Achieving Broad Access to Satellite Control Research with Zero Robotics

by

Jacob G. Katz

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Aeronautics and Astronautics
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2013

Author .....
Department of Aeronautics and Astronautics
May 23, 2013

Certified by .....
David W. Miller
Professor, Aeronautics and Astronautics
Thesis Supervisor

Certified by .....
Alvar Saenz-Otero
Principal Research Scientist, Aeronautics and Astronautics
Thesis Supervisor

Certified by .....
Jeffrey Hoffman
Professor of the Practice, Aeronautics and Astronautics
Thesis Supervisor

Certified by .....
Emilio Frazzoli
Professor, Aeronautics and Astronautics
Thesis Supervisor

Accepted by .....
Eytan H. Modiano
Professor, Aeronautics and Astronautics
Chair, Graduate Program Committee

Achieving Broad Access to Satellite Control Research with Zero Robotics

by

Jacob G. Katz

Submitted to the Department of Aeronautics and Astronautics
on May 23, 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Aeronautics and Astronautics

Abstract

Since operations began in 2006, the SPHERES facility, including three satellites aboard the International Space Station (ISS), has demonstrated many future satellite technologies in a true microgravity environment and established a model for developing successful ISS payloads. In 2009, the Zero Robotics program began with the goal of leveraging the resources of SPHERES as a tool for Science, Technology, Engineering, and Math education through a unique student robotics competition. Since the first iteration with two teams, the program has grown over four years into an international tournament involving more than two thousand student competitors and has given hundreds of students the experience of running experiments on the ISS.

Zero Robotics tournaments involve an annually updated challenge motivated by a space theme and designed to match the hardware constraints of the SPHERES facility. The tournament proceeds in several phases of increasing difficulty, including a multi-week collaboration period where geographically separated teams work together through the provided tools to write software for SPHERES. Students initially compete in a virtual, online simulation environment, then transition to hardware for the final live championship round aboard the ISS. Along the way, the online platform ensures compatibility with the satellite hardware and provides feedback in the form of 3D simulation animations. During each competition phase, a continuous scoring system allows competitors to incrementally explore new strategies while striving for a seat in the championship.

This thesis will present the design of the Zero Robotics competition and supporting online environment and tools that enable users from around the world to successfully write computer programs for satellites. The central contribution is a framework for building virtual platforms that serve as surrogates for limited availability hardware facilities. The framework includes the elaboration of the core principles behind the design of Zero Robotics along with examples and lessons from the implementation of the competition. The virtual platform concept is further extended with a web-based architecture for writing, compiling, simulating, and analyzing programs for a dynamic robot. A standalone and key enabling component of the architecture is a pattern for

building fast, high fidelity, web-based simulations. For control of the robots, an easy to use programming interface for controlling 6 degree-of-freedom (6DOF) satellites is presented, along with a lightweight supervisory control law to prevent collisions between satellites without user action.

This work also contributes a new form of student robotics competition, including the unique features of model-based online simulation, programming, 6DOF dynamics, a multi-week team collaboration phase, and the chance to test satellites aboard the ISS. Scoring during the competition is made possible by possible by a game-agnostic scoring algorithm, which has been demonstrated during a tournament season and improved for responsiveness. Lastly, future directions are suggested for improving the tournament including a detailed initial exploration of creating open-ended Monte Carlo analysis tools.

Thesis Supervisor: David W. Miller
Title: Professor, Aeronautics and Astronautics

Thesis Supervisor: Alvar Saenz-Otero
Title: Principal Research Scientist, Aeronautics and Astronautics

Thesis Supervisor: Jeffrey Hoffman
Title: Professor of the Practice, Aeronautics and Astronautics

Thesis Supervisor: Emilio Frazzoli
Title: Professor, Aeronautics and Astronautics

Acknowledgments

This work encompasses four years of design and development of the Zero Robotics program, none of which would have been possible without the efforts of a diverse team spanning academia, government, and industry.

The original inspiration for Zero Robotics began with **Dr. Gregory Chamitoff**. Following his chance to operate his own algorithms on SPHERES during Expedition 17/18 aboard the ISS, he returned with a challenge to us to give the same experience to students around the country. Since then he has been a guiding influence on the evolution of the program. I am personally grateful for his advice throughout the course of my research, especially for his detailed feedback in preparing the final version of this thesis.

