AFRL-RI-RS-TR-2022-071

ASSURED AUTONOMY USING DYNAMIC MONITORS AND SIMULATION (ADAM DMS)

DYNAMIC OBJECT LANGUAGE LABS, INC.

MAY 2022

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by AFRL Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2022-071 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN L. DRAGER
Work Unit Manager

/ S /
GREGORY J. HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
MAY 2022	FINAL TECHNICAL REPORT	MAY 2019	SEPTEMBER 2021

4. TITLE AND SUBTITLE

ASSURED AUTONOMY USING DYNAMIC MONITORS AND SIMULATION (ADAM DMS)

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8750-19-C-0088	N/A	62303E
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**
		R2S1

6. AUTHOR(S)

Paul Robertson and Prakash Manghwani

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Dynamic Object Language Labs Inc.
2 Parsonage Hill Rd.
Haverhill MA 01832

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIT
525 Brooks Road
Rome NY 13441-4505

DARPA
675 North Randolph St.
Arlington VA 22203-2114

10. SPONSOR/MONITOR'S ACRONYM(S)

RI

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

AFRL-RI-RS-TR-2022-071

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# AFRL-2022-2301
Date Cleared: 16 MAY 2022

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Systems that are forced out of the certified operating configuration can, and do, fail catastrophically. In many cases, the systems can be saved by executing often counter intuitive actions. With a complex system, it is unlikely that a human operator would be able to "discover" these recovery actions before the system is destroyed. Using a high precision simulator, machine learning enables the research to learn action policies that can save a system from catastrophic failure, if it is caught quickly enough. The Assured Autonomy using Dynamic Monitors and Simulation (ADAM DMS) system learns the safe operating configuration and deep learning allows synthesis of a monitor and the learned recovery policy can be synthesized to perform the recovery. A simulated quadcopter was used as a demonstration example.

15. SUBJECT TERMS

ADAM DMS, Cyber-Physical System (CPS), Learning Enables Components (LECs)

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
U	U	U	SAR	38

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
STEVEN L. DRAGER	N/A

PREVIOUS EDITION IS OBSOLETE.

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Table of Contents

List of Figures

List of Tables

1. SUMMARY

No Cyber-Physical System (CPS) can be guaranteed to never fail. It can be guaranteed to do what it was supposed to do if no assumptions are violated, but there will always be assumptions that can be violated. What do we do when assumptions have been violated and the system manages to get into a state, through no fault of its own, that could be catastrophic? What can we know about extreme vulnerabilities that were not expected in the system design? Can we make Learning Enables Components (LECs) that will allow us to monitor for excursions from safe operation (MONITOR 3.2) and can we learn to intervene in order to recover from some of these excursions (RECOVERY CONTROLLER 3.4)

There is a strong need for a run-time capability to handle the cases that violate the assumptions underlying verified systems, and that fall outside the tested and validated cases. We proposed to build a run-time monitoring approach capable of detecting and immediately reacting to surprise, as well as diagnosing and learning from surprise.

Our approach has been to use extensive simulation testing to learn where the vulnerabilities lie so that we can learn from simulation data, and enact MONITORS for boundaries to safe behavior.

Given a machine learning based autonomous CPS that has been rigorously verified and validated (at least in simulation) via extensive testing, the primary source of surprise arises from the system attempting to operate in a context for which it was not adequately trained. For that reason, we focus on monitoring and learning the contexts in which the CPS operates.

It is a well acknowledged problem with neural networks, that have been trained within a context to very accurately classify their inputs with low false positives and negatives within that context, are capable of making classifications with high confidence that are very bad, in the sense of not even being near misses. Such a system will perform very badly outside of the context within which it was trained. One approach is to attempt to see if the inputs are similar to the data upon which it was trained. But we did not propose to solve that problem, rather we assumed that contexts would exist for which the LECs of CPSs would fail and to be able to detect such critical cases and to use machine learning methods to try to get the system back into a stable state where the LEC based CPS can continue to safely operate.

Simulation environments are increasingly realistic, and therefore, can effectively validate a system consisting of electro-mechanical elements, where the scope of operation is limited and well defined. However, an autonomous system operating over an extended period in an unknown environment has a very wide scope of operation and requires significant decision-making complexity to operate successfully. Brute force simulation of the type used to validate simple electro-mechanical systems is not applicable to the more complex autonomous systems; the combinatorics require astronomically large numbers of simulation tests.

Our goal, therefore, was to develop a validation system that can be used for realistic systems but is feasible in terms of computation.

In summary, our objective was to explore how high-fidelity simulators could be harnessed to synthesize run-time monitors that detect deviations from safe operation and further to apply machine learning to synthesize controllers that could intervene in order to recover from, at least some, of the deviations. Some dangerous deviations from safe operation are not recoverable, but many are. Examples of recoverable dangerous situations are well known, in some cases, we have learned recovery techniques and they are taught to human controllers. Pilots are taught how to recover from a stall or a spin, for example, but if they occur too close to the ground, none of these techniques will work. A car that hits a patch of ice and gets into a skid, similarly, can be recovered if there is enough space before going off road or hitting another vehicle. For complex modern CPSs, the failure modes can be more complicated than the simple examples cited above.

Our system, "Assured Autonomy using Dynamic Monitors and Simulation" (ADAM DMS), is a research platform consisting of a collection of capabilities that realize the objectives described above and consist of the testbed and algorithms that support the clustering, machine learning, and risk-aware planning. Whereas we have used a single test-bed for our research, it has been our intention that none of what we do should depend in any way on the CPS being a quad-copter, and that the principles should be applicable to nuclear reactors, or other complex CPSs.

2. INTRODUCTION

We set out to use an unsupervised (clustering 3.3) approach to learning contexts of operation of the target CPS system for offline context learning. By using continuous re-clustering, novel contexts to existing context sets can be learned based on the detected response of the CPS to the context.

It turns out that that also gives us a performance advantage as will be described in the section on clustering. The ability allows us to separate out new clusters that represent dangerous situations. These data points can be separated out from the benign cases, and allow a deep learning network to learn the run-time monitors that efficiently watch out for unsafe areas of the state space.

We utilize high-fidelity simulations in order to discover vulnerabilities that are likely to have correlates in the physical system. We cannot afford to crash thousands of jet fighters or melt down large nuclear reactors to learn, so simulation seemed to be a viable approach. There remains, however, the weakness that if the vulnerabilities in the physical system are completely different from the simulation, and that no simulator can ever be good enough, how will we know? We formulated a response to that question as a transfer learning problem, and towards the end of our research period, we were attempting to demonstrate that, but we ran out of time before being able to solve that last goal. We will say more about that towards the end of the report.

We had proposed to use the recently developed machine learning techniques based on Deep Reinforce- ment Learning, but the clustering approach gave us good results and we continued with that. There are benefits to both approaches, but we didn't have time to explore and compare the two.

We use explicit risk awareness and risk management approaches for the recovery controller. The intuition here is clear. A cluster-based reinforcement learner can learn policies for recovery from dangerous situations but being able to reason about acceptable risk would add so much complexity to the definitionof the reinforcement learning problem that it would converge very slowly, if at all. What we did instead, was to allow the reinforcement learner to learn without the complexity of risk so that it could converge relatively quickly and then to characterize the resulting policy in terms of variables that we wish to reason over and then to make choices at run-time that are biased by risk constraints. An example helps to make the issues clear, and also serves to introduce our testbed.

Having used a quadcopter (QC) in a previous autonomous operation project, and having both the physical quadcopters and a working control interface, produced a good appreciation of how they can fail. The built-in controller allows for stable hovering, movement from one location to another way point, and the ability to joystick control it without having to worry about stability. Whereas the built-in controller is robust to minor disturbances, a serious disturbance can come from, for example, flying past a building and encountering a wind gust that hits the Unmanned air vehicle (UAV) asymmetrically and thus flipping it into an orientation from which its built-in controller fails to recover. Even at a great height, the built-in controller fails to restore stable flight with disastrous consequences. A highly skilled human operator, however, running in manual mode, in which the pilot is solely responsible for stability, can save the quadcopter from crashing, in many cases, and in general is able to perform the kind
of aerobatics that the built-in control is incapable of.

While the quadcopter is in an orientation more or less perpendicular to the ground, it is falling like a brick and recovery must occur very rapidly in order to save the vehicle. Even if stable level flight is reestablished, it can still be falling with such speed that there is not enough time to avoid a collision with the ground.

Consider the situation in which the quad copter is almost flipped over and is initially thrusting itself towards the ground even faster than gravity would do alone. We can learn to operate the rotors so that the copter reestablishes level flight. To avoid hitting the ground, we raise the quadcopter to an altitude that will allow for recovery after trial and error. This is a helpful strategy with reinforcement learning, because it allows some unsuccessful attempts before hitting upon a good solution. The good solution updates the policy and later a good solution will be found sooner, and eventually, the recovery time decreases.

One form of stability is found when the quad copter is fully inverted, and the rotors are going in the reverse direction to normal. For copter that is almost inverted anyway, achieving stable inverted flight is much faster than rotating back to upright flight. Our learner is able to find both of these solutions and the different solutions have recovery times and lost altitude as part of what is learned. The learned policy will have entries for multiple paths to stability but some of those paths will fail if there is not enough altitude. Now, we can evaluate, what decision to make given a policy that has multiple successful paths. The built-in controller requires upright flight. If we pass control back to the QC when it is inverted, it will crash very quickly. If there is not enough altitude to do a full rotation to upright orientation, the controller might try and crash anyway. With risk-aware planning, we can adjust the equality of choices to consider the probability of success given the observed sensors, one of which is recording height above ground. This reformulation prefers the choice of stable inverted flight followed by stable inverted gain in altitude, until the probability of doing a 180-degree rotation is high, at which point it can perform the rotation to stable upright flight without crashing.

2.1 ADAM-DMS Testbed Simulators

We built our quad-copter simulator using off the shelf and open-source components. For Autopilot features, we adopted ArduPilot (https://ardupilot.org/) and a Simple Quadcopter Simulator which can be found at https://github.com/dollabs/Quadcopter_simulator.git. The ARDU pilot runs within Gazebo and is a high-fidelity simulation of the physical quadcopter that responds to winds and other disturbances. The simple simulator represents a similar quadcopter but is a crude simulation that has no support for wind disturbances, but which can be run many times real-time, and is thus useful for learning where a large number of episodes are necessary in order to learn good policies.

ArduPilot is quite advanced, it is designed to operate in real-time and for flying real vehicles. It is this advanced nature of ArduPilot that made it difficult to speed up simulations with ArduPilot. To speed up our simulations and reinforcement learning, we adopted the 'Simple QC Sim '; see Figure 2.

2.1.1 The Ardu Gazebo simulator

ArduPilot is autopilot control software and interfaces with other components such as Ground Station, Planners etc. via the MavLink protocol. For our simulation, we adopted ArduPilot Software In The Loop (SITL) implementation that interfaced with GazeboSim and the provided Robot Operating System (ROS) based interface named Mavros (which implements the MavLink interface). We call the collection of these components the Ardu Gazebo Simulator as shown in Figure 1. It has a valuable property, that it can be used to control our physical quadcopter which makes it easy to test that our learned controllers and monitors function correctly with the physical hardware.

