traj, ScoringModel)
2      Traj_c = ∅
3      Frontier' = ∅
4      SortedFrontier = sort(Frontier.scores)
5      for i = 0; i < Width; i++ do
           // Select From Frontier
6          traj = SortedFrontier[i].traj
7          Frontier = Frontier ∩ not traj
8          if |traj| == N_traj, and traj[N_traj].position == wy_end
            then
9              Traj_c = Traj_c ∪ traj
10         end
11         if |traj| < N_traj then
12             last_s = traj[last].state
               // Calculate Reachable Set
13             Reach = calculateReachSet(last_s, KD, Res)
14             for wy in (G_W ∩ Reach)) do
15                 new_s = estimateRobotState(last_s, wy)
16                 traj_n = traj ∪ new_s
                   // Expand Frontier
17                 Frontier' = Frontier' ∪ (traj_n, Null)
18             end
19         end
           // Assign Scores
20         Frontier' = assignScores(Frontier', ScoringModel)
21     end
22     return Frontier', Traj_c
23
```

4.2 Efficiently Exploring the Frontier

Algorithm 2 describes the four part *exploreFrontier* function. First, *exploreFrontier* selects trajectories from the *Frontier* based on trajectory scores. Second, *exploreFrontier* computes the physical space reachable by the robot given the current robot state. Third, *exploreFrontier* expands the frontier by building a new set of trajectories by estimating the robots future state at each waypoint within the reachable space, and then using the estimated state to build new trajectories. Finally, *assignScores* gives scores to each of the new trajectories in the frontier.

More precisely, *exploreFrontier* starts by sorting the current *Frontier* based on each trajectory score. The top *Width* trajectories are selected for further processing. The larger the *Width*, the more trajectories are explored per call to *exploreFrontier*, and the more computationally expensive the operation is. However, the larger the *Width*, the more likely the algorithm will process a trajectory that will induce stress as approximated by our scoring function.

Selecting from the frontier, in line 6-10, consists of removing the i^{th} most promising trajectory and checking if it meets the requirements to be a complete trajectory. If so it is added to the $Traj_c$ set. In lines 11-19, if the selected trajectory is shorter than N_{traj}, the search continues by expanding the selected trajectory and adding it to *Frontier*′.

Before the selected trajectory is expanded and *Frontier*′ computed, a reachable set *Reach* needs to be computed. *Reach* defines the physical space the robot can achieve in a time step given its current state. Thus all *wy* inside both G_W and *Reach* are feasible for the robot. More specifically, the reachable set is computed using the robots last known state $last_s$, the robots *KD* model, and a sample resolution *Res*. Computing *Reach* is described in §4.2.1.

Once all the feasible waypoints for the robot are known, the algorithm expands the frontier. A trajectory is a sequence of states. Thus for each of the feasible waypoints, a new robot state is estimated based on the robots last known state. For each of the possible future states, a new trajectory is created by appending the new state onto the current trajectory. Each new trajectory is then added to the frontier and scores assigned to them before being returned.

4.2.1 Reachability Analysis to Explore the Feasible Frontier. The computed reachable set allows the algorithm to precisely identify which waypoints in G_W are achievable given the robots *KD* model and $last_s$. Thus trajectories that the robot could not physically achieve can be rejected during trajectory generation, as opposed to during trajectory execution.

In this work, we explore two techniques to compute reachable sets and later compare them to a baseline technique that sets the entire space as reachable. The first approach over-estimates the reachable space, by setting the reachable set to a sphere around the current position, whose radius is equal to the maximum velocity the quadrotor can travel in $\Delta t = 1s$.

The second approach leverages the full KD model to compute the reachable set. Computing such reachable sets is an active area of research[6, 22, 25, 34, 44, 68, 76]. For simplicity, we implement a brute force technique to compute it. Given $last_s$, we generate a set of input samples and apply the KD model to produce a set of potential reachable states. The convex hull of this state set is computed to serve as an approximation of the reachable set. This approach requires Res^x evaluations of the forward KD model equations, where x is the number of input variables for the KD model equation, and *Res* is the number of input samples taken [7]. For example, in the case of the quadrotor, which we later study, there are 4 input variables, as shown in Equation 1. If permutations of 5 linearly sampled inputs are taken, the approach would need to perform $5^4 = 625$ computations resulting in 625 achievable future states.

4.2.2 Estimating Robot State for Trajectory Building. The robot's state at a new waypoint is estimated based on the robot's state at a previous waypoint and the current input. Approaches to state estimation can vary in cost and precision. At two extremes in this spectrum are estimators that 1) assume the robot is *at rest* when reaching a waypoint, and 2) solve the inverse of the KD model equations. The first is inexpensive, but imprecise and the second is precise, but expensive.

We implement a hybrid of these that uses only portions of the KD model equations to estimate state while setting the remaining state variables to their resting values. The portions of the state to reset are configurable. For example, for the drone systems we later study, the approach computes quadrotor velocity at each waypoint

Algorithm 3: Assign Scores

```
1 Function assignScores(Frontier, ScoringModel)
2     for traj in Frontier do
3         score = 0
4         for each pairOfStates in traj do
5             score += scoringModel(pairOfStates)
6         end
7         traj.score = score
8     end
9     return Frontier
```

(euclidean distance between the waypoints over the timestep), but sets the attitude and angular velocity to 0 at each waypoint (This would happen if a quadrotor entered a waypoint level).

4.2.3 Assigning Scores to Select Next Trajectory. The *scoringModel* is used as described in Algorithm 3. For each *traj* in the *Frontier* we start with an initial score of 0. The algorithm iterates through each pair of states in the trajectory and assigns a score to the state pair. The final trajectory score is then computed by accumulating the state pair scores for that trajectory.

The scores are assigned by a *scoringModel* and are calculated based on an estimate of the stress that the robot will incur. A good *scoringModel* will accurately estimate this stress given two robot states and associated waypoints. The stress is defined through a scalar stress metric. Depending on the application of the robot, stress can be measured using different stress metrics. For example, three possible metrics are maximum deviation, maximum acceleration, or total time. The only requirement is that the selected stress metric must be measurable during robot execution. The main stress metric we use in our study is maximum deviation, which is illustrated in Figure 3. The maximum deviation, a standard measure associated with navigation safety, is a measure of the largest error between the expected position of a robot and its actual position. In this work, we explored two classes of scoring models that we later compare to a baseline scoring model that randomly selects a score.

The first scoring model leverages a user's domain knowledge to create rules likely to maximize some goal, for instance in our case, maximizing deviation. In our evaluation, for example, we identified the trajectories velocity v_{in}, v_{out}, and the trajectory angle Θ, as shown in Figure 3, as attributes likely to be correlated to maximum deviation. For example, a large v_{in} and Θ correspond with the intuition that entering a waypoint with high velocity might result in a significant deviation if the robot is also required to take a sharp turn. In general, the effectiveness of such a model will depend on a domain expert's ability to identify the attributes as well as how closely the attributes align with the robot behavior, which depends on the robot planner, robot controller, and robot sensing and actuation capabilities.