Dr. Lorna Finman is directly responsible for making the concept of Zero Robotics a reality by providing financial support for the first pilot season of the competition in 2009. In that season our two founding teams, Absolute Zero from Bonners Ferry, Idaho, and Team Delta from Post Falls, Idaho, helped to establish the key components of the competition, including the first suggestion to create a fully online environment for hosting the program. A special thanks to the mentors of these teams, **Brian Induni**, **Salvatore Lorenzen**, and my father, **Edward Katz**, for their patience while simultaneously teaching students and helping us to smooth out the wrinkles in our structure.

Bringing Zero Robotics to a national scale involved an extensive collaboration with two industry partners, **Aurora Flight Sciences** and **TopCoder**. Aurora created the graphical programming interface for the Zero Robotics IDE and the initial prototype of the 3D simulation visualization tool. I have had the privilege of working with **James Francis** to design and improve these tools, and **Wendy Feenstra**, who was instrumental in helping to manage the 2012 tournament and in launching a new middle school tournament for the summer of 2013. I am also thankful for Aurora's generous support during my tenure as an Aurora Flight Sciences Fellow in 2010. With the assistance of many competing software developers, TopCoder took our prototype

online platform to a cloud-based production architecture in the course of four short months. Thank you to **Ira Heffan**, **Mike Lydon**, **Andrew Abbott**, and **Ambi Del Villar** for their strong enthusiasm for the program and for involving me firsthand with all levels of the TopCoder crowdsourcing process.

At the government level, Zero Robotics has had strong support from both **NASA** and **DARPA**, including funding, access to crew time for running the live tournaments, publicity efforts, and operations support. I would like to thank **Jason Crusan** at NASA for his steady support and constructive input for improving the program.

The Zero Robotics team at MIT has included both graduate research assistants and undergraduates as part of the Undergraduate Research Opportunities Program (UROP). Thank you to graduate students **Dr. Swati Mohan** for helping to run the 2009 pilot season and assemble the first proposal for funding Zero Robotics, **Sreeja Nag** for serving as the student lead during the 2011 tournament, **Sonny Thai** for designing the first collaborative IDE for the online platform, and **Prashan Wanigasekara** for his assistance with the 2012 season and the 2013 game design. A special thanks to the many UROPs that have contributed designs for games during each season and for giving me the opportunity to teach as well as learn.

More than 200 teams have participated in Zero Robotics tournaments, and many have given invaluable feedback through surveys, online forums, and emails. I would like to specially thank **Rich Kopelow** and his team **y0b0tics!**, and **Steven Pendergrast** and his team **Kuhlschrank** for never hesitating to give us candid feedback, and for their dedication to improving the program. Thank you also to **Anne Contney** and **Team Rocket**, also known as "The Sledgehammers," for battering our servers on a consistent basis and calmly waiting for us to fix the problems.

My time in the Space Systems Laboratory has seen the fulfillment of many childhood dreams–working with astronauts, watching a space shuttle launch, controlling a robot in space, and so many more–for which I owe an immense gratitude to my advisors **David Miller** and **Alvar Saenz-Otero**. They have not only given me once in a lifetime opportunities, but have also equipped me to keep pursuing them. I have also worked excellent colleagues, including my officemates **Brent Tweddle** and

Jaime Ramirez, with whom I've been fortunate to share many discussions and the occasional venting of steam.

To my loving parents **Edward** and **Jill Katz**, thank you for your unending support through 10 amazing years of MIT and experiencing, right along with me, the now countless wonderful experiences here. Most of all, thank you for the joy of learning and a life to pursue it.

To **Betar Gallant**, thank you for the gift of sharing this adventure with you through every up, down, and sideways. May we have many more.