2.1.2 The Simple Quadcopter Simulator

This simulator is a very simple implementation of a quad-copter, developed in Python and designed for headless operation with support for time scaling which allows the simulator to run as fast as the processor can support. The implementation itself is comprised of few classes, namely Propeller, Quadcopter and Controller. The propeller class defines the thrust generated by a propeller at a given speed of rotation and specified size. The implementation of this class is based on equations as provided here: http://www.electricrcaircraftguy.com/2013/09/propeller-static-dynamic-thrust-equation.html.

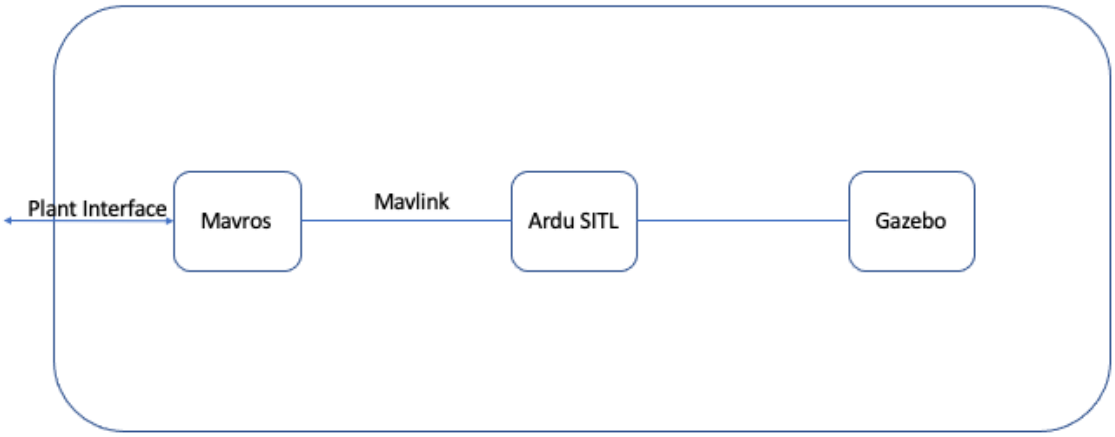

Figure 1 Ardu Gazebo Simulator

The quad-copter class performs the simulations of the dynamics based on the state space solution of a quad-copter. The state space representation of a quad-copter model have been adapted from **Quadcopter Dynamics, Simulation, and Control by Andrew Gibiansky and Quadrotor Dynamics and Control by Randal Beard**. The state space is defined as

```
X = [x,y,z,x_dot,y_dot,z_dot,theta,phi,gamma, theta_dot, phi_dot, gamma_dot]
```

and state update over a period of time dt is provided by vode, the Ordinary Differential Equation (ODE) solver from the SciPy library. The controller class is intended to provide point to point control to move the QC to a desired location in 3D space. It also has an optional Graphical User Interface (GUI) which we instantiate in a separate process and integrate with the sim via message passing.

2.1.3 Plant Interface with Sim(s)

Our reinforcement learning and other components work in conjunction with both simulators by means of a 'plant interface'.

To integrate both sims, we developed a MAVROS plant interface to communicate with Ardu Gazebo Simulator and a simple python module that implements the plant-interface to bind the Simple QC sim with our components, see Figure 3.

In this way we can use the two simulators interchangeably which gives us the opportunity to choose between speed and accuracy. We had hoped that we could use this to speedup learning by starting out with the crude simulator to get to an initial level of flying competence and then switch to the Ardu simulator to improve accuracy of the learned policy. This is described later under transfer learning. Unfortunately, our period of performance ended before we could achieve this stretch goal.

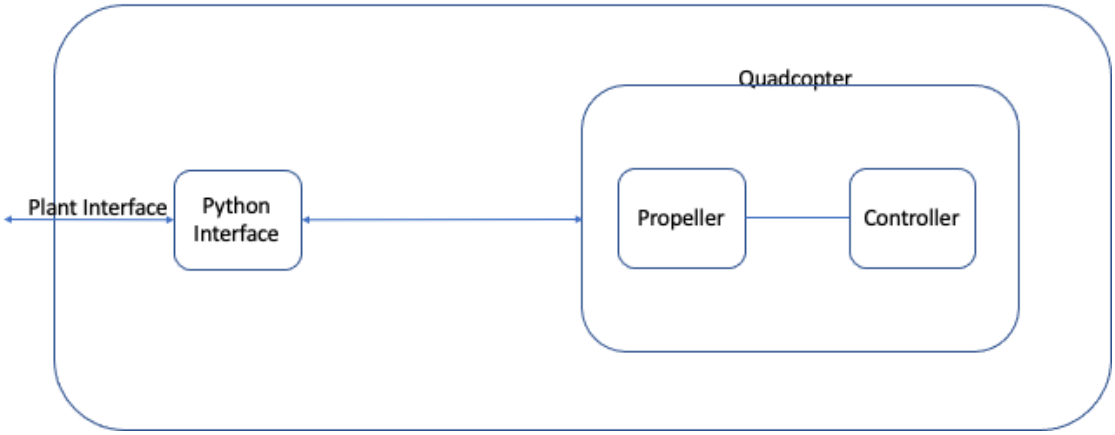

Figure 2 Simple QC Simulator

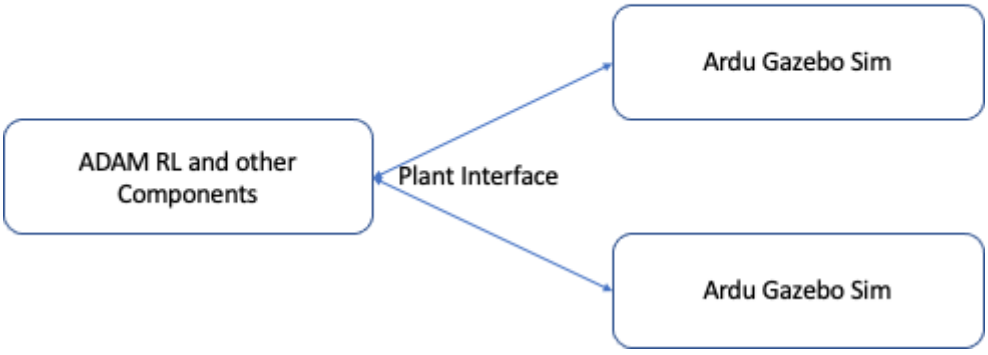

Figure 3 Sim Connections

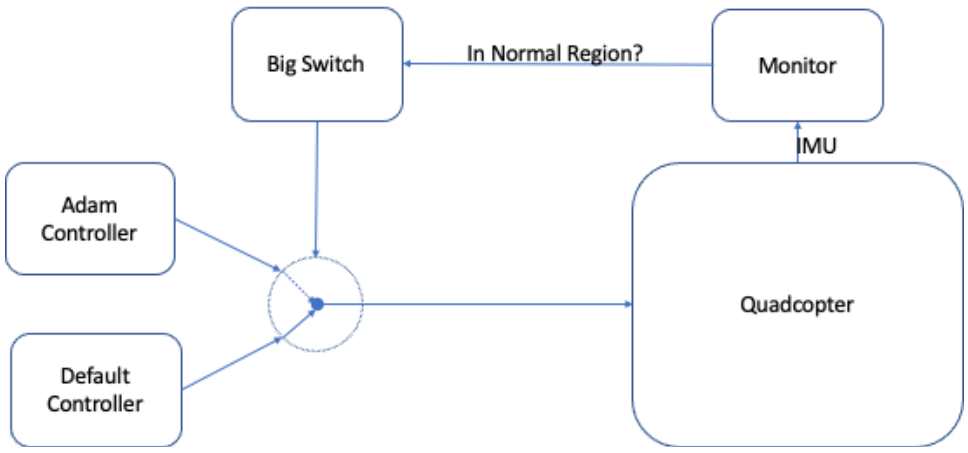

Figure 4 High Level Architecture

3. METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Architectural Overview

The ADAM-DMS enabled QC is shown in Figure 4, of the Adam Learned Controller, the Ardu built-in Controller, the Monitor and the Big Switch. In this section, we give a brief overview of the ADAM components and their role and interaction with other components.

In an ADAM-DMS enabled system, we track two mutually exclusive regions of operation. The normal operating region when the QC is under its default (Ardu Controller in our case) controller and the out-of-normal regions when the QC is controlled by the Adam Learned Controller. Whenever the QC slips out of the normal region, in the event of unexpected situations, and its default controller is unable to bring it back within normal operating specs, we transfer control to the ADAM-DMS controller which then is responsible for stabilizing the QC to a point where it is back within its normal operating region. After the QC's state has switched from abnormal region to normal region, we hand back the control to QC's default controller.

Figure 4 gives a high-level diagram of the ADAM components as used in the quad-copter test-bed. We envision the same architecture working with little or no change with other, dissimilar, CPSs.

3.1.1 The Monitor

The monitor receives select sensor data from the CPS, for example, in the case of the quadcopter, the Inertial Movement Unit (IMU) data, but in general it could be any of the sensor data that is available, some subset of which the learned monitor uses.

3.1.2 The Big Switch

Big switch, as shown in Figure 4, is responsible for observing operating region of the quadcopter and handing over control to either the ADAM-DMS Controller or the built-in (Ardu) Controller. It receives input from the monitor that indicates whether the CPS is in its safe operating region, which should usually be the case, but when the monitor detects that the CPS is no longer operating with its safe region of the state space, it causes the big switch to take control from the built-in controller and gives control to the ADAM-DMS controller. There are numerous complexities to the big switch in terms of how control can be switched without suffering transition effects. We will not discuss that further in this report because it is part of establishing an ADAM-DMS wrapper for a new CPS. The simplistic view of how it works is that actuation commands issued by the deselected controller are simply ignored, the problem comes when the controller doesn't know that its commands are being ignored and yet it sees sensor data that indicates the results, not of its own actions, but of the other controller. The ADAM-DMS controller is under our control, so we can signal that it has been deselected, but the built-in controllers are assumed to be black boxes and thus some effort needs to be put into avoiding sudden unwanted violent control actions when it suddenly regains control.

3.2 Monitor

3.2.1 ADAM Learned Runtime Monitor

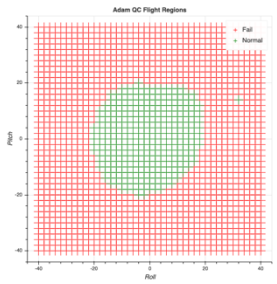

Figure 5a: Wind 0

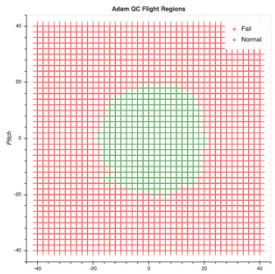

Figure 5b: Wind 1

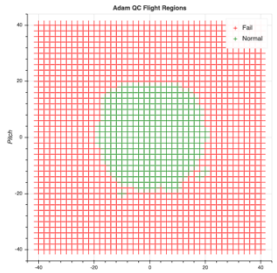

Figure 5c: Wind 2

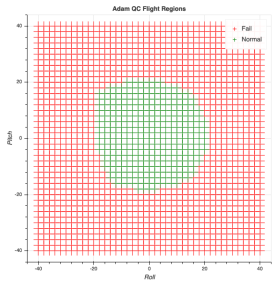

Figure 5d: Wind 3

Figure 5: Safe and Unsafe regions under different wind conditions

Learning the run-time monitor is a two-step process with some engineering decisions involved in deciding how to explore the state space.