The second scoring model learns from previous data. It consists of using a collection of trajectories generated using a random scoring model and subsequently identifying the factors that lead to particularly stressful trajectories. This knowledge can then be used to score future trajectories on their ability to cause stress. As an example, assume that there is a series of generated trajectories. The

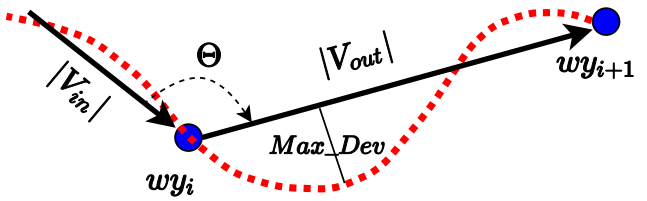


Figure 3: Trajectory attributes and the stress metric maximum deviation. The solid line is the expected trajectory while the dotted line is the true behavior.

robot could then execute the trajectories to render an actual maximum deviation. The traversed trajectories could then be broken down into pairs of waypoints like that of Figure 3. The maximum deviation associated with each pair of waypoints *max_dev* and a set of attributes that may be associated with that deviation (e.g., v_{in}, v_{out}, Θ) could be used as training data. Then a learning technique can be used to produce a *scoringModel* that, given a pair of waypoint attributes, can estimate the expected maximum deviation.

In our evaluation, we generated a *scoringModel* using a polynomial regression model where the loss function is the linear least-squares function, and regularization is given by the ℓ_2-norm[21]. We determined the best polynomial degree using 10-fold cross-validation. If the resulting model provides a good fit (i.e., strong correlation and low cross-validation loss), then it can be used to assign predictive scores to future trajectories without executing them. This approach incurs the cost of trajectory execution to generate the data to train the model. Thus its applicability depends in part on the cost of such execution. In many cases, such costs can be mitigated, for example through simulation, and it is beneficial in that it does not rely on the user's expertise.

4.3 Example Trajectory Generation

Figure 4 shows a step-by-step illustration of our approach. In this example, the G_W is generated using 6 random waypoints, we set *Width* to 2, and N_{traj} to 4. After PRM construction, we select from the frontier, which after the initialization in Algorithm 1 lines 3-8, is a single trajectory that contains wy_{start}. We calculated the *Reach* for the last and only waypoint (wy_{start}) in the trajectory as described in §4.2.1. We then expanded the frontier using each of the waypoints inside the W and *Reach*. The waypoints are added to the current trajectory by estimating three new states based on the current state and the new waypoint as described in §4.2.2. Scores are assigned to each of the new trajectories based on a scoring model as described in §4.2.3.

On the second iteration, due to the *Width* of 2, the two highest-scoring trajectories are selected from the frontier (filled circle). For each of the selected trajectories last waypoints, a *Reach* is calculated. The frontier is expanded using the waypoints in each *Reach*. This results in 4 new trajectories. Note that a waypoint can be used in multiple trajectories, as seen by the most central waypoint, which

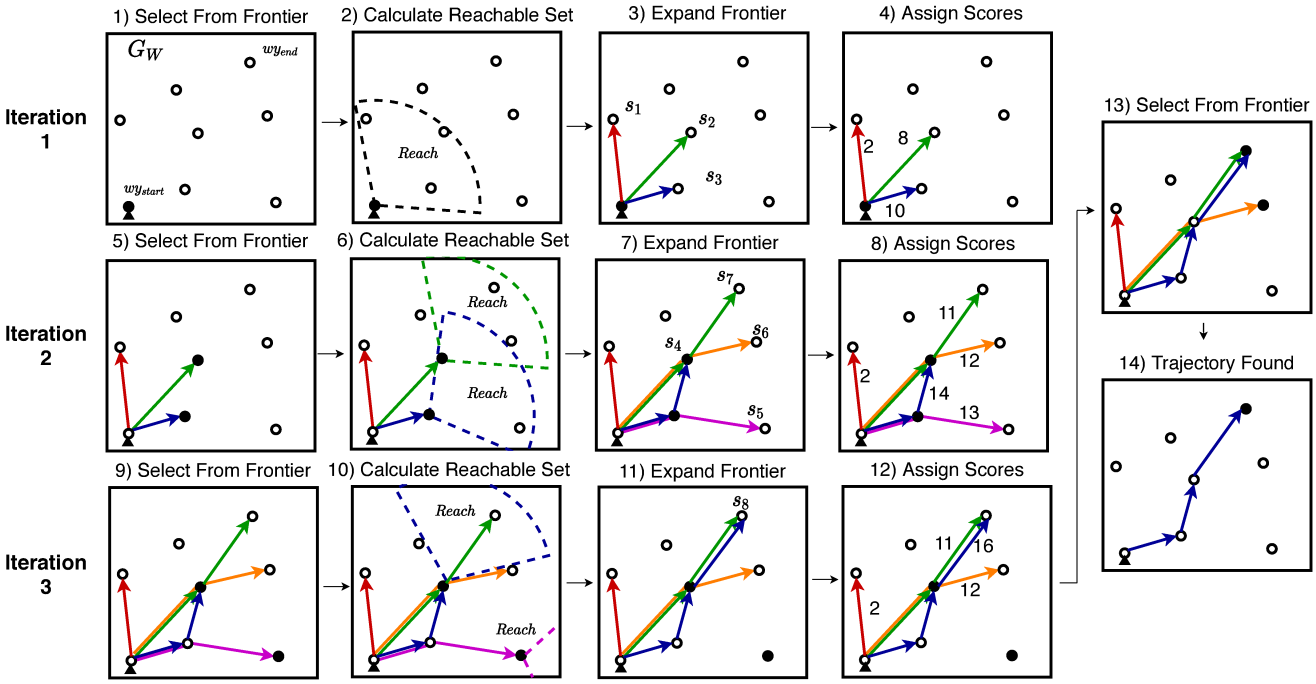


Figure 4: Our approach illustrated with waypoints (circles), trajectories (solid lines), and reachable sets (dotted lines). The example considers a 2D world with 6 random waypoints, a beam width of 2, and a trajectory length of 4. The mobile robot in this example, starts with 0 velocity and is facing directly upward (small triangle).

at this point is part of three trajectories. Scores are assigned to each trajectory and trajectories selected from the frontier.

On the third iteration the most central waypoint is again processed as it is the last state in the trajectory with a score of 14. Note however, the resultant *Reach* is different as the trajectory selected arrived at the waypoint with a different final state. The second *Reach* is computed for the trajectory with a score of 13. There are no waypoints inside this *Reach*, and thus the trajectory is removed from the frontier. Scores are assigned to the new trajectories and trajectories selected from the frontier. Two trajectories with a score of 16 and 12 are chosen for processing. The trajectory with score 16 meets the criteria of starting at wy_{start}, ending at wy_{end}, and being of length 4. This trajectory is added to the $Traj_c$, and the algorithm repeated with a new G_W. Although this is a hypothetical example, the approach still selects a stressful trajectory. The final trajectory requests the robot to take an immediate ≈70 degree right turn, followed by a ≈45 degree left turn before moving to wy_{end}.