Contents

1 Introduction **23**

1.1 Introduction 23

1.2 Problem Statement and Objectives 24

1.2.1 Motivating Problem 24

1.2.2 Scope 25

1.2.3 Objectives 25

1.3 Literature Review 26

1.3.1 Competition Robotics for STEM Education 26

1.3.2 ISS Utilization for Education 28

1.3.3 Simulation and Games for Education 30

1.3.4 Automated Ranking Systems 31

1.3.5 Literature Gaps 32

1.4 Broad Access Platform Design Principles 34

1.5 Approach 35

1.5.1 STEM Outreach Program 36

1.5.2 Zero Robotics Platform 38

1.5.3 Scoring Methods 42

1.5.4 Collision Avoidance Algorithm 42

2 Zero Robotics Tournaments **43**

2.1 Introduction 43

2.1.1 Tournament Nomenclature 43

2.2 Game Design Methodology 44

2.2.1 Game Example: RetroSPHERES 44
2.2.2 Recommended Components 45
2.2.3 Commonly Featured Components 55
2.2.4 Implementing Decentralized Games 57
2.2.5 Game Balancing 60
2.2.6 Game Manual 66
2.3 Tournament Design Methodology 66
2.3.1 Season Timeline Overview 66
2.3.2 Game Design 67
2.3.3 Registration 67
2.3.4 Kickoff 68
2.3.5 Competitions and Game Evolutions 68
2.3.6 Tournament Scoring 69
2.3.7 Alliance Phase 70
2.3.8 ISS Finals 74
2.3.9 Other Zero Robotics Tournaments 77
2.4 Tournament History 78
2.4.1 2009 Pilot 78
2.4.2 2010 SoI 82
2.4.3 ZRHS2010: HelioSPHERES 85
2.4.4 ZRMS2011 and ZRHS2011: AsteroSPHERES 89
2.4.5 ZROC #1 Zero Robotics Autonomous Space Capture challenge 94
2.4.6 ZRHS2012 RetroSPHERES 99
2.4.7 Evaluation 102
2.5 Summary 104

3 The Zero Robotics Platform **107**
3.1 Introduction 107
3.1.1 Contributions of Industry Partners 108
3.1.2 SPHERES Software Architecture 109

3.2 Spheres Simulation History 111
3.2.1 Common Features 111
3.2.2 GSS, C GSP, and MATLAB Simulations 112
3.2.3 SWARM Simulation 115
3.2.4 v2009 MATLAB Engine 119
3.2.5 Overall Lessons 124
3.3 Detailed Design of Current Simulation 126
3.3.1 Top Level Block Diagram Layout 127
3.3.2 Repeatable Seeding of Random Variables 128
3.3.3 Dynamics 130
3.3.4 Sensors 133
3.3.5 SPHERES Software Simulation 141
3.3.6 Dynamic Loader for Satellite Libraries 149
3.3.7 Code Generation Capability 151
3.3.8 SPHERES Code Profiler 151
3.4 Zero Robotics API 152
3.4.1 History 152
3.4.2 Software Architecture 154
3.4.3 User-Facing API Design 157
3.4.4 Internal API Design 166
3.4.5 Catching Common C/C++ Coding Errors 171
3.5 Zero Robotics Simulation and Compilation Interfaces 177
3.5.1 General Architecture 178
3.5.2 2009 Pilot: Downloadable Standalone Simulation 178
3.5.3 2010 Pilot: First Web-Based Simulation Service 183
3.5.4 2011 Onward: Current Design 186
3.5.5 Step-by-Step Simulation Outline 191
3.6 Zero Robotics IDE 193
3.6.1 Graphical IDE 193
3.6.2 Text-Based IDE 196

3.7 Data Analysis Tools 201
3.7.1 3D Visualization 201
3.8 Zero Robotics Website 203
3.9 Summary 204

4 Zero Robotics Scoring Systems 207
4.1 Introduction 207
4.2 Other Zero Robotics Ranking Systems 208
4.2.1 ZRHS 2010 and ZRHS 2011: Round-Robin Competitions . . . 208
4.2.2 ZROC 2012: Relative Scoring Leaderboard 209
4.3 ZRHS 2012: Whole History Rating Leaderboard 211
4.3.1 Overview of the WHR Algorithm 211
4.3.2 Improvements and Implementation Considerations 214
4.3.3 Presentation to Users 222
4.3.4 Case Study: ZRHS 2012 224
4.3.5 Recommendations for Future WHR Competitions 238
4.4 Summary 239

5 Close-Proximity Collision Avoidance for Satellite Game Players 241
5.1 Introduction 241
5.2 Steering Law 242
5.2.1 Relative Kinematics 242
5.2.2 Avoidance Controller 244
5.3 Implementation Considerations 246
5.3.1 Distance Threshold and Time Horizon 246
5.3.2 Distance Target 246
5.3.3 Nominal Controller Override 247
5.3.4 Multiple Satellites 247
5.4 Initial Development ISS Test Session Results 248
5.5 Conclusions 251