The first stage involves forcing the simulator into a point in the state space so that we can see what happens when we let the built-in controller control the CPS in that established state. We give the simulator enough time to see if the quadcopter will crash. If it crashes, then that point in the state space is marked as **bad** otherwise it is marked as **good**. The data collected in this way forms the training data used for subsequent learning. It is important to notice the difference between the state space and the sensor data. We can put the device into a very specific state, but the sensors don't tell us state, they tell us sensor readings. In order to be able to recognize the dangerous parts of the state space, in the second phase, we employ a simple deep learning training that learns to recognize bad state from the available sensor data.

In the case of the quadcopter, we collect data for differing wind strengths as follows. We collect IMU data for different wind conditions, ranging from no wind to velocity of 4 m/s. For each wind condition, we programmatically perturb the QC by varying Roll and Pitch values of the QC for a moment and let QC's Ardu Controller try and recover from it. The perturbation can be viewed as some- one taking a hammer and hitting the QC in such a way that it's Roll and Pitch changes to the given value. For each perturbation, we collected IMU data points and the final condition of the QC which was CRASHED or recovered by Ardu Controller.

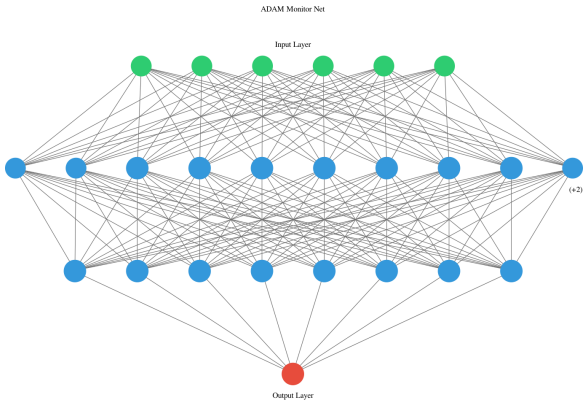

Figure 6: Monitor Neural Network

Figure 5 shows some examples of collected data under different wind conditions. Notice that the wind has little effect on the quadcopter, because the quad copter moves with the wind.

For each wind condition, we varied roll and pitch from -40 to 40 and observed the final state of QC to be crashed or recovered under Ardu Control.

In each of the Figures 5a – 5d involving different wind conditions, the green areas represent points in the state space where the Ardu controller could safely control the device. The red areas represent the pointsin the state space that inevitably lead to a crash. The Ardu controller will keep the QC within the green areas except for exceptional conditions where the quadcopter is perturbed by an outside event.

The monitor that we produce from these data is a learned Neural Network (NN), see Figure 6 that keeps track of normal flight operational envelope based on IMU variables (linear acceleration and angular velocities). We developed a fully connected NN with 6 IMU variables as input and 1 variable as output. Output variable, CRASHED, is a Boolean and value true implies that the inputs indicate the the QC is in a state that will lead to a crash with the current controller. The Monitor will use this signal to switch over to the learned recovery controller.

For each of the wind condition, we split IMU data into learn and eval datasets. We kept the size of each learn data set constant to 10000 out of 179k+ total data points. i.e. We used maximum of 5.5% of the data for learning. When splitting data, we randomly sampled data points while ensuring that distribution of output variable CRASHED is consistent with the complete dataset. Then we combined the learn data set for each wind condition into a single dataset and used it for learning.

We evaluated our learned model with the eval dataset and found our model to have accuracy of 91% or higher. Evaluation results were:

Eval results

```
data file: eval.wind_0_imu.csv
accuracy: 93.19%

data file: eval.wind_1_imu.csv
accuracy: 91.37%

data file: eval.wind_2_imu.csv
accuracy: 92.93%

data file: eval.wind_3_imu.csv
accuracy: 92.28%

data file: eval.wind_4_imu.csv
accuracy: 91.64%
```

Total IMU Data points

```
179572 ../wind_0_imu.csv
203840 ../wind_1_imu.csv
223979 ../wind_2_imu.csv
241794 ../wind_3_imu.csv
200867 ../wind_4_imu.csv
```

Subset of data used for learning

```
10000 learn.wind_0_imu.csv
10000 learn.wind_1_imu.csv
10000 learn.wind_2_imu.csv
10000 learn.wind_3_imu.csv
10000 learn.wind_4_imu.csv
```

Learning sample distribution

```
../../src/imu_analyse.py learn.wind_0_imu.csvCrashed / Not Crashed %: 56.87 43.13

../../src/imu_analyse.py ../wind_0_imu.csv
Crashed / Not Crashed %: 56.86911099726015 43.13088900273985
```

3.3 CLUSTERING

3.3.1 Simultaneous Learning of State Space and Policies

Natural and Cyber-Physical Systems, alike, deal with very large state-spaces, yet animals learn effectivelywith a small number of training episodes. Massive, annotated training corpora are not found in nature, neither is a massive number of episodes that would be infeasible for a physical living agent to perform.

Dimensional reduction and state space limitation is a standard technique, but it requires human involvement to identify the important dimensions. What is important in the state space should be learned as the policies are being learned. There will be parts of the state space that represents things that are happening in the world unrelated to any actions taken and which can safely be ignored, and there will be parts of the state space that are not observable Partially Observable Markov Decision Process (POMDP) not a Markov Decision Process (MDP). A generalized learner must deal with real-world issues in which not all changes are the result of the robot's action.

3.3.2 Prior Work

Generally, the number of actions is limited, they tend to be discrete, and the state space is limited Q-learning depends upon representing the state space and the action space. This usually involves discretizing the state space and the action space too. Fine discretization leads to massive data structures and the need for an unreasonably large number of learning trials, whereas a rough discretization lacks the precision to capture important differences between places in the state space that map to a single entry whereas it would be better if it were finer.

Being too rough, has a negative impact on the learning because it groups parts of the state space that should naturally be separate. This adds stochasticity of the problem domain in a bad way. If in state **x** the action **a** can lead to two outcomes r_0 or r_1 it can be because the action is stochastic in its nature or because two distinct states have been captured as a single state, but if the state had been encoded as two separate states, the actions would be deterministic, or at least less stochastic. If r_0 1% of the time leads to success whereas r_1 leads to disaster 99% of the time, the Bellman equation learns to avoid this action. While Deep Q-learning somewhat alleviates the problem by using deep-learning to estimate Q values, it brings other weaknesses that we discus later. Various approaches to representing continuous state space models have been proposed that use dueling Double-Deep Q-Networks (DDQN's) to represent continuous state spaces.

ReinforcementLearningconvergesonanoptimalpolicyforanMDP.AnMDPpolicycanberepresented as a table. A table lookup will give for every position in the state space the optimal action to perform.

The size of this table grows as the product of the number of discretized values of each dimension and the action dimension (the number of actions can be taken at any point). It would be infinite for continuous dimensions, but this can be discretized to avoid that problem while introducing others. Learning the optimal policy is achieved as a dynamic programming problem driven by the Bellman Equation [4].

3.3.3 Clusters as compact representations of state space

The approach is to learn to discretize the state space as is required by what is learned. The mechanism provides state space discretization even without any physical world correspondence. Because the state space grows only as it is needed. A learning system can represent the context space compactly which grows linearly with the number of contexts. High dimensional state spaces, which are normal in animal brains, and which cause state spaces to explode, no longer pose a scaling problem since each visited state is represented as a point in N-Dimensional space, which clusters with other points to form clusters representing contexts of which are only sufficient in number to support the fidelity required to represent the learning.

3.3.4 Overview of the approach

Rather than representing the immense Q-table directly, we frame the problem as contexts which are learned from values in the state.

Contexts are built from states that have occurred in the learning system. Initially, there is nothing. As points in the state space are observed, they are added to a growing collection of points that are continually being observed. When an action is taken, observations indicate a state transition.

`<oldstate, action, newstate>` represents a point in a space that is clustered. The clusters grow, divide, and coalesce as new points are found.

When the system observes a state, it maps to the cluster to which it belongs. From this, the actions and the new states can be read, just as they are in a conventional Q-table.

With this approach, continuous state spaces map naturally to clusters whose boundaries represent where an action will have a different outcome. The precise boundaries of the important points in a continuous state dimension can thus be learned simultaneously with the learning of the policy. Many clusters will be formed where necessary to capture the important parts of the state space whereas almost no clusters are generated where nothing interesting happens.

For any state, we can know where it is in a cluster and how close it is to a cluster boundary.

This approach can be used equally well for goal-less exploration for schema learning and reward driven learning for reinforcement learning. The key computational part is the incremental clustering algorithm can be computed, using a graphics processing unit (GPU), to track many points in parallel.

Initially we kept all points, which at some point takes up a lot of memory and takes longer, even for a GPU to compute. To solve this problem, when the number of saved states grows beyond a fixed number, we replace two points that are close together with a single point, set a position between the original two and which is given a count of 2. Later, in general, the points can be collapsed to form higher order points. As such we can place a parametric upper bound on the number of points stored with some loss of maximum attainable precision.

In the following sections, we describe the clustering algorithm and how that allows non-convex inter-tangled clusters to be learned incrementally.

3.3.5 Contexts

The algorithm builds upon previous development of a system called Grounded Reflective Adaptive Vision Architecture (GRAVA), that segments and labels aerial images in a way that attempts to mimic the competence of a human expert.

The states achieved by the learning system provide multiple positive examples of a structure that we wish to model. The structures in question have one or more dimensions, and the available observations provide examples of the structure that enable us to model the location within the appropriate multidimensional space. One way of doing this is to model the structures as a probability distribution function (PDF). Consider two-dimensional space and the collection of positive examples shown in Figure 7. Given a set of data points it is possible to find a mean point and the principle and secondary eigenvectors. In general, for a **n** dimensional space, there are **n** eigenvalues and Eigenvectors,

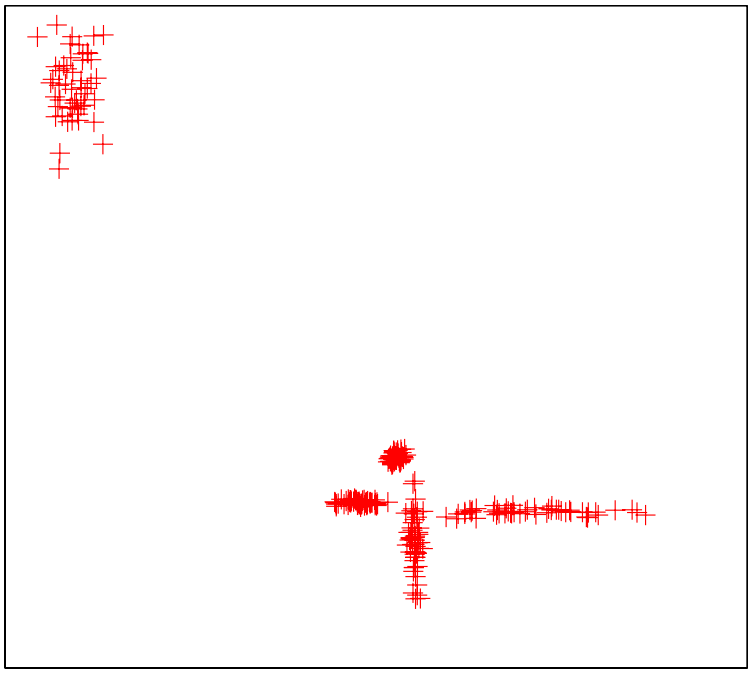

Figure 7. The Need for Decomposition

We provide two dimensional examples in this report because they can be graphically illustrated on a two dimensional medium. In practice the number of dimensions is far larger than two.