4.4 Implementation

The implementation consists of 4 main software modules[2], as seen in Figure 5. The first module, trajectory generation, implements the approach as described in §4.1, while the next three modules run the experiments and are vital for collecting the data used later in the study.

[2]Artifact available at: https://hildebrandt-carl.github.io/RobotTestGenerationArtifact/

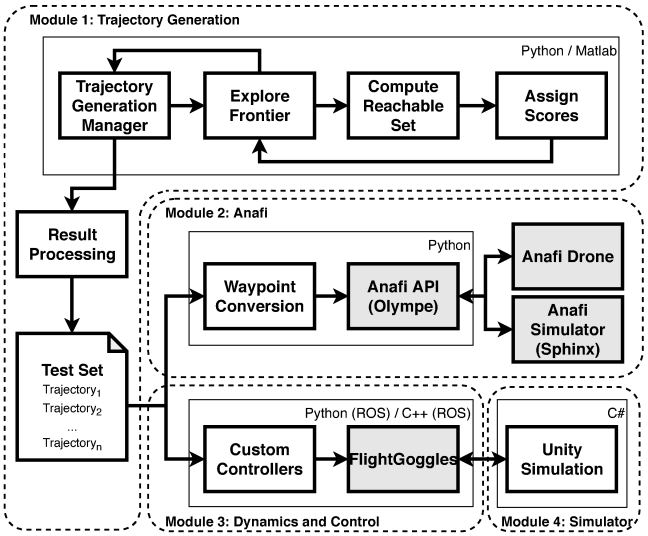


Figure 5: An overview of the implementation. Existing software is highlighted in a darker shade.

The first module consisted of both the trajectory generation and result processing toolchain. The trajectory generation uses the trajectory manager to explore the frontier using the reachable set and scoring model. The resultant trajectories are then processed and converted into data files and images that can be accessed by

both the Anafi and FlightGoggles control software. The majority of this module is implemented using Python3. The module consists of 36 python scripts with a total of approximately 7,000 SLOC. For certain functions, such as computing the convex hull, it was more convenient to use MATLAB, and so the approach calls these functions through the MATLAB API for Python[39].

The second module implements the integration with the Anafi quadrotor in both simulation[51] and the real-world through the Anafi API[50]. The module consists of software used to convert the trajectory into waypoints that are readable by the Anafi API. The Anafi API sends the waypoints to the Anafi quadrotor and records the returned GPS data through either a virtual ethernet or Wi-Fi connection.

The final two modules contain the control and simulator code to fly the FlightGoggles quadrotor. The FlightGoggles simulator has two parts. The first part emulates the dynamics and control of the quadrotor, while the second part simulates the quadrotors sensor data and collision information. At the time of writing, the FlightGoggles simulation uses proprietary graphics assets. Thus we only use the part of FlightGoggles that emulates the quadrotor dynamics, and we re-engineer the FlightGoggles simulation tool in Unity based on the available documentation. The control code uses ROS[59] and is written in C++. The implementation of the 4 custom quadrotors is integrated into the original code base using 11 Python classes consisting of approximately 2150 SLOC. The portions of the FlightGoggles simulator that were redeveloped in Unity are written in C#. The new simulator integrates with the base ROS code using the original TCP link in FlightGoggles. The new simulator uses assets that are freely available from the Unity store.

5 EVALUATION

The goal of the evaluation is to assess the proposed approach and determine what benefits the introduction of the KD model and scoring models has on automated trajectory generation for robots. More specifically, we aim to answer the following research questions for automated trajectory generation:

RQ1) Does the introduction of a KD model improve the ability to generate feasible and valid trajectories?

RQ2) Does the introduction of a scoring model improve the ability to generate stressful trajectories?

5.1 Setup

The test world is set to a $30m \times 30m \times 30m$ map with 250 randomly placed waypoints. This selection matches the volume ($27000m^3$) and size of a typical outdoor aerial testing facility[33, 46, 69].

5.1.1 Robot Configurations. The systems we used are listed in Table 1. The first is an autonomous racing quadrotor executed in the publicly available FlightGoggles simulator [23]. The quadrotor has a weight of $1kg$, and a body length of $0.45m$ [55]. Its maximum velocity in simulation is $18m/s$ [38].

The FlightGoggles quadrotor comes with a built-in angular rate controller to manage roll, pitch, and yaw. To evaluate the wide variety of trajectory following techniques exhibited by today's quadrotors, we implement four commonly used quadrotor controllers[66] into the FlightGoggles simulator. Two controllers are of a waypoint control type, using a cascade of three PID controllers; the first controls the angle of the quadrotor, the second controls the velocity of the quadrotor using the angle controller, the third sets the velocity of the quadrotor based on the distance to a waypoint. The first implementation replicated poorly written controllers that has overshoot and oscillation around waypoints. The second implementation mimics tuned controllers that are stable and converge to the waypoint. The next instantiated controller was a fixed velocity controller. This controller assigns a shared proportion of a fixed velocity over each the x, y, and z-direction based on the location of the next waypoint. We set the controller to maintain a velocity of $2m/s$, allowing the quadrotor to maneuver easily. The final controller computed a minimum snap trajectory and follows it using the waypoint PID controller. It was fundamentally different in that it builds a new trajectory through the waypoints that minimize snap, the 4th derivative of position[42], which means that it does not adhere to the assumption of the expected behavior being the shortest straight line between consecutive waypoints.

Table 1: Robot configurations studied

Robot Hardware	Robot Software	Execution
Flightgoggles Quadrotor[23]	Unstable Waypoint Controller[66]	Simulation
	Stable Waypoint Controller[66]	Simulation
	Fixed Velocity Controller	Simulation
	Minimum Snap Controller[42]	Simulation
Parrot Anafi Quadrotor [48]	Waypoint Controller[50]	Simulation
		Real World

A second quadrotor, the Anafi Parrot, is studied later in §6. Trajectories on the Anafi Parrot were executed both in simulation and the real-world using its proprietary control software.

5.2 Trajectory Generation with KD Models

To answer RQ1, we need to assess the cost and benefit of incorporating a KD model into the trajectory generation technique. To the best of our knowledge, there are currently no automated approaches or tools available for the automated generation of stressful target trajectories for mobile robots. The state-of-the-practice consists of handcrafted stress tests built by experts, which tend to be effective but limited in the scale of exploration. Thus, to identify the benefits explicitly introduced by the KD models, we adapted how the reachable set in line 13 of Algorithm 2 is computed using 3 different techniques. The first approach, **No KD**, returns all waypoints in the world, without considering any form of a KD model. The second approach weakly approximates the reachable set, **Approx KD**, by computing a sphere whose radius is the distance the quadrotor could travel at maximum velocity in $\Delta t = 1s$. The final approach, **Full KD**, uses a full KD model as described in §4.2.1. While expensive [22], this guarantees that all explored trajectories are valid by construction.