6 Conclusions and Future Work **255**

6.1 Thesis Summary 255

6.1.1 Engage and Educate 255

6.1.2 Accessibility 256

6.1.3 Incremental Difficulty 256

6.1.4 Efficient Inquiry 257

6.1.5 Authenticity 257

6.2 Contributions 257

6.3 Future Work 259

6.3.1 Research Directions 260

6.3.2 Monte Carlo Tools for the Zero Robotics Platform 261

6.3.3 Formal Evaluation Studies 261

6.3.4 Scaling Challenges 262

6.3.5 A Development Roadmap For Zero Robotics 263

A Zero Robotics-Specific Implementation Details **265**

A.1 Game Implementation 265

A.1.1 Scoring Systems 265

A.1.2 Code Size Limits 265

A.1.3 Standard Game Phases 266

A.1.4 Communications 270

A.1.5 Game Manual 271

A.1.6 Game Development Timeline 272

A.1.7 Code Preparation for ISS 274

A.2 Simulation Details 274

A.2.1 S-Function Interface 274

A.2.2 S-Function Interface Inputs and Outputs 276

A.2.3 Thruster Transient Modeling 277

B SPHERES Parameters and Uncertainty Quantification **281**

B.1 SPHERES Thruster Geometry 281

B.2 Sources of Uncertainty in ISS Testing 282
B.2.1 Mass Properties 283
B.2.2 Thruster Performance 283
B.2.3 Metrology Errors 287

C A Monte Carlo System for Open-Ended Robustness Analysis 293
C.1 Introduction 293
C.1.1 Motivation 293
C.1.2 Requirements 294
C.2 Monte Carlo Robustness Testing 295
C.2.1 A Note About Parameter Sampling 295
C.2.2 Overview of Method 295
C.2.3 Response Surface Fitting 298
C.2.4 Choosing a Constraint Function 302
C.2.5 Additional Implementation Considerations 303
C.2.6 Multi-Dimensional Data Display with Parallel Coordinates . . 303
C.2.7 Algorithm Summary 304
C.2.8 Phased Deployment to Zero Robotics Platform: 305

List of Figures

1.1 Two Components of Student Robotics Competition Taxonomy 26
1.2 NASA ISS Education Framework 29
1.3 Approach Overview 36
1.4 Software development cycle 38

2.1 ZRHS2012 Game Layout 46
2.2 A Well-Designed Strategy Landscape 63
2.3 2011 Alliance Selection Algorithm 72
2.4 2012 Alliance Selection Method 73
2.5 2012 Final Competition Bracket 76
2.6 Summer of Innovation Game Layout 84
2.7 Satellite Initialization Circle for HelioSPHERES 86
2.8 ZRASCC Capture Zone Positioning 95
2.9 ZRASCC Capture Zone Alignment 95
2.10 ZRASCC Collision Avoidance Region and Avoidance Cone 95
2.11 ZRASCC Capture Area 96
2.12 Strategy Landscape for RetroSPHERES 101
2.13 2012 Subject Area Self-Reported Skill Improvement Results 104

3.1 SPHERES Software Interface 110
3.2 SPHERES Simulation Components 113
3.3 GFLOPS SPHERES Simulator Architecture 114
3.4 SWARM Simulation Architecture 119
3.5 Software-In-the-Loop Implementation for 2009 MATLAB Simulation . 123

3.6 Top Level Layout of SPHERES Simulation 128
3.7 Accelerometer Noise Data 136
3.8 Accelerometer Noise Model Comparison 137
3.9 Global Metrology Timing Diagram 138
3.10 Ultrasound Receiver Geometry 142
3.11 Layers of the Software Model 144
3.12 Thread Synchronization Example 150
3.13 Web-Based Simulation Service Architecture 179
3.14 The 2009 Zero Robotics Simulation Interface 182
3.15 2010 Website Architecture 186
3.16 The 2010 Graphical Editor Prototype 194
3.17 Example of a Waterbear Graphical Editor Program 196
3.18 The Zero Robotics 3D Visualization 202
3.19 The Zero Robotics Report Tool 203