Unfortunately, the resulting model is unsuitably crude since most of the points that it generates are not suggested by the data. The problem gets successively worse for higher dimensional spaces.

We experimented with a number of standard clustering approaches such as K-means [1] and K-medoids but our use requires that the clustering be performed automatically on dynamically collecteddata and we needed an algorithm that was non-parametric. In particular, we should not have to specify the number of clusters or to specify typical values. We need to have the number of clusters and their composition be determined automatically. Furthermore, since natural clusters are often non-convex, we need an algorithm that can separate intertwined non-convex clusters. The algorithm described below depends solely on the notion of minimal description length and as such requires no parameters.

Principle component decomposition is the interpretation of a set of data points into the component collections (five in this example) by analyzing the principal components of the interpretation space.

The algorithm builds upon two earlier works. The first is a classification program developed by Wallace. Wallace's [2] SNOB program worked by finding a minimum message length (MML) description of a set of points. The second is the practice of using principal component analysis (see [3]) to reduce the dimensionality of high dimensional problems so that the separate populations can be modeled.

Our algorithm applies principal component analysis recursively to separate the collection into successively smaller clusters. At each point the criterion for separating a population is that it reduces the global description length of the original population.

Below we present our algorithm for producing such principal component decompositions.

3.3.6 A Statistical Model for Clusters

Given an n-dimensional space Sn containing m points. we can interpret the points in this space as being:

1. Unrelated points.
2. Unrelated points.
3. All members of a single cluster.
4. Grouped into a number of clusters.

A model has a shorter description length if it reduces the amount of uncertainty about the values of features. The best interpretation of the data points that constitute the collected state observations, therefore, is the interpretation that reduces the uncertainty about where the data points appear in the multidimensional space.

The entropy of the collection data points in a data set is given by:

$$H = -\sum_{d \in S_n} P(d) \log_2 P(d) \qquad (1)$$

The lower bound Minimum Description Length (MDL) of a description that represents all of the points in the S_n is given by:

$$DL = -\sum_{d \in S_n} \log_2 P(d) \qquad (2)$$

In order to compute this theoretical description length, it is necessary to know the PDF for points in S_n. A data set doesn't specify every possible point in the space. It provides a collection of *representative* points in the space. The job of interpreting the data set involves modeling the PDF. There are many choices for modeling a PDF. One model that is simple, predictive, and which often pertains to naturally occurring distributions is the Gaussian.

The description of a Gaussian model consists of a mean and variance of the distribution $< \mu, \sigma^2 >$. For a set of points the Gaussian model can be fitted simply by computing the mean μ and the variance σ^2. Given this characterization, for any point d we can compute the probability P(d) as follows:

$$P(d) = erf\left(\frac{(pos(d) - \mu_n + \epsilon/2)}{\sigma_n}\right) - erf\left(\frac{(pos(d) - \mu_n - \epsilon/2)}{\sigma_n}\right) \quad (3)$$

where pos(d) is the position of the point d, E is the position resolution, μ_n is the n-dimensional mean, σ_n^2 is the n-dimensional variance, and erf() is the error function.

The choice of whether to consider the points in the data set as (1) unrelated individual points, (2) all members of the same model, or (3) divided into groups each of which is modeled, is to select the choice that yields the minimum description length. The interpretation task can therefore be characterized as dividing the data points in S_n into n proper subsets $C_{i,n}$ such that:

$$S_n = \bigcup_{i=1}^{n} C_i \quad (4)$$

$$\arg\min_{C_{1,n}} \sum_{i=1}^{n} \left\{ \left(\sum_{d \in C_i} -\log_2 P(d|C_i) - \log_2 P(d \in C_i) \right) + ddl(C_i) \right\} \quad (5)$$

where $ddl(C_i)$ is the description length of the distribution used to model C_i. The description of a point is divided into two parts. The first part identifies its position in the space ($-\log_2 P(d|C_i)$) and the second part identifies to which collection it belongs ($-\log_2 P(d \in C_i)$).

The statistical models chosen for C_i determine the size of the point descriptions. In order to specify the position of a point we choose a resolution E to be used uniformly since otherwise a point can have an arbitrary precision and its representation would be arbitrarily large.

If the representation of a collection includes its mean position μ, the positions of the points in the collection can be described as distances δ from the mean. Figure 8 shows the representation of a point within a collection C_i as an n-dimensional mean (n = 2 in this example) and a n-dimensional displacement. So, any point d can be described as:

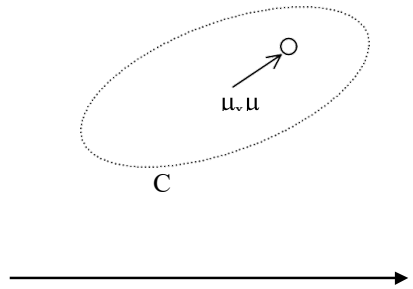

Figure 8: Representation of a Point in a Collection

$$\mu + \delta - \frac{\epsilon}{2} \leqslant d \leqslant \mu + \delta + \frac{\epsilon}{2} \quad (6)$$

Given the original set of points it is possible to reconstruct the statistical model that was used to represent it. So, to communicate the collections, all that is required is the mean position represented to an accuracy of E. The points are represented as a description of which collection they belong to and the offset from the mean: $< C_i, \delta >$.

If all points are separate collections, each of a single point, the size of their offset will be 0 since all points reside at the collections mean. Since clusters and points have a one-to-one relationship,and the point description contains no information other than the collection assignment, the representation only requires the positions of the individual points in the collection. Collections that are represented as individual points in this way have no predictive value.

If all points are members of a single collection the representation of a point doesn't need to identify to which collection it belongs because there is only one collection.

As the data points in a data set are divided up into smaller collections the description length of the individual points is reduced if the distribution that characterizes the collection is more predictive about the position of its component points than the distribution for the entire data set was. Any suitable statistical distribution can be chosen for a collection.

3.3.7 Algorithm for Decomposition

Having defined the criteria for an optimal division of the data points into separate models we are left with the task of defining an effective procedure for achieving such a division. To accomplish this, we developed an efficient algorithm that approximates a solution to Equation 4. Our algorithm, which we call "principal component decomposition" (PCD), attempts to divide the data by searching for dividing hyper planes tangential to the eigenvectors of the data. The intuition behind the algorithm is that the principal eigenvectors represent the dimensions with the greatest spread. The spread can be caused by a single phenomenon with a large variance, or it can be caused by more than one phenomenon distributed throughout the space. To distinguish these two cases, we compute the entropy of the data points as a whole and then we compute the sum of the entropies of the two collections formed by dividing the data points into two collections with a hyper plane perpendicular to the eigenvector. We do this for all possible cut points along the eigenvector. If all sums of divided collections yield a higher description length than the original combined collection the collection is not divided, otherwise the collection is divided at the place that yields the minimum description length.

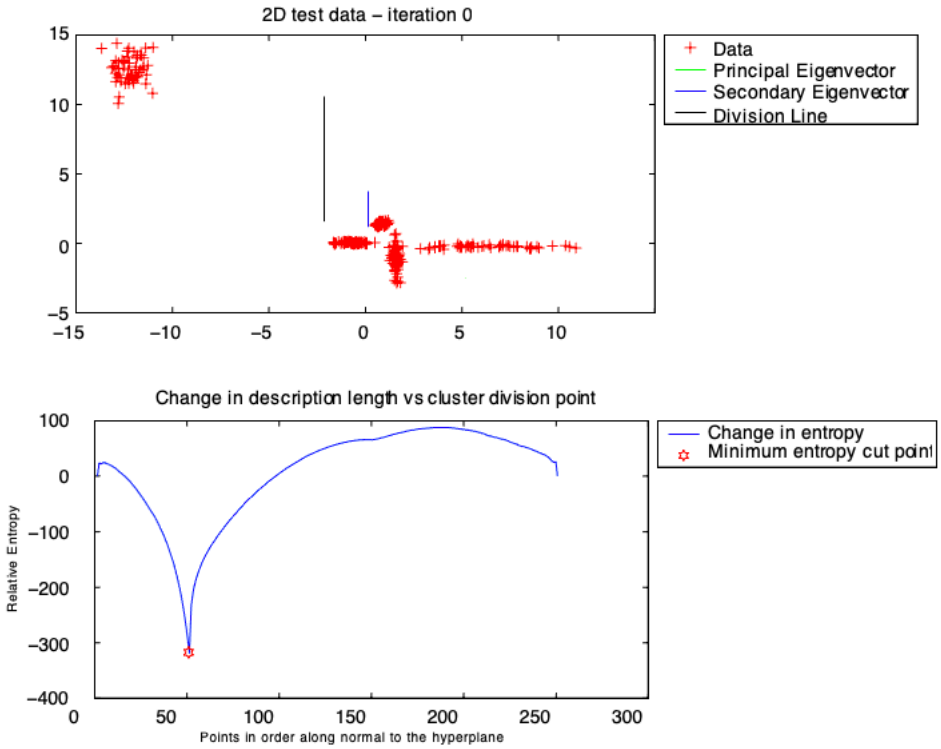

Figure 9 Dividing the Data Points to Reduce Description

Figure 9 shows the 2-dimensional data introduced earlier. There are two eigenvectors. The lengths of the principal and secondary lines are the square root of the corresponding eigenvalues.

The hyper plane that is used for cutting the data is perpendicular to the principal eigenvector. The graph below shows the change in the total description lengths resulting from cutting the collection at any point along the eigenvector.

A 2-dimensional hyper plane is a line.

The eigenvectors are computed from a co-variance matrix, so the eigenvalues are variances, and the square root of the eigenvaluesare standard deviations.

When the change is greater than zero, cutting makes the description length larger. In this case the total description length is significantly reduced by cutting the collection at the point where the "division" line is drawn. This point can be seen as the minimum point in the entropy curve.

This procedure is repeated for each eigenvector of the collection starting from the eigenvector that corresponds to the largest eigenvalue until either a division occurs or until all eigenvectors have been tried. Once a collection has been split the algorithm is applied to each of the newly divided collections. Eventually there are no collections of points that split. The algorithm consists of two parts CHOP and MERGE.

CHOP looks for places to divide a collection of data points into two collections by finding a dividing hyper plane as described above. CHOP thus produces two collections that have the property that if collection C_0 is divided into C_1 and C_2, $C_0 = \cup\{C_1C_2\}$, and $DL(C_0) > DL(C_1) + DL(C_2)$. MERGE finds two collections of data points (say C_1 and C_2) that have the property that $DL(\cup\{C_1C_2\}) < DL(C_1) + DL(C_2)$. If the collection of data points is non-convex CHOP can cause some points to become separated from the collection to which they naturally belong. MERGE re-associates points severed in this way with their natural collection. The advantage of this approach is that it is possible to construct non-convex collections of data points.

First, we describe the algorithm for CHOP(S) that chops the collection into separate collections.