Each technique was given 2 hours to generate and execute trajectories. Algorithm 2 was set to have a beamwidth of 5 and trajectory length between 3 and 50. Varying trajectory length allows us to

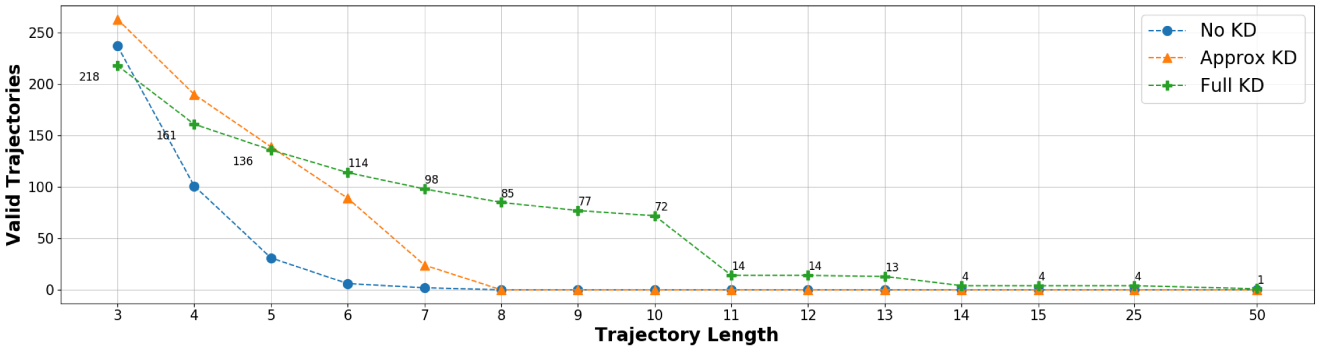


Figure 6: Valid trajectories generated of varying length.

assess the efficiency of techniques as the problem scales in complexity. For example, the number of possible trajectories of length 3 in a world with 250 possible waypoints is 1.5×10^7 and that increases to 4.1×10^{117} for trajectories of length 50.

For each technique, trajectories returned in line 22 of Algorithm 2 were checked for validity using the robot full KD model. For each valid trajectory, we model its execution time in proportion to its length and decrement the total experiment time. The number of physically valid trajectories returned by each technique within the 2 hour limit is shown in Figure 6.

Figure 6 shows that for simpler trajectories of length 3 for **No KD** and trajectories of length 5 for **Approx KD** the computationally cheaper approach produces more valid trajectories as opposed to our **Full KD** approach. This is because generating short, physically valid trajectories is easier, as only a few valid waypoints need to be selected. However, we can see that as the trajectories become longer and more complex, it becomes beneficial to use the computationally more expensive **Full KD** model. Figure 6 shows that **Full KD** start to outperform both **No KD** and **Approx KD** for trajectories of length 4 and 6 respectively, in terms of the number of physically valid trajectories produced. In fact, for trajectories of length 8 both **No KD** and **Approx KD** are unable to produce any valid trajectories in the given amount of time, while the **Full KD** can produce 85 valid trajectories. Even for trajectories of length 50, **Full KD** can still find 1 valid trajectory in the given time. This is because the **Full KD** approach provides information to Algorithm 2 on which waypoints in the world lead to invalid trajectories. This information, although expensive to generate, allows the search technique to reject invalid trajectories during trajectory generation as opposed to trajectory execution. This is especially important since the number of possible trajectories grows exponentially with their length – making pruning invalid paths cost-effective.

We then computed several performance metrics for valid trajectories of length 10. We ran each of the valid trajectories from the **Full KD** approach using the FlightGoggles simulator with the stable waypoint controller. Figure 7 shows the distribution of 3 performance metrics, namely: maximum deviation (m) from the optimal trajectory, the maximum acceleration (m/s^2) of the robot, and total execution time (s). We chose these because they are diverse in that deviation captures the potential for the robot to operate unsafely,

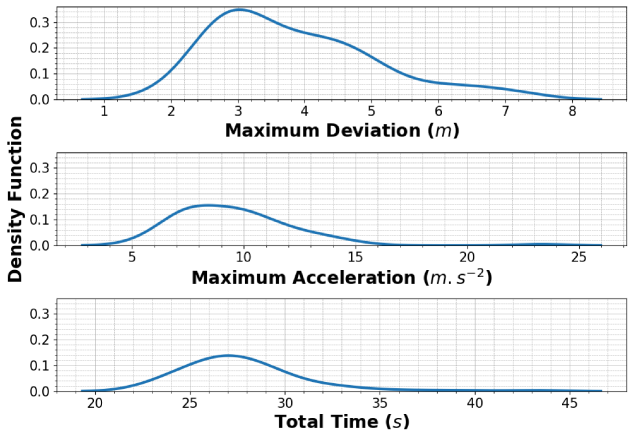


Figure 7: The distribution of performance metrics obtained by executing the FlightGoggles quadrotor in simulation.

acceleration captures the stress placed on the robot hardware, and total time reflects the robots ability to operate effectively.

Figure 7 shows that for each of the metrics, the quadrotor exhibits a broad range of possible values and that each of the distributions is positively skewed, with long tails to the right (where there is more stress). The fact that the distribution has long tails to the right shows that even though most trajectories produce little stress, there are trajectories which significantly stress the robot and lie outside the normal operating profile. Figure 7 shows that not only do the valid trajectories with no scoring model result in a range of behavior but that we are also able to measure multiple performance metrics on them.

RQ1 Findings: Although it is computationally more expensive to use a KD model, incorporating it into trajectory generation is critical for efficiently generating **valid** trajectories, especially as the trajectory length increases. We also found that, independent of the chosen measure, the stress induced valid generated trajectories shows high variability.

Table 2: The different scoring models and their descriptions.

High Velocity	Assigns high scores to trajectories with high velocities.
High Velocity + 90 Deg	Assigns high scores to trajectories with high velocities and include 90 degree turns.
High Velocity + 180 Deg	Assigns high scores to trajectories with high velocities and include 180 degree turns
Learned	Learns a scoring model based on the execution of prior trajectories

5.3 Incorporating a Scoring Model

To answer RQ2, we need to determine whether computing and including a scoring model, line 20 of Algorithm 2, leads to the generation of more stressful trajectories. We explore 4 different scoring models as described in Table 2. The first 3 scoring models are designed to represent scoring models designed by experts. Intuition tells us that for a quadrotor, the higher a robot's velocity, the more deviation we can expect given a turn. Using this intuition, three handcrafted scoring metrics were created. The first assigned higher scores to **High Velocity** trajectories without consideration to turns. The second assigned higher scores to trajectories that had both high velocity and waypoints that resulted in 90 degree turns (**High Velocity + 90 Deg**). The last handcrafted scoring model was similar to the second, except it placed a high score on 180 degree turns (**High Velocity + 180 Deg**).