4.1 Effect of Minimum Time Period on Ranking History 220
4.2 Effect of Ranking System Improvements on 3D Competition 221
4.3 Leaderboard Match History View 223
4.4 Leaderboard Histogram View 224
4.5 Percentage of Teams Making at Least N Submissions 227
4.6 Percentage of Teams Submitting At Least Once By Days Before Deadline 227
4.7 Percentage of Total 3D Competition Simulations Run Per Day in 2011 and 2012 228
4.8 Total 3D Competition Simulations Run Per Active Team by Day in 2011 and 2012 229
4.9 The Scaling of Win Percentage by Total Scored Matches 235
4.10 Filtered Ranking History vs. Experienced Ranking History 237

5.1 3-Satellite Test Trajectory for Collision Avoidance 249
5.2 Distance at Closest Point of Approach for Initial ISS Testing 249
5.3 2-satellite Collision Avoidance Trajectory 250

5.4 Planar View of Head-On Collision Avoidance 252
5.5 3-Satellite Collision Avoidance Test 252
5.6 3-Satellite Uncooperative Avoidance 253

A.1 Simplified Model for Transient Thruster Response 279

B.1 ISS Thruster Attenuation Scatter Plot 286
B.2 Simulation Thruster Attenuation Scatter Plot 288
B.3 ISS Thruster Attenuation Time Histories 289
B.4 Initial Positioning Histograms for ISS Matches 290

C.1 Comparison Between Uniform Random Sampling and Quasi-Random Sampling 296
C.2 Illustration of Homothetic Deformation 297
C.3 Parallel Coordinates Example 304

List of Tables

2.2.1 ISS and Zero Robotics Boundary Limits 55

2.3.1 Tournament Timeline 67

2.4.1 Team Participation Information from 2009-2012 103

3.2.1 Simulation History Overview 125

3.3.1 Typical Values for Random Noise Variables 142

4.3.1 Match Outcome Prediction Performance 231

4.3.2 Effect of WHR Improvements on 3D Competition (42235 Matches) . 231

B.2.1 Monte Carlo Parameters and Ranges 291

Nomenclature

AMI Amazon Machine Image

CRTP Curiously Recurring Template Pattern

DSP Digital Signal Processor

EC2 Elastic Compute Cloud

GFLOPS Generalized FLight Operations Processing Simulator

GSP Guest Scientist Program

GSS GFLOPS SPHERES Simulator

IDE Integrated Development Environment

IMU Inertial Measurement Unit

IR Infrared

JPM Japanese Pressurized Module

LCG Linear Congruential Generator

SVM Support Vector Machine

SVR Support Vector Regressor

SWARM Self-assembling Wireless Autonomous Reconfigurable Modules

TLC Target Language Compiler

WHR Whole History Rating

Chapter 1

Introduction

1.1 Introduction

Starting with the rapid change of the American educational system following the launch of Sputnik in 1957, space exploration has a long history as a driving source of educational inspiration. There is growing evidence that we are in need of another transformational change in education to preserve the ingenuity built in the years since Sputnik. The 2005 *Gathering Storm* report [11] brought attention to a looming threat in the form of declining skills in Science, Technology, Engineering, and Math (STEM) among students and the general population. The followup report in 2010 noted a number of actions taken in response to the original report, but classified the storm as having grown to an even more perilous "category 5" [12]. In this work we will examine a platform for using a space research facility for achieving broad educational impact.

Since its inception NASA has worked to promote education, an effort that is of general benefit to society, but also vitally important to an agency whose mission is highly dependent on a skilled workforce. With the completion of the International Space Station (ISS) in 2011, the station has been designated a U.S. National Laboratory, accessible to public, private, and academic institutions for ongoing research. In this new phase of utilization, both NASA and the Center for Advancement of Science in Space (CASIS), the non-profit agency that will control research on the ISS, have

many possibilities to leverage this resource for education.

With the urgent need to raise the math and science proficiency of the American public, innovative solutions to draw students to these topics and keep them engaged are in high demand. Despite the level to which space captivates young students, there are few national programs that use space as a context to engage students in substantive problem-solving challenges. While NASA and many aerospace contractors are heavily involved in funding high impact programs like student robotics competitions and produce large quantities of educational materials, the ISS has primarily been used as a demonstration platform. A driving factor is the relatively limited amount of crew time available and the extremely high cost of sending experiments into space. Achieving broad reach on the order of thousands or tens of thousands of students is only possible through careful allocation of on-orbit resources.