CHOP(S):

1. S is a set of n-dimensional data points. Let $\bar{m}$ be the mean and C be the co-variance matrix.
2. Let $v_1 \ldots v_n$ and $\lambda_1 \ldots \lambda_n$ be the eigenvectors and corresponding eigenvalues, respectively, sorted into decreasing order of eigenvalue.
3. For each eigenvector v_i starting with v_1 (the one with the largest eigenvalue—the principle eigenvector), search for the best place to cut the data points into two collections as follows:
   a) Establish the cutting hyper plane. The cutting hyper plane is the plane that is perpendicular to the eigenvector v_i. We arbitrarily choose the hyper plane that passes through the mean $\bar{m}$.

$$n = \bar{m}^T v_i$$
$$\bar{r} v_i = n \qquad (7)$$

where $\bar{r}$ is a point specified as a row matrix.

This is the perpendicular form of the equation of a hyper plane. This representation is convenient because it permits fast calculation of the distance of a point from the hyper plane. For any point d the distance from the plane in equation 7 is given by $n - dv_i$.

b) Sort the points in S in order of distance from the cutting hyper plane. Since the hyper plane cuts through the mean, approximately half of the points will be on one side of the hyper plane, with the rest on the other side. Approximately half of the points, therefore, will have a negative distance from the plane. The distance is not the absolute distance from the plane, it is how far to move along the normal to the hyperplane to reach the plane in the direction of v_i.

c) Let A be the sorted list of data points.

d) Let B be an empty list.

e) Let the cutPoint = 0 and position = 0

f) Now we simulate sliding the cutting hyperplane along the eigenvector from one end of the set of data points to the other, by taking points one at a time from A, putting them into B, and computing the description length of the two collections as follows:

For each point d_j in A do:

i. Remove d_j from A.

ii. Add d_j to B.

iii. Increment the position (position = position + 1).

iv. Compute the new description length as newDL = DL(A) + DL(B).

v. If newDL < minDL set minDL = newDL and cutPoint = position.

g) If cutPoint > 0 divide the data points S into two collections S_1, and S_2 at the position indicated by cutPoint. Then recursively apply CHOP to both sub-collections to see if further chopping can be performed. Finally return the complete list of chopped collections:

return append(CHOP(S_1), CHOP(S_2))

4. At this point, all of the eigenvectors of S have been searched for chop points, and none have been found. The data points cannot be represented with a smaller description length by chopping along an eigenvector so return the list of collections as the single collection S:

return list(S)

The nature of the way the collections are divided up results in some groups of data points being divided unnecessarily. This can be seen in the second and third iteration for this example data.

In the first iteration (Figure 10 top) one point is chopped off the right most collection. Later, in the fifth iteration (Figure 10 bottom right) two points are completely severed, so as to be small, disembodied collections of points (one point and two points respectively in this example). These accidentally severed

fragments are corrected in phase two of the algorithm—the merge phase.

In phase two of the algorithm (MERGE) pairs collections of points are checked to see if the description length would be reduced if they were to be merged. If the description length would be reduced by merging them they are merged. This is repeated until no more useful merges can be found.

MERGE(C):

If C is a collection of clusters resulting from the application of CHOP to the original data, the merge phase proceeds as follows:

1. For each pair of clusters $C_i \in C$ and $C_j \in C$ check to see if the description length can be reduced by merging them ($DL(C_i) + DL(C_j) > DL(C_i \cup C_j)$).
2. If no merge candidates are identified in (1) the merge phase is complete.
3. Produce C^1 by replacing each pair of mergeable clusters with their merged union.
4. Repeat steps 1, 2, and 3 on the new set of clusters.

In the given example, after 16 iterations the division of the data points in to separate collections is complete.

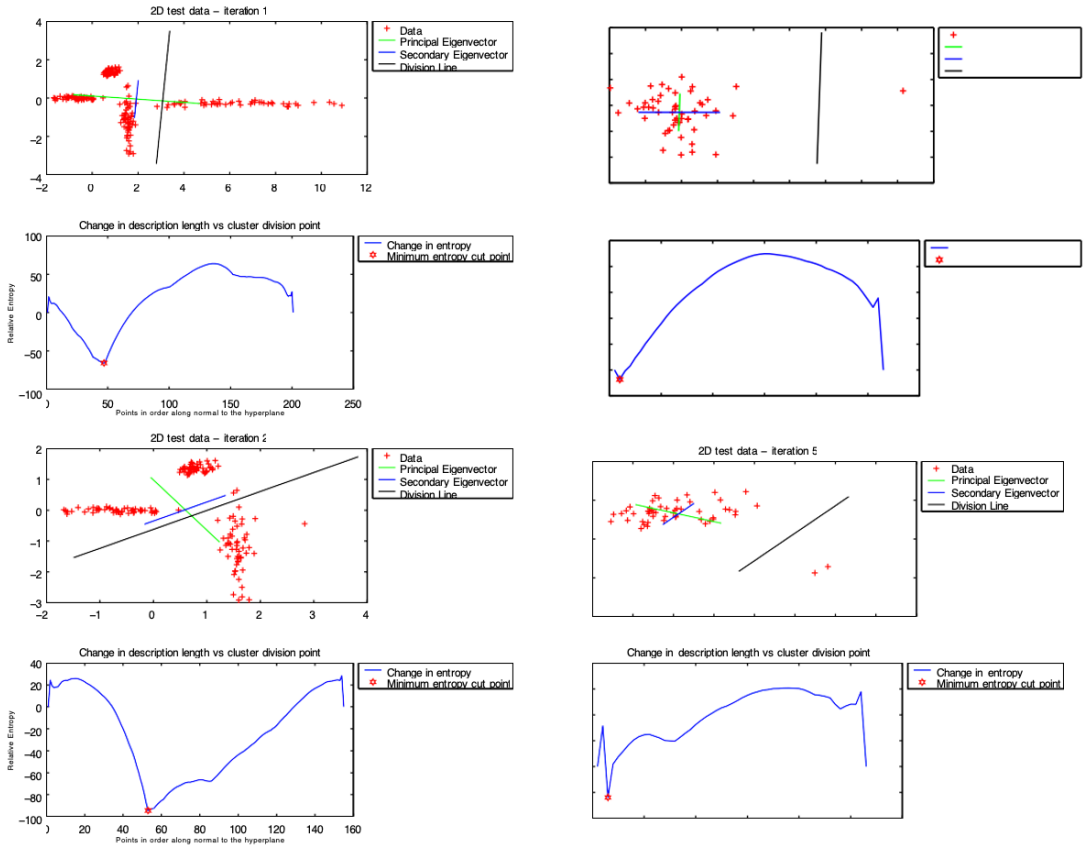

Figure 10: Accidentally Severed Points

Figure 11 shows the final result of decomposition.
The algorithm described above has a number of interesting characteristics:

1. PCD produces a structural description of the data points that is an approximation to a global MDL description of the points.
2. Each remaining collection of points can be represented efficiently by the statistical model chosen for it since if the collection could not be represented well it would have been divided.
3. Each collection is a good candidate for Principal Component Analysis (PCA) modeling because of (2).
4. The algorithm can be implemented efficiently and can produce good decompositions very quickly using a GPU implementation. The number of "chop" and "merge" operations that are performed in producing a decomposition is very small compared to the number of points.

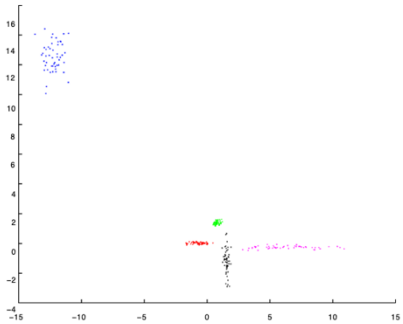

Figure 11: Example Result of Principal Component De- composition

5. The algorithm can produce non-convex collections.

The final point (5) is an interesting feature ofthe algorithm that is not obvious from the example given above. Non-convex collections cannot be disentangled by using the "chop" operation alone, but inclusion of the "merge" operation allows two convex collections to be joined to produce a non-convex merged collection.

To demonstrate this capability, we generated a set of data points by picking points randomly along two interlocking 'C' shapes in a ying-yang configuration. Even though the data is quite dense and intertwined the algorithm manages to "chop" it apart and then "merge" the severed parts back together. Figure 12 shows the first iteration on the data.

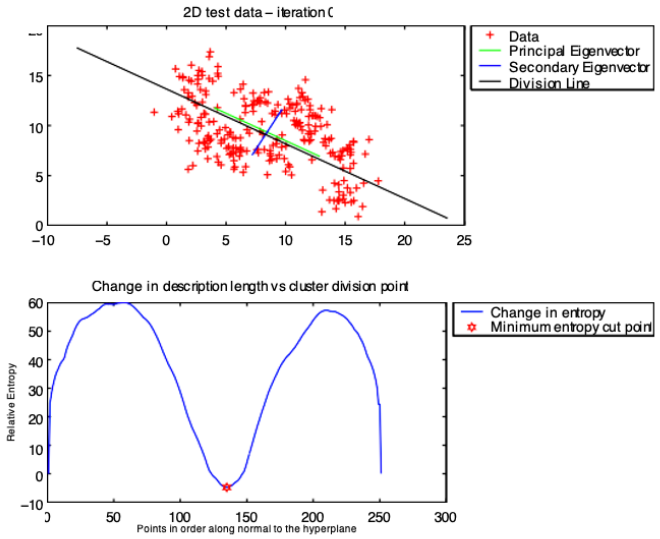

Figure 12: Intertwined Non-Convex Shapes

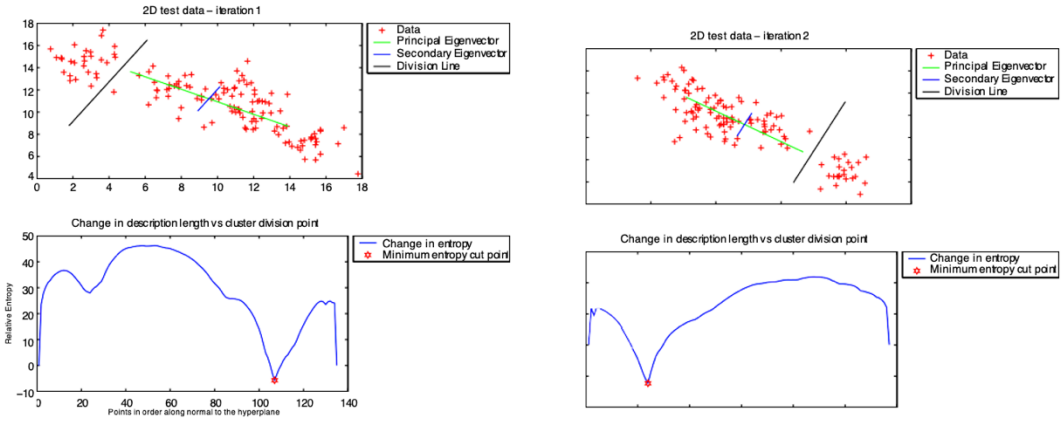

Figure 13: Curved example: CHOP

The second and third iterations chop the data down further as shown in Figure 13.

Later iterations merge the severed portions back into their rightful places as shown in Figure 14.

The final decomposition of the data is shown in Figure 15.

It should be noted that separating clusters in a 2D space is harder than in a higher dimensional space because higher dimensional spaces are naturally sparser. This is a similar observation to that used to good effect in support vector machines. Sometimes, high dimensionality is a benefit and not a problem.