These three approaches require domain knowledge, which is not always readily available. We thus tried a final scoring model, which **Learned** a scoring model based on the maximum deviation of each controller on the initial trajectories in RQ1. The learned scoring model uses 10-fold cross-validation to determine the polynomial degree used in a ridge regression model implemented using Pythons Scikit-Learn library[52]. For each of the software controllers tested in RQ2, we extract attributes from their initial execution. The input and output velocity, the angle between the waypoints, and the actual maximum deviation is extracted, as shown in Figure 3. Using this as training data, we produced four independent scoring models that, given a pair of waypoints, predict the maximum deviation for the respective software controller.

For each new scoring model, we generated a new set of trajectories using a total time of 1 hour, a beamwidth of 5, and a trajectory length 10. That is half of the time given in the RQ1 study to determine if the scoring model could produce more stressful resultant trajectories and do so in less time. For comparison, we also generated a baseline where each of the FlightGoggles software controllers was executed on the trajectory set generated using a **Full KD** model and no scoring model as per RQ1 with trajectories of length 10 and 2 hours of generation.

The resulting trajectories were run on each of the drone controllers, and the maximum deviation recorded.[3] To determine whether the introduction of a scoring model was beneficial, we divided each of the resultant maximum deviations with the mean maximum deviation from the baseline trajectory set. Thus any test that induced more stress and had a maximum deviation greater than the initial test set with no scoring model from RQ1, would result in a value greater than 1. Similarly, a test with a value of less than 1 means that it induced less stress than the average test in RQ1.

The results are shown in Figure 8. When considering only the handcrafted scoring models, Figure 8 shows that for each of the controllers, at least 1 of the 3 handcrafted scoring models results in a more stressful test set. For both waypoint controllers, including a scoring model that favors trajectories of high velocity result in test sets that are 70% and 76% more stressful. For the fixed velocity controller, a scoring model that favors 180 degree turns resulted in a test set that is 10% more stressful. The low increase in stress is attributed to the controller's slow constant speed, however, we note that our approach still finds test cases that are ≈40% more stressful than the given random test set. For the minimum snap controller a scoring model that favored 90-degree turns induces on average 69% more stress. These findings are consistent with the operation of these controllers. Moreover, taking the mean of the best scoring models shows that, on average, having a handcrafted scoring model results in a **55.9% increase in maximum deviation** on the stressful trajectories. These findings show that handcrafted scoring models are beneficial when domain knowledge is available.

Figure 8 also shows that for all controllers, it is possible to learn scoring models that can generate stressful trajectories for a specific quadrotor. This is useful, especially when there is no domain knowledge available, for instance, when testing a new robot. Moreover, the quality of learned models is high, since for each controller we found the learned model produced a distribution of performance metrics similar to the best handcrafted scoring model. Taking the mean of all scoring models showed that on average a learned scoring model **increased the maximum deviation by 41.3%**.

Recall that the experimental setup for RQ2 used half of the time compared to RQ1, so the observed improvements in the performance metrics were also significantly less costly to produce.

> **RQ2 Findings:** Introducing both handcrafted and learned scoring model into trajectory generation produces test that on average are **55.9% and 41.3% more stressful** than trajectories without a scoring model respectively. Moreover, learned scoring models can be generated without any prior domain knowledge.

6 FOLLOW-UP STUDY

We performed a preliminary study to explore the application of the proposed approach to a commercial drone operating in an outdoor flying cage of $30m \times 30m \times 30m$, and analyzed the differences between executing the trajectories in simulation versus the real-world. We selected the popular Parrot's Anafi quadrotor[48], which has a weight of $0.5kg$, maximum horizontal velocity of $15m/s$, and an arm length of $0.1m$. The Anafi has an autonomous flight mode, which can follow a series of waypoints through change positions commands using a controller that is not publicly available.

For this study, since we are not certain about the particular controller used by the Anafi, we learned a scoring model from an initial set of trajectories that we executed sending waypoints to Anafi's API. To reduce the cost of collecting the training set of trajectories,

[3] Due to space constraints and without loss of generality, we just use the maximum deviation since it relates to safety – the further a quadrotor is away from the expected trajectory, the more significant the safety risk.

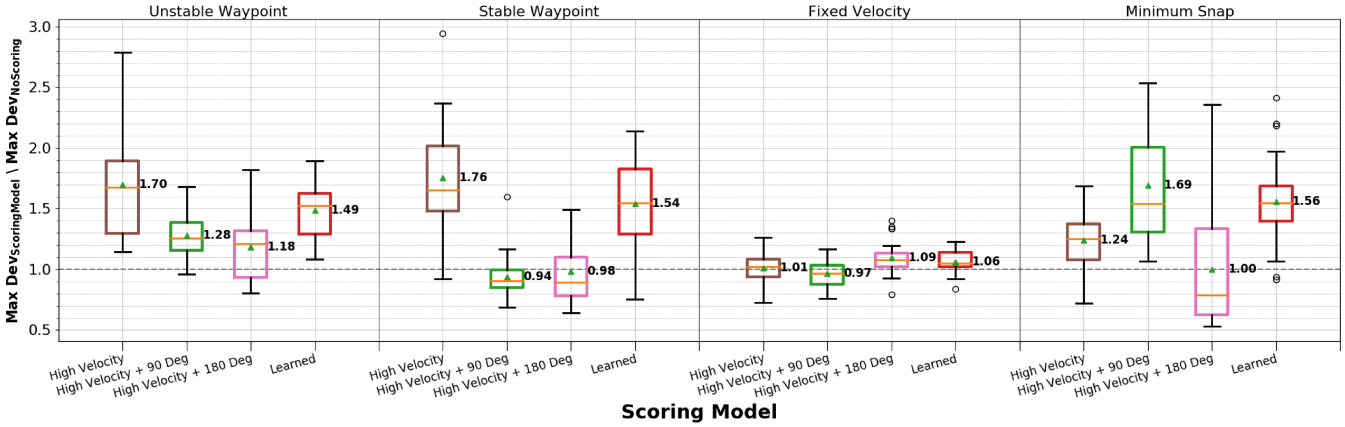


Figure 8: The ratio of maximum deviation with a scoring model to maximum deviation without one. Here the initial trajectory set with no scoring model would have a mean value of 1. Any trajectory set that produced more stress than the initial trajectory set would have values greater than 1. The medians (central line) and mean (triangle and number) are shown.

we executed those initial set of random trajectories in the Parrot-Sphinx[51] simulator. We bound the initial generation to 2 hours, a beamwidth of 5, and a trajectory length of 10, and the learning was meant to generate a model that increases the maximum deviation in a trajectory. We then used the learned scoring model to generate stress-inducing trajectories using a total time of 1 hour, a beamwidth of 5, and a trajectory length of 10.

Figure 9 shows the findings in the form of a boxplot. The first pair of boxes show the results from the execution of trajectories in simulation, while the second pair of boxes show the results from executing the drone in the real world. Each pair represents the deviation of the initial trajectory set and the stress-inducing trajectory set, respectively. Each box is normalized by the mean maximum deviation of the corresponding initial trajectory set. As mentioned earlier, the initial trajectory set was generated without a scoring model, and it shows similar means and variation in simulation and in the real world.