This thesis will present a student robotics competition called Zero Robotics that uses the SPHERES nanosatellites aboard the ISS both as robots and as a motivating tool for STEM education. More generally, the central contribution is a framework for building virtual platforms that serve as surrogates for limited availability hardware facilities.. This system includes high fidelity physics models, control algorithms, and software tools that, applied to SPHERES, have opened the ISS to thousands of young students and members of the general public.

1.2 Problem Statement and Objectives

1.2.1 Motivating Problem

The MIT Space Systems Laboratory's SPHERES facility is compelling enough to engage and educate broad swaths of young people because it involves space, astronauts, and the ISS. The facility includes three volleyball-sized satellites that fly in the habitable volume of the space station as well as three satellites on the ground. At the graduate level, the technology has been proven to be accessible and understandable while at the same time offering a platform for solving technically challenging prob-

lems. In the past seven years of operation, the SSL has used SPHERES to involve dozens of undergraduate and graduate students in unprecedented levels of access to microgravity for experimentation and analysis. The key question to be addressed by this work is: how do we best leverage the ISS, and more specifically, SPHERES for substantive educational impact on a national scale?

1.2.2 Scope

To reasonably constrain the scope, the research in this thesis will primarily examine a model for outreach and education based on a student robotics competition. While there are other ways of engaging students, robotics competitions have been shown [48, 65, 47] to have an effective blend of excitement and difficulty to draw students in, keep them coming back, and along the way deliver many valuable social and technical skills. Furthermore, the inherent structure of a tournament-style competition with a series of down-selection phases is an initial step toward achieving broad reach from a limited resource. For application to the ISS and SPHERES, on-orbit time is only used in the final phase of the competition while the ISS and space still prominently serve to attract students to the competition.

1.2.3 Objectives

There are three main objectives of this research:

1. Identify the best structure for extending the capabilities of a low availability research facility to a wide audience.
2. Identify supporting algorithmic tools for effective use of the infrastructure.
3. Utilize the ISS for substantive, broad-impact outreach.
    (a) Leverage the inherent draw of space and the ISS to create an engaging and educational program.
    (b) Maximize the number of students that get the experience of running an experiment in space.

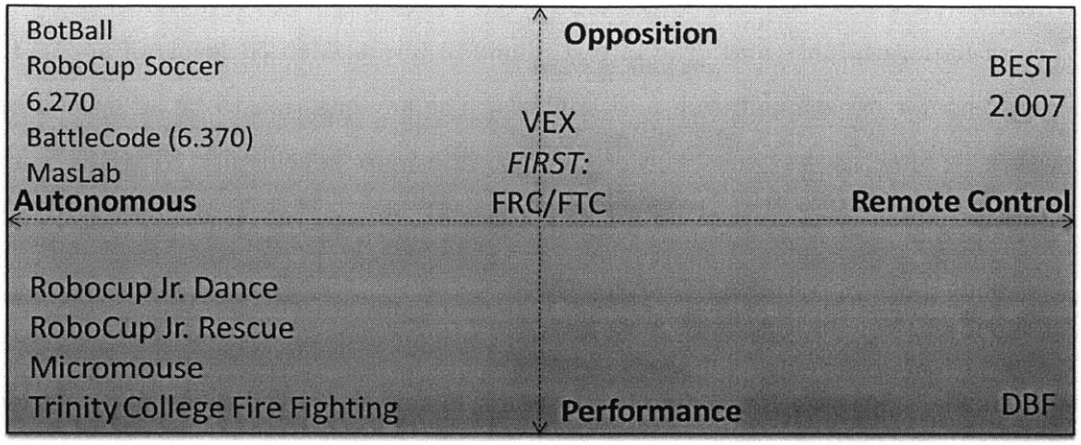


Figure 1.1: Miller, Nourbakhsh, and Siegwart defined a taxonomy for classifying student robotics competitions in [49]. Example student competitions are shown for two of the main components: Autonomous vs. Remote Control, and Opposition vs. Performance.

1.3 Literature Review

1.3.1 Competition Robotics for STEM Education

A significant portion of this study will be devoted to the design, implementation, and operation of a student robotics tournament for educational outreach. The review by Miller, Nourbakhsh, and Siegwart in [49] covers the history of robotics in education as well as the origins of competition robotics as an educational tool. Enhanced interest in engineering and science, improved teamwork, and better problem solving are listed as direct consequences of participating in robotics activities. The authors also include a useful taxonomy for classifying robotics competitions.