We have developed an approach to dynamically constructing a state space map by decomposing complex models into collections of simpler models. This forms a backbone mechanism for interpreting learning policies.

The algorithm has some important features:

1. The algorithm supports non-convex shapes.
2. The algorithm uses the MDL criteria for interpretation.
3. The algorithm doesn't over fit. The algorithm doesn't try to circumscribe a set of data points. Itsimply tries to separate collections of points by finding the best place to cut.
4. The algorithm is fast because it only searches for cut points along eigenvector dimensions.
5. The algorithm is non-parametric which removes one more barrier to automation.

The important observation from the standpoint of learning a recovery policy using a reinforcement learning approach, is that instead of having a static huge representation of a discretized state space, we have a dramatically smaller number of clusters, each one of which represents something that is importantto the policy being learned. If more divisions are necessary, the clusters divide naturally as new points are learned.

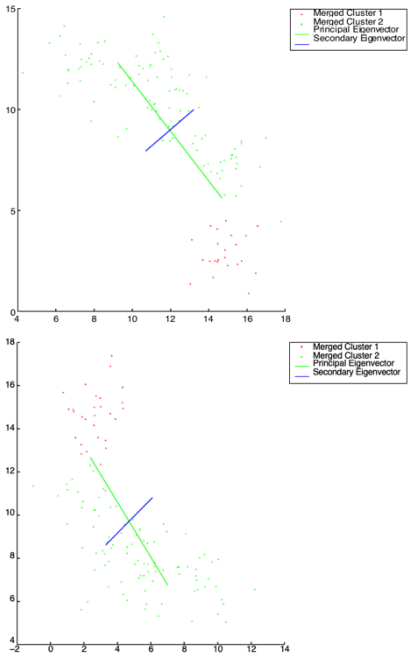

Figure 14: Curved example: MERGE

3.4 LEARNED RECOVERY CONTROLLER

Given the above apparatus, learning the recovery controller simply involves initializing the simulatedquadcopter outside of the safe operating configuration such as is shown in Figure 16.

Regardless of the shape of the region, we learn by picking red points on the grid that are adjacent to a green node and run episodes that succeed if the quadcopter achieves a green area, and which fails if the

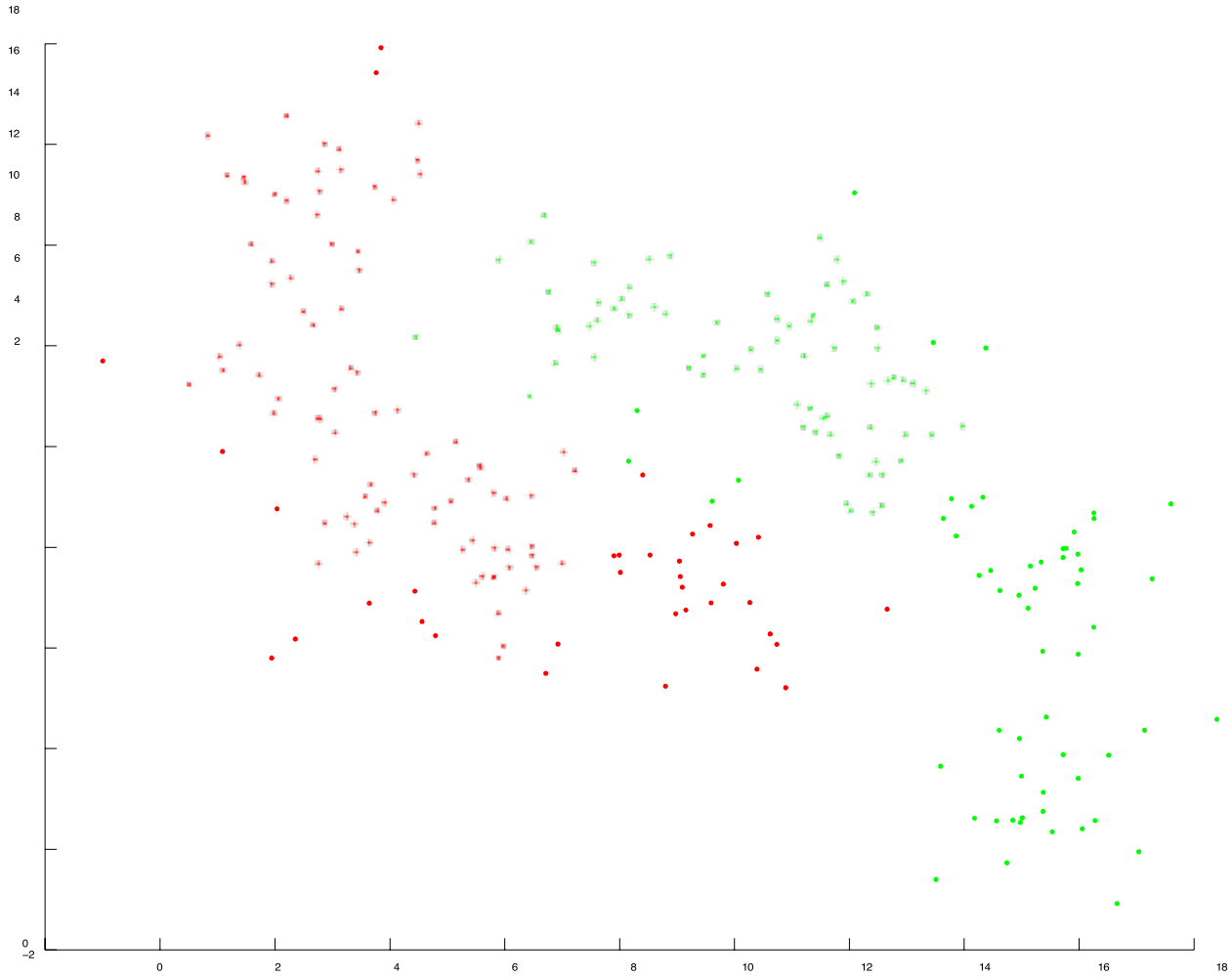

Figure 15: Final Decomposition of Example Non-Convex Data quadcopter crashes.

quad-copter crashes. We have a parameter to decide to consider the episode failed if it has not achieved the green zone before a fixed number of steps. Fortunately, for this scenario, the quad-copter crashes rather quickly in the absence of a good solution. When a policy has been learned for the first point, we restart with another point close to the first, usually adjacent, and run the learning again. Finding a solution to the first case takes a long time, but subsequent points, which are close in the state space, are learned fast because they differ only slightly from the previous case. Eventually, this results in a policy that can recover from any quad-copter configuration. Note, that in general, we would expect some parts of the state space to be unrecoverable and in those cases, we should develop a separate run-time monitor for cases where the learned controller has been unable to find a policy that can recover it. Maybe, we should have three colors for the zones: Green for the cases where the built-in controller has no problem; Orange, where the built-in controller cannot control the CPS, but the learned controller can recover it; and Red where neither the built-in controller nor the learned recovery controller are able to regain control. This latter category would potentially give designers possibilities for minimizing damage. Since our quad-copter scenario did not have any such regions, given sufficient altitude, we did not implement a second monitor for unrecoverable configurations.

It should be noted that the learning time was of the order of weeks, but the result was a controller capable of reestablishing control, albeit with some crazy looking maneuvers.

Figure 17 shows the start of an episode. The quadcopter starts up on the ground and climbs to a preset altitude. We tried instantiating the quadcopter already at altitude, but the built-in controller couldn't control the vehicle starting from rotors off at altitude. In order to make the scenario realistic in terms of QC start, rotors running at the correct speed to maintain altitude, we always start the episode on the ground and fly up to the desired altitude and then perturb the orientation of the QC and let the learner attempt to recover from it. We used the same strategy for learning the safe configuration for the monitor. It adds a few seconds to the startup of an episode, but it guarantees that the QC state is realistic at the start of each episode.

Once stable at the desired altitude, the quadcopter is perturbed into the selected Red configuration, as shown in Figure 18. In this case we have selected an orientation that is the least necessary for the built-in controller to fail. It is adjacent to a Green node in the safe/unsafe map. Although this configuration seems benign to the naked eye, it is a point at which the quadcopter begin to fall fast. The built-in controller never recovers from this orientation and the quad-copter crashes quickly. In this run, we are learning the recovery controller, and early in the training all episodes fail.

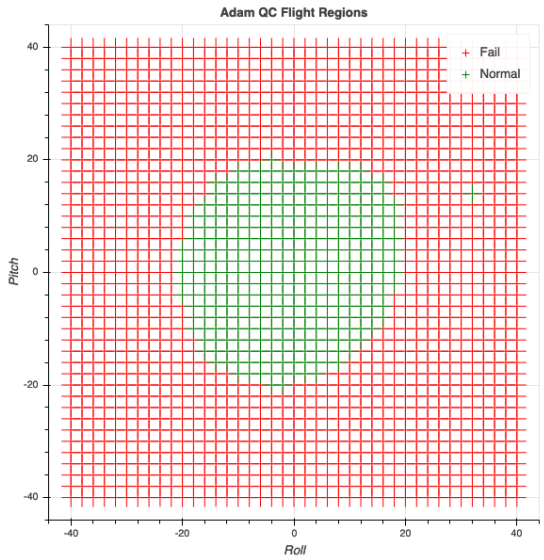

Figure 16: Circumnavigating the dangerous (red) region

Figure 18 shows the quadcopter just a fraction of a second after the previous one.

And finally, Figure 19 shows the quadcopter inverted on the ground after a crash.

It takes several seconds to restart after each episode and in early episodes, all episodes end in failure, with some amusing twitching of the falling quadcopter as it tries actuation commands that might save it without success. It would be easy to believe that the approach wasn't working but after several days of trials it produces a success. Success is output into the log, so we can see that it happened, but you must be in front of the screen to see it, and that is implausible.

As is normal with reinforcement learning, after the first success, subsequent successes begin to occur more frequently and finally, they begin to occur frequently enough that watching the screen for a few minutes will demonstrate a recovery.

The nature of the early recoveries is laughably haphazard, and the quadcopter can be seen flailing about before recovering. Over time the learned recovery controller learns to recover efficiently with few deviations from an optimal recovery.

3.5 TRANSFER LEARNING

Our last experiment was to try transfer learning from a policy learned on the crude but fast simulator and used as a starting point for the Gazebo simulator. Time ran out on our experiment before we could claim success, but the reasoning for this is worthy of discussion.

The idea behind transfer learning was originally to solve the question of how we could trust a recovery controller that was learned in simulation and which we dare not try with a physical CPS because if it doesn't work, bad things will happen, expensive bad things!

The thought was that if we learned the best that we could from a good simulator, there would inevitably be differences when transitioning to the learned controller on the physical platform. If those differences were largely parametric, we could compare the results of actuation commands with what was expected and make parametric adjustments so that we could have faith in the learned controller. In many cases, these parametric adjustments could be automated and could occur without ever putting the CPS in danger.