As shown by the second box, the scoring model learned in simulation allows our approach to generate trajectories that, when executed in simulation, cause on average a 26% increase maximum deviation in a trajectory. More interesting, however, is that when the same generated trajectories are executed in the real-world, they also cause a similar degree of additional deviations, albeit with greater variation (whiskers of the fourth box) introduced by external environmental factors such as GPS-localization noise and wind. This confirms that it is possible to mitigate the cost of learning a scoring model through simulation and apply trajectories generated with that model in real-world contexts.

From a testing point of view, one might be interested in trajectories (test inputs) that violated certain specifications. For example, might specify that the maximum deviation from the expected trajectory cannot exceed some threshold. Figure 11 shows the percentage of automatically generated tests that violate a given maximum distance specification. The results indicate that, regardless of the specified maximum deviation, using a scoring model produces a larger percentage of tests that violate the specification. For example, given a specified maximum deviation of 4m, Figure 11 shows

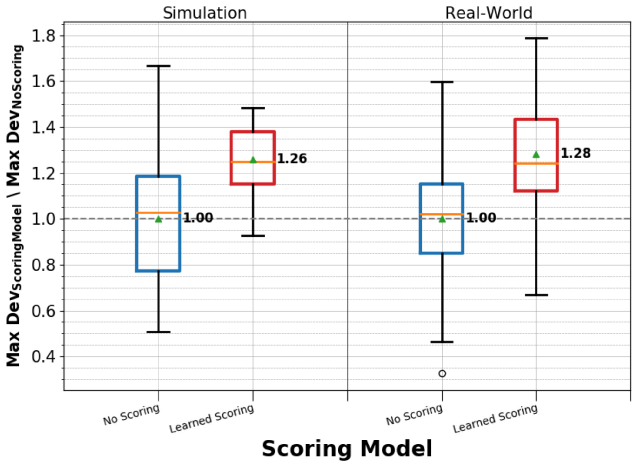


Figure 9: Maximum deviation for simulation and outdoors trajectories normalized by the mean of the trajectory set with no scoring model.

that with no scoring model, only 30% of tests generated would violate that constraint. However, our approach using a scoring model would generate a test set with approximately 70% of tests that violate the same specification. Additionally, these results show that using a scoring model not only generates a higher percentage of tests that violate the constraints, but also generates the test with the largest maximum deviation. The maximum deviation we observed outdoors with the generated trajectories was 6.2m, with the average being 4.5m. This highlights how the approach can generate stressful trajectories that push the drone to deviations that go way beyond the expected deviation for this kind of drone.

Developers can also use these trajectories to further investigate the behaviors which led to these violations. For example, using the Anafi quadrotor, we plotted the test that produced the largest maximum deviation. Figure 10 shows the generated test trajectory

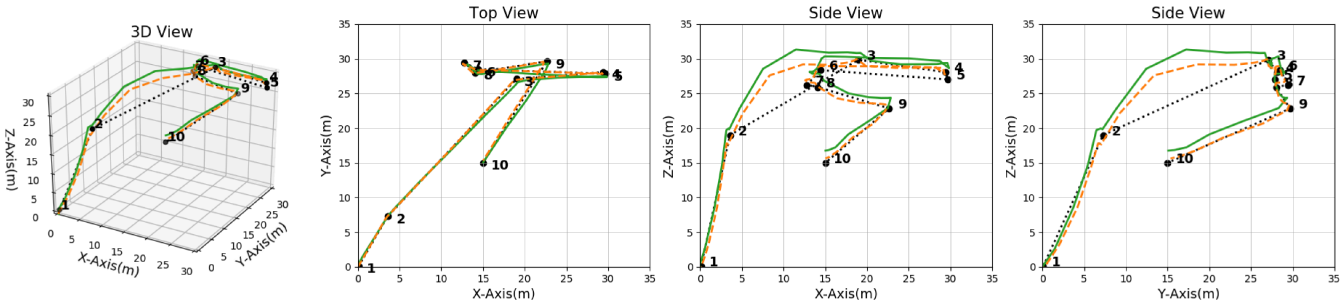


Figure 10: Anafi's position in the real-world and simulation as it traverses one of the stress-inducing trajectories. The expected behaviour is marked as dots. The simulated data is marked with dashes. The real-world data is a solid line.

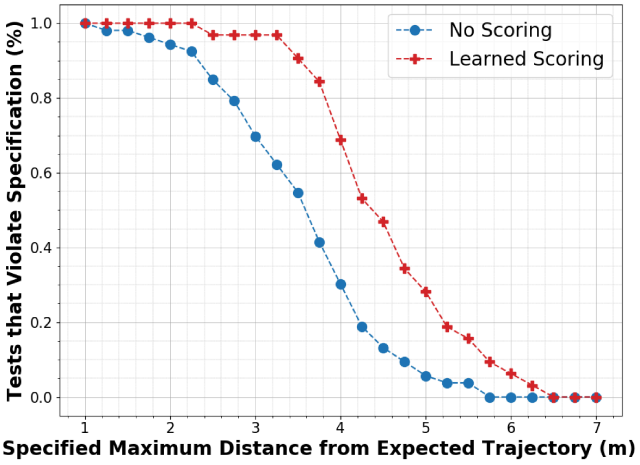


Figure 11: The percentage of outdoor tests which violated the specified maximum deviation

(dotted line) of the drone in both simulation (dashed line) and real-world (solid line). From the top view, it appears that the Anafi follows the expected trajectory precisely. However, from a side view, it seems like the Anafi follows the expected trajectory in all cases except when large changes in all x,y, and z-directions are requested, for instance, when flying from waypoint 2 to 3. The 3rd waypoint is the position (19.0, 27.0, 29.9). In simulation, although it did not follow the expected short line trajectory, it flew to a height of 29.9m as expected. In the real world, the Anafi similarly did not follow the expected trajectory, however it flew to a height of ≈31.34m high, 1.34m over the designated flying altitude of 30m, even though all waypoints are within the flying volume. A pilot flying this quadrotor who was not aware of the distinct behavior shown through this trajectory would at best be surprised and, at worst, experience a collision.

7 CONCLUSION

We have introduced a novel approach for the automatic generation of feasible and stressful trajectories for mobile robots. The approach is unique in that it integrates the kinematics and dynamics of the robot to generate trajectories that are feasible given its physical limitations, it builds on algorithms from robotics planning and graph exploration to more efficiently search the input space, and it incorporates a highly parameterizable scoring model to guide trajectory generation towards those that induce high-stress in the system. The approach was able to generate valid trajectories that caused a mean increase of maximum deviations of 55.9% and 41.3% in the two systems we studied. These deviations are significant as in the commercial quadrotor, the maximum deviation recorded in a small 30m^3 area was 6m.