Autonomous vs Teleoperated The level of autonomy of the robot, ranging from fully autonomous to purely remote controlled. Teleoperated robotics competitions tend to have more focus on hardware design while autonomous robotics competitions have a focus on algorithms and software, though even most modern teleoperated competitions include some level of programming to configure the robot controller. Examples range from BEST (only teleop) [35] to FIRST (mixed autonomy and operator control) [37] to BotBall (fully autonomous) [36].

Performance vs Opposition Participants either compete head-to-head in matches or they are scored against an absolute performance measure, such as time to completion. With a consistent environment, performance-based competitions can result in more complex strategies, while opposition matches tend to add more excitement.

New Game vs Old Game Some competitions unveil a new game at the start of each season while others have a recurring challenge to be optimized over several seasons. Recurring challenges can also be enhanced with additional features from year to year.

The tournament model has been highly successful for FIRST (For Inspiration and Recognition of Science and Technology) robotics. FIRST began in 1992 and consists of four programs in different age groups that motivate young people to pursue career opportunities in STEM fields. Through a unique combination of hands-on hardware design activities and arena events, the program brings more than 25,000 teams, over 300,000 students, and 100,000 mentors to participate every year. A report by Melchior *et al.* [48] lists many statistics showing clear benefit to students, a small subset of which include:

- "Almost all participants felt FIRST had provided them with the kinds of challenging experiences and positive relationships considered essential for positive youth development
- "Significantly more likely to attend college on a full-time basis than comparison students (88% vs. 53%)
- Overall satisfaction with the program was high. Ninety-five percent of the alumni rated their experience as 'good' or 'excellent' "

In additional to full spectrum studies, another way to evaluate the impact of robotics programs is by examining a measure called *self-efficacy*, the self-confidence in one's ability to perform a specific task [3, 4]. In an academic context, subject-specific

self-efficacy evaluations have been shown to be predictors of academic performance [5].

To see why robotics programs are particularly effective in improving STEM skills, we can look to ways that self-efficacy is influenced. In a study by Lucas *et al.* of undergraduate engineering student internships, one of the most important factors in improving self-efficacy is participation in *authentic* experiences [43]. Students that have an active role in building or creating something that they believe is representative of a realistic engineering task and receive feedback are more likely to improve confidence in their abilities. A second influencing factor is *vicarious experience*, or the opportunity to observe positive behaviors from expert demonstrators. Robotics programs tend to combine both components, immersing students in realistic, challenging problems, under the guidance of expert mentors that can instruct and demonstrate good engineering practices.

1.3.2 ISS Utilization for Education

In [64], we give a brief overview of the current efforts for utilizing the International Space Station for educational outreach. A key component of the plan outlined in the 2006 International Space Station: National Laboratory Education Concept Development Report [39] is a pyramid of activities related to reaching students at various levels of interest (see Figure 1.2). For the broadest reach, materials promoting awareness of space-based activities will be used to *inspire* students, of which a subset will be *engaged* in hands-on activities that make use of NASA and ISS resources. At the top of the pyramid are *educate* activities, which are targeted at specific populations.

While this pipeline of activities makes effective use of limited ISS resources, NASA has had limited success in building broad reach programs that extend beyond the *inspire* level of the pyramid. Through video conferences and on-orbit demonstrations, many students have had the opportunity to get a brief picture of life in space, but they have not had the chance to engage directly with the research. A NASA report by Thomas *et al.* lists fewer than 15 activities in the period of 2000-2006 which have allowed students to engage directly (i.e. send hardware to ISS or affect experiments