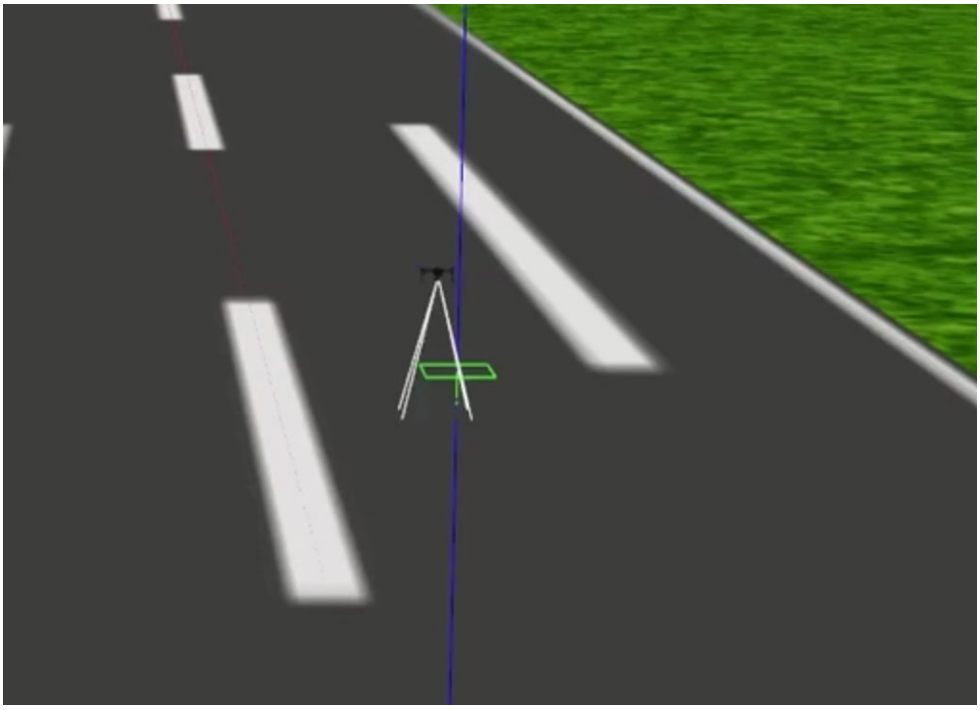

Figure 17: Starting a learning run: Takeoff

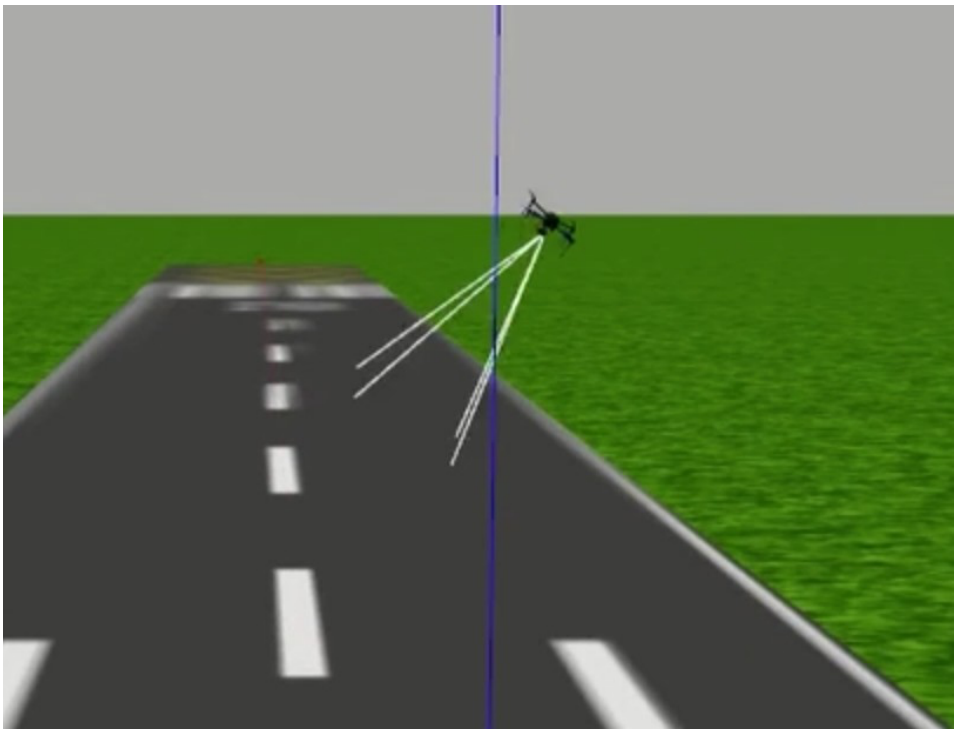

Figure 18: Establishing a "Red" configuration at the start of an episode

Consider the quadcopter. If the thrust required to maintain level hovering in the simulator is slightly different from the physical controller, we can put the physical quadcopter into level flight and upon switching to the learned controller it would observe a gradual increase or decrease in altitude and could adjust the rotor speeds accordingly. Of course, we want to do this in a way that is general and does not depend upon the specific platform. One approach is to have the learned controller "watch" the built-in controller during normal use and continually update its model, so that when the learned controller is invoked, it will be perfectly adjusted. This is an especially good idea when the platform is likely to change during its lifetime. In real CPS systems, such as the Mars Spirit rover, a wheel became unusable, others CPSs just degrade gradually over time. This kind of wear can affect jet engine performance, wheel stiffness, and may, in some cases, change after a maintenance event, like adding tire pressure! It is clear, that this kind of reparameterization should take place continuously at run-time, and that the controller should be able to self-diagnose if it is unable to self-adjust. That way, we can be assured that the recovery controller can be trusted, on the physical system, despite have been trained on a simulator.

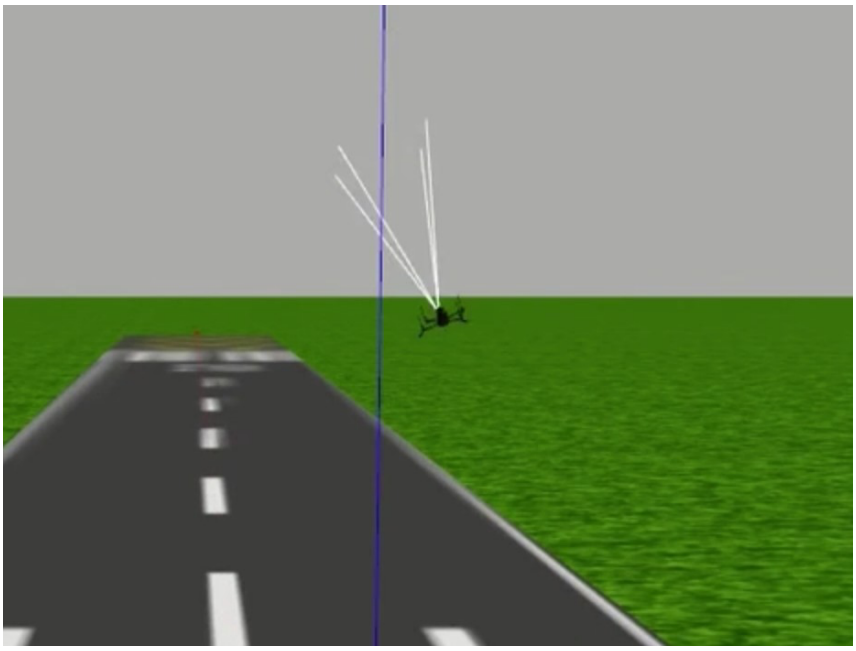

Figure 19: Flight deteriorates rapidly and accelerates towards the ground.

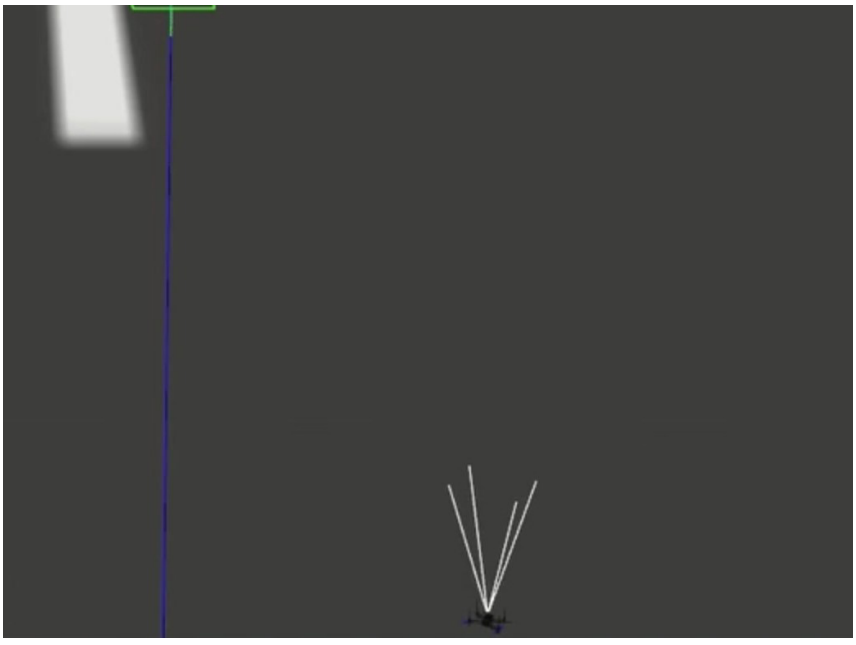

Figure 20: All early episodes end in a crash.

4. RESULTS AND DISCUSSION

When the monitor detects the "out of the green" configuration and switches to the learned recovery controller, the learned recovery controller uses the learned policy to select actuation commands after receipt of new sensor data. The actuation commands are to set the four rotor speeds that can range from a negative value, representing reverse thrust to a positive value representing normal thrust. We used a fixed period for the sample and respond cycle. When new sensor data arrives, it is used to lookup the possible actuation commands and the quality. One is selected and the loop waits a few milliseconds before its next iteration.

Figure 21 shows a successful recovery from a perturbation. It was an easy case, so not too much altitude is lost in recovering.

Severe cases, however, take much longer to recover.

Figures 22, 23, 24, and 25 show the recovery from a completely inverted case. It gathers a lot of downward speed before being able to recover an upright configuration. Even when upright, it takes a while for the thrust to compensate for the downward momentum achieved during the first phase of the fall and during the rotation to upright configuration. Finally in the last image (Figure 25), the quadcopter has recovered, it proceeds to regain its altitude and pass control back to the built-in controller.

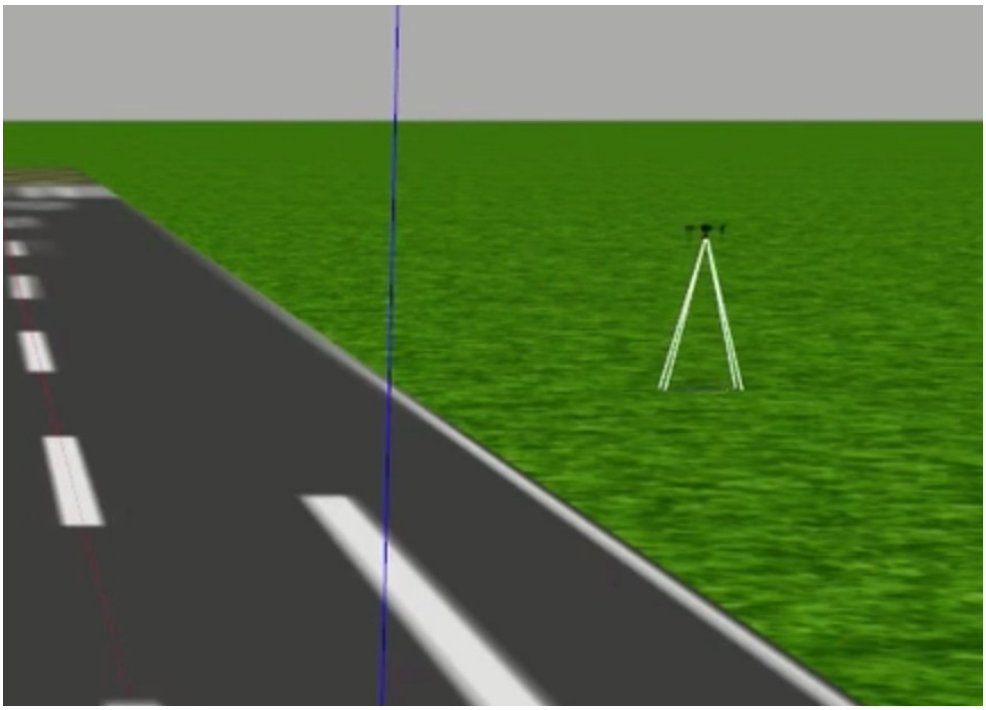

Figure 21: Fast recovery from an easy case.