In future work, we will investigate reduction techniques that let us explore the trajectory space more efficiently by, for example, exploiting the physical commutativity of sub-trajectories. Another exciting avenue that we intend to explore is combining our trajectory generation technique with languages used to specify environments[16]. Using this, we hypothesize that we could generate not only trajectories but entire environments that stress any given robot. These environments could be combined with a mix-mode execution where, for example, the drone flies in the real world, but the obstacles are present only in simulation.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation Awards #CCF-1853374 and #CCF-1901769 as well as the U.S. Army Research Office Grant #W911NF-19-1-0054. We thank the developers of FlightGoggles[23] for making their systems and software available.

REFERENCES

[1] Matthias Althoff and Sebastian Lutz. 2018. Automatic Generation of Safety-Critical Test Scenarios for Collision Avoidance of Road Vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1326–1333.
[2] Joseph Stiles Beggs. 1983. *Kinematics*. CRC Press.
[3] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2016. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 63–74.
[4] Robert Binder. 2000. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional. 745–746 pages.
[5] Brian Garrett-Glaser. 2019. Avionics - Drone Delivery Crash in Switzerland Raises Safety Concerns As UPS Forms Subsidiary. https://www.aviationtoday.com/2019/08/08/drone-delivery-crash-in-switzerland-raises-safety-concerns/. [Online; accessed 5-November-2019].
[6] Mo Chen, Sylvia Herbert, and Claire J Tomlin. 2016. Fast reachable set approximations via state decoupling disturbances. In *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE, 191–196.
[7] Gregory S Chirikjian. 1996. Synthesis of Discretely Actuated Manipulator Workspaces via Harmonic Analysis. In *Recent Advances in Robot Kinematics*. Springer, 169–178.

[8] Anders Lyhne Christensen, Rehan O'Grady, Mauro Birattari, and Marco Dorigo. 2008. Fault detection in autonomous robots based on fault injection and learning. *Autonomous Robots* 24, 1 (2008), 49–67.

[9] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An open urban driving simulator. *arXiv preprint arXiv:1711.03938* (2017).

[10] Tommaso Dreossi, Alexandre Donzé, and Sanjit A. Seshia. 2017. Compositional Falsification of Cyber-Physical Systems with Machine Learning Components. arXiv:1703.00978 [cs.SY]

[11] Tommaso Dreossi, Shromona Ghosh, Alberto Sangiovanni-Vincentelli, and Sanjit A. Seshia. 2017. Systematic Testing of Convolutional Neural Networks for Autonomous Driving. arXiv:1708.03309 [cs.CV]

[12] Homer D Eckhardt. 1998. *Kinematic design of machines and mechanisms*. McGraw-Hill New York.

[13] Unity Game Engine. 2008. Unity game engine-official site. *Online][Cited: October 9, 2008.] http://unity3d. com* (2008), 1534–4320.

[14] Michelle Chaka Eric Thorn, Shawn Kimmel. 2018. A Framework for Automated Driving System Testable Cases and Scenarios. https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13882-automateddrivingsystems_092618_v1a_tag.pdf.

[15] Artur Filipowicz, Jeremiah Liu, and Alain Kornhauser. 2017. *Learning to recognize distance to stop signs using the virtual world of Grand Theft Auto 5*. Technical Report.

[16] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. 2019. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 63–78.

[17] Daniel J Fremont, Xiangyu Yue, Tommaso Dreossi, Shromona Ghosh, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. 2018. Scenic: Language-based scene generation. *arXiv preprint arXiv:1809.09310* (2018).

[18] Alessio Gambi, Tri Huynh, and Gordon Fraser. 2019. Generating effective test cases for self-driving cars from police reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 257–267.

[19] Alessio Gambi, Marc Mueller, and Gordon Fraser. 2019. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 318–328.

[20] Maxime Gautier. 1986. Identification of robots dynamics. *IFAC Proceedings Volumes* 19, 14 (1986), 125–130.

[21] Aurélien Géron. 2017. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.".

[22] Antoine Girard and Colas Le Guernic. 2008. Efficient reachability analysis for linear systems using support functions. *IFAC Proceedings Volumes* 41, 2 (2008), 8966–8971.

[23] Winter Guerra, Ezra Tal, Varun Murali, Gilhyun Ryou, and Sertac Karaman. 2019. FlightGoggles: Photorealistic Sensor Simulation for Perception-driven Robotics using Photogrammetry and Virtual Reality. *arXiv preprint arXiv:1905.11377* (2019).

[24] WuLing Huang, Kunfeng Wang, Yisheng Lv, and FengHua Zhu. 2016. Autonomous vehicles testing methods review. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 163–168.

[25] Inseok Hwang, Dušan M Stipanović, and Claire J Tomlin. 2005. Polytopic approximations of reachable sets applied to linear dynamic games and a class of nonlinear systems. In *Advances in control, communication networks, and transportation systems*. Springer, 3–19.

[26] Jackie Wattles. 2019. CNN Business - Tesla on Autopilot crashed when the driver's hands were not detected on the wheel. https://www.cnn.com/2019/05/16/cars/tesla-autopilot-crash/index.html. [Online; accessed 5-November-2019].

[27] Reza N Jazar. 2010. *Theory of applied robotics: kinematics, dynamics, and control*. Springer Science & Business Media.

[28] Matthew Johnson-Roberson, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, Karl Rosaen, and Ram Vasudevan. 2016. Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks? *arXiv preprint arXiv:1610.01983* (2016).

[29] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation* 12, 4 (1996), 566–580.

[30] BaekGyu Kim, Akshay Jarandikar, Jonathan Shum, Shinichi Shiraishi, and Masahiro Yamaura. 2016. The SMT-based automatic road network generation in vehicle simulation environment. In *2016 International Conference on Embedded Software (EMSOFT)*. IEEE, 1–10.

[31] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. RVFUZZER: finding input validation bugs in robotic vehicles through control-guided testing. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 425–442.

[32] Philip Koopman and Michael Wagner. 2016. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety* 4, 1 (2016), 15–24.

[33] Kurt Barnhart. 2015. Partners Kansas State University Salina and Westar Energy build one of the largest enclosed flight facilities for UAS in the nation. https://www.k-state.edu/media/newsreleases/oct15/pavilion101415.html. [Online; accessed 22-August-2019].

[34] Alex A Kurzhanskiy and Pravin Varaiya. 2006. Ellipsoidal toolbox (ET). In *Proceedings of the 45th IEEE Conference on Decision and Control*. IEEE, 1498–1503.

[35] Xuehua Liao, Zhousen Zhu, Yusong Yan, and Tao Lv. 2012. Traffic accident reconstruction technology research and simulation realization. In *2012 IEEE Symposium on Electrical & Electronics Engineering (EEESYM)*. IEEE, 152–155.

[36] J. Liebelt and C. Schmid. 2010. Multi-view object class detection with a 3D geometric model. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 1688–1695.

[37] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. 2011. Automatic track generation for high-end racing games using evolutionary computation. *IEEE Transactions on computational intelligence and AI in games* 3, 3 (2011), 245–259.

[38] Massachusetts Institute of Technology. 2019. Flight Goggles. https://flightgoggles.mit.edu. [Online; accessed 01-January-2020].

[39] Mathworks. 2020. Matlab and Python. https://www.mathworks.com/products/matlab/matlab-and-python.html. [Online; accessed 26-January-2020].

[40] Reza Matinnejad, Shiva Nejati, Lionel Briand, Thomas Bruckmann, and Claude Poull. 2013. Automated model-in-the-loop testing of continuous controllers using search. In *International Symposium on Search Based Software Engineering*. Springer, 141–157.

[41] Mitch McCaffrey. 2017. *Unreal Engine VR Cookbook: Developing Virtual Reality with UE4*. Addison-Wesley Professional. Chater 7: Character Inverse Kinematics.

[42] Daniel Mellinger and Vijay Kumar. 2011. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*. IEEE, 2520–2525.

[43] microdrones. 2019. Microdrones Inspection Service. https://www.microdrones.com/. [Online; accessed 5-November-2019].

[44] Ian M Mitchell, Alexandre M Bayen, and Claire J Tomlin. 2005. A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. *IEEE Transactions on automatic control* 50, 7 (2005), 947–957.

[45] Galen E Mullins, Paul G Stankiewicz, and Satyandra K Gupta. 2017. Automated generation of diverse and challenging scenarios for test and evaluation of autonomous vehicles. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 1443–1450.

[46] Nicole Casal Moore. 2019. M-Air autonomous aerial vehicle outdoor lab opens. https://news.umich.edu/m-air-autonomous-aerial-vehicle-outdoor-lab-opens/. [Online; accessed 22-August-2019].

[47] Matthew O'Kelly, Aman Sinha, Hongseok Namkoong, John Duchi, and Russ Tedrake. 2018. Scalable End-to-End Autonomous Vehicle Testing via Rare-event Simulation. arXiv:1811.00145 [cs.LG]

[48] Parrot. 2019. Anafi. https://www.parrot.com/us/drones/anafi. [Online; accessed 11-November-2019].

[49] Parrot. 2019. Bebop 2. https://beamng.gmbh/research/. [Online; accessed 26-January-2020].

[50] Parrot. 2019. Olympe Documentation. https://developer.parrot.com/docs/olympe/. [Online; accessed 20-November-2019].

[51] Parrot. 2019. Parrot-Sphinx. https://developer.parrot.com/docs/sphinx/whatissphinx.html. [Online; accessed 22-August-2019].

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[53] Francisca Rosique, Pedro Navarro Lorente, Carlos Fernandez, and Antonio Padilla. 2019. A Systematic Review of Perception System and Simulators for Autonomous Vehicles Research. *Sensors* 19 (02 2019), 648. https://doi.org/10.3390/s19030648

[54] Atrisha Sarkar and Krzysztof Czarnecki. 2019. A behavior driven approach for sampling rare event situations for autonomous vehicles. *CoRR* abs/1903.01539 (2019). arXiv:1903.01539 http://arxiv.org/abs/1903.01539

[55] Thomas Sayre-McCord, Winter Guerra, Amado Antonini, Jasper Arneberg, Austin Brown, Guilherme Cavalheiro, Yajun Fang, Alex Gorodetsky, Dave McCoy, Sebastian Quilter, et al. 2018. Visual-inertial navigation algorithm development using photorealistic camera simulation in the loop. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2566–2573.

[56] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. 2018. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*. Springer, 621–635.

[57] Thierry Sotiropoulos, Jérémie Guiochet, Félix Ingrand, and Hélene Waeselynck. 2016. Virtual worlds for testing robot navigation: a study on the difficulty level. In *2016 12th European Dependable Computing Conference (EDCC)*. IEEE, 153–160.

[58] Dimitar Stanev and Konstantinos Moustakas. 2019. Modeling musculoskeletal kinematic and dynamic redundancy using null space projection. *PloS one* 14, 1 (2019).

[59] Stanford Artificial Intelligence Laboratory et al. [n.d.]. *Robotic Operating System*. https://www.ros.org

[60] Jan Erik Stellet, Marc René Zofka, Jan Schumacher, Thomas Schamm, Frank Niewels, and J Marius Zöllner. 2015. Testing of advanced driver assistance towards automated driving: A survey and taxonomy on existing approaches and open questions. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*. IEEE, 1455–1462.

[61] TASS International. 2019. PreScan - A Simulation and Verification Environment for Intelligent Vehicle Systems. https://tass.plm.automation.siemens.com/prescan. [Online; accessed 22-August-2019].

[62] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[63] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski. 2018. Simulation-based Adversarial Test Generation for Autonomous Vehicles with Machine Learning Components. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. 1555–1562.

[64] Cumhur Erkan Tuncali, Theodore P Pavlic, and Georgios Fainekos. 2016. Utilizing S-TaLiRo as an automatic test generation framework for autonomous vehicles. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 1470–1475.

[65] University of Michigan. 2020. Mcity. https://mcity.umich.edu. [Online; accessed 26-January-2020].

[66] Kimon P Valavanis and George J Vachtsevanos. 2015. *Handbook of unmanned aerial vehicles*. Springer.

[67] David Vazquez, Antonio M Lopez, Javier Marin, Daniel Ponsa, and David Geronimo. 2013. Virtual and real world adaptation for pedestrian detection. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2013), 797–809.

[68] Abraham P Vinod, Baisravan HomChaudhuri, and Meeko MK Oishi. 2017. Forward stochastic reachability analysis for uncontrolled linear systems using fourier transforms. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*. ACM, 35–44.

[69] Virginia Tech. 2018. Virginia Tech Drone Park officially open . https://vtnews.vt.edu/articles/2018/04/ictas-droneparkopens.html. [Online; accessed 22-August-2019].

[70] Waymo LLC. 2019. Waymo - Self Driving Car. https://waymo.com. [Online; accessed 5-November-2019].

[71] Waymo Team. 2018. Where the next 10 million miles will take us. https://medium.com/waymo/where-the-next-10-million-miles-will-take-us-de51bebb67d3. [Online; accessed 1-December-2019].

[72] Edmund Taylor Whittaker. 1988. *A treatise on the analytical dynamics of particles and rigid bodies*. Cambridge university press.

[73] Thomas Wallace Wright. 1898. *Elements of mechanics including kinematics, kinetics and statics, with applications*. D. Van Nostrand Company.

[74] Yaroslav S. Yatskiv. 2007. Kinematics and Physics of Celestial Bodies. https://www.springer.com/journal/11963. [ISSN: 0884-5913].

[75] Esen Yel, Tony X Lin, and Nicola Bezzo. 2017. Reachability-based self-triggered scheduling and replanning of uav operations. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 221–228.

[76] Esen Yel, Tony X Lin, and Nicola Bezzo. 2018. Self-triggered adaptive planning and scheduling of uav operations. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 7518–7524.

[77] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 132–142.

[78] Zhi Quan Zhou and Liqun Sun. 2019. Metamorphic testing of driverless cars. *Commun. ACM* 62 (03 2019), 61–67. https://doi.org/10.1145/3241979