4.1 Estimates of Technical Feasibility

With the clustering approach to state space discretization, we believe that the approach is capable of handling complex cases, especially when the approach is optimized to take full advantage of massive parallelism. The side of the state space is a function of the complexity of the problem rather than on the dimensions of the sensors.

We believe that the approach is general enough to support a wide range of CPS types. We made an effort to not bake in any knowledge of the quadcopter. The plant interface to the simulator, of course, needs to enumerate what actuation commands are available for the reinforcement learning (RL) learner to try and simulate what sensor data is available. We believe that recovering an imminent nuclear reactor melt down would fit in this model.

There will always be some domain specific knowledge, such as those just mentioned, but the core approach is general.

The difficulty with implementing the big switch will be different for every CPS, and in some cases the transition back to the built-in controller might have to be phased in. Wrapping black boxes is hard! Of course, new systems that are designed for this kind of architecture would be able to design in hand-offs of control that are less complicated.

The period of the sense act loop will vary from system to system as will the frequency of availability of sensor data. For our case, using ROS and RabbitMQ as the means for communicating sensor data introduces some lag in response, which is not ideal, but with trial and error, we were able to find a period that worked well enough. Arguably, a faster response time would make the recoveries less jerky, especially when things are changing rapidly. A quadcopter that is spinning out of control moves a lot in a few milliseconds.

4.2 Open Issues

As described above, we didn't get the chance to complete our experimentation of transfer learning, which was disappointing. We hope to be able to find opportunities to complete this step, perhaps as an internal research and development (IR&D) effort.

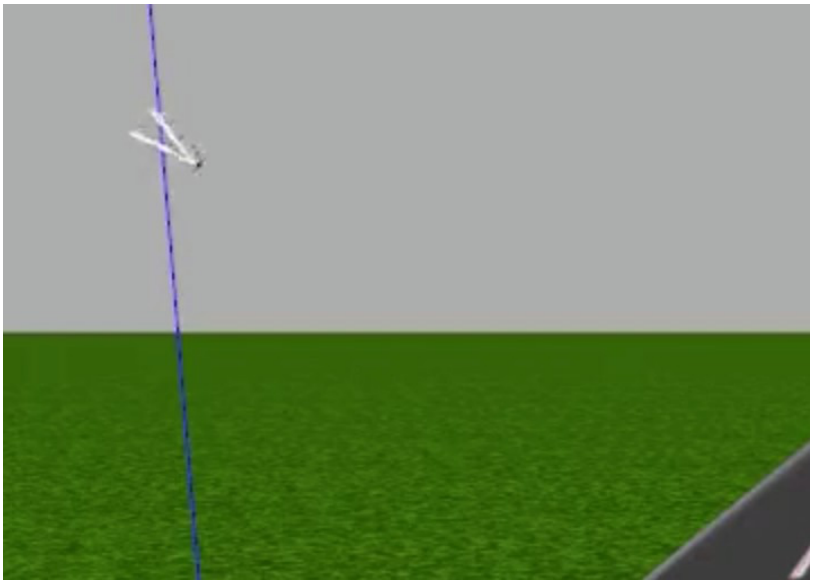

Figure 22: Recovering from an almost inverted perturbation

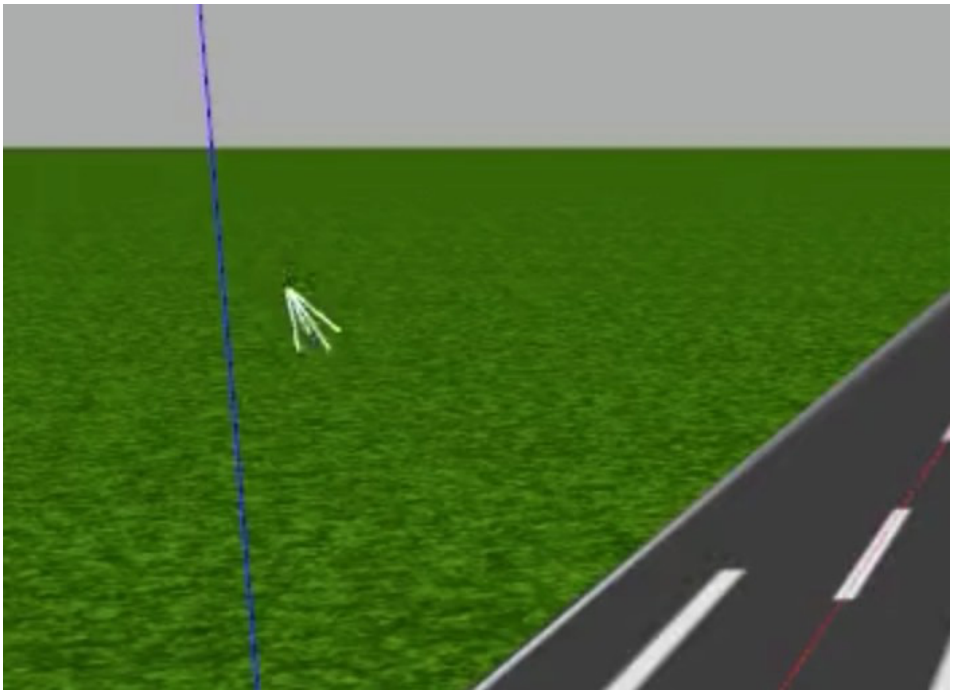

Figure 23: Recovered orientation

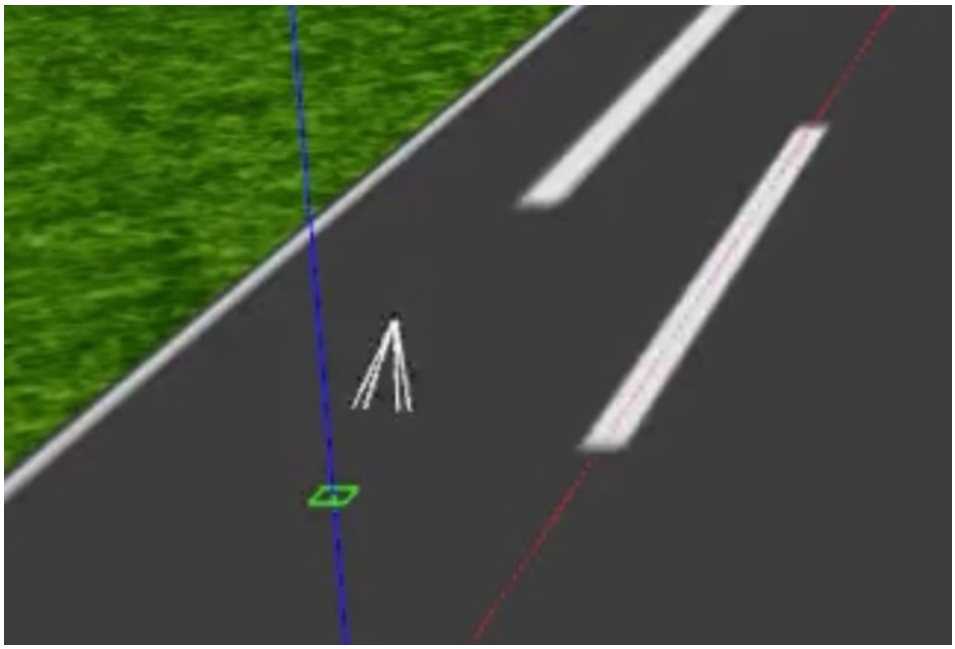

Figure 24: Falling fast

Figure 25: Avoided impact

5. CONCLUSIONS

Whereas some failure modes will always be unrecoverable, our experiments have shown that we can use good simulators to learn fast run-time monitors. The monitors are fast because they depend upon a fairly simple deep neural network. These monitors, even in the absence of recovery can be useful for designers of new CPSs. They provide a sensor that says that the CPS is operating out of its safe region.

The big switch/learned recovery controller provides for automatic recovery from serious situations. One can imagine a car that takes control when it detects a skid on ice in order to regain control as efficiently as possible before handing control back to the human or autonomous operator. Similarly, automatic recovery from stalls and spins could be built into planes. These, however, are skills that drivers and pilots are expected to learn and arguably don't need to have automated. Twin engine planes give their owners confidence that in the event of an engine outage, the plane can be flown on a single engine. A lot of accidents occur when the single engine failure occurs because the owners are not sufficiently experienced in flying in the single engine mode in which thrust is asymmetric.

Undoubtedly the most convincing use cases involve CPSs whose complexity is beyond what a human can reasonably be expected to manage. Electrical power grids, power plants, and high-tech aircraft come to mind as candidate targets for this approach. As systems become more complex, the need for this kind of run-time monitoring and recovery controllers will continue to grow.

6. REFERENCES

[1] Hart, and D.G. Stork. Pattern Classification. John Wiley and Sons, 2001.

[2] C.S. Wallace. Classification by minimum-message-length inference. In G. Goos and J. Hartmanis, editors, Advances in Computing and Information–ICCI'90, pages 72–81. Springer-Verlag, 1990.

[3] J.E. Jackson. A user's guide to Principal Components. John Wiley and Sons, New York, 1991.

[4] R. Bellman. Dynamic Programming. Princeton Landmarks in Mathematics, Princeton University Press, 1957.

7. LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

Acronym	Definition
ADAM-DMS	Assured Autonomy using Dynamic Monitors and Simulation
CPS	Cyber-Physical System
DDQN	Double Deep Q-Network
GPU	Graphics Processing Unit
GRAVA	Grounded Reflective Adaptive Vision Architecture
GUI	Graphical User Interface
IMU	Inertial Motion Unit
IR&D	Internal Research and Development
LEC	Learning Enabled Component
MDL	Minimum Description Length
MDP	Markov Decision Process
MML	Minimum Message Length
NN	Neural Network
ODE	Ordinary Differential Equation
PCA	Principal Component Analysis
PCD	Principal Component Decomposition
PDF	Probability Density Function
POMDP	Partially Observable Markov Decision Process
QC	Quadcopter
RL	Reinforcement Learning
ROS	Robot Operating System
SITL	Software In The Loop
SNOB	A clustering algorithm due to C. S. Wallace based on a minimum description length formulation.
UAV	Unmanned Air Vehicle