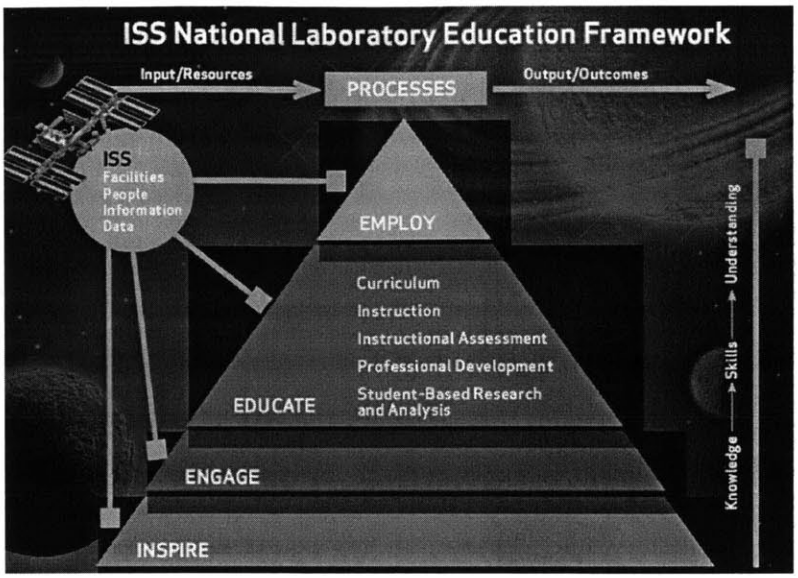


Figure 1.2: NASA's framework for utilizing the ISS for education. Image from Johnson [39].

on the ISS) in space activities.

Part of the problem is the arduous process of deploying new hardware to the ISS. With an extensive design and safety qualification process, it may take years for an experiment to go from concept to final demonstration. The long time scale cuts down the pool of potential participants because a dedicated team must be created to carry out the effort over multiple school years. Furthermore, unless the opportunity to develop the experiment is shared between schools, only a select few have the opportunity to have a hands-on role. Modular facilities such as NanoRacks and the CubeLab standard [44] are tackling this problem by providing a small, standard experiment container with simplified power and data interfaces. The experiments can be easily swapped out by crew members and transported in significant quantities (10-20) to the ISS by manned and unmanned cargo vehicles. These new approaches are making progress in cutting down development time and directly engaging students on the order of thousands of participants. One limitation is the small footprint restricts the exploration of space dynamics and robotics experiments.

In [63], Saenz-Otero enumerated the core principles for creating well-designed microgravity research facilities by examining the history of ISS experiments and the

development of SPHERES. Since an objective of ISS educational outreach is to engage students in authentic research experiences, educational platforms should be built on the same principles if they are not already re-using an existing facility. Of particular importance are:

Principle of Focused Modularity This principle includes the ability to reconfigure the facility to accommodate new experiments. This is very important for educational efforts because many participants can use the same facility without replacing the facility itself.

Principle of Remote Operation and Usability If the facility is designed to be remotely operated by a non-expert user, there is an explicit framework for preparing reliable experiments and conveying qualitative and quantitative results of experiments to researchers.

While it is possible to build an educational platform from an existing research facility, the design principles do not address the additional features necessary for achieving broad access to younger students. For instance, SPHERES, one of the embodiments of the principles, has opened access to tens of graduate students and multiple principal investigators, but until the beginning of Zero Robotics, there was no clear avenue for thousands of students to use the facility.

1.3.3 Simulation and Games for Education

For learning science, Honey et al. [33] introduces a useful picture of a continuum between games and simulations. Simulations are defined as computational models used to clarify or expose processes that would otherwise be difficult to interpret. Games, while often built on some form of simulation, are usually distinguished from simulations by incorporating rules and explicit goals beyond the basic physical laws in the simulation. While games are predominantly focused on casual and enjoyable play, there are many types of serious games where the goals may include self-improvement, learning, or training.

The report also surveys a broad literature base to arrive at several important features that affect learning:

- A clear focus on learning goals
- Scaffolding or support structures to help users gain confidence
- Representations focused on learning goals, not necessarily graphical realism
- Carefully balanced level of user control
- Some form of narrative or motivation for the task
- Detailed feedback about performance
- Adaptive features to cater to different learning abilities

The report concludes there is moderate evidence that simulations can advance learning goals, while for games, the literature is somewhat inconclusive. Nonetheless it notes there is a strong potential for carefully designed simulations and games to have a meaningful impact when paired with specific educational goals.

1.3.4 Automated Ranking Systems

An interesting issue that arises in large scale competitions is the efficient and effective ranking of competitors. Absolute performance measures like scoring systems have to be carefully crafted to give accurate results and not be subject to exploitation by the competitors. An alternative approach used by competitive board game and online gaming communities is the use of a "skill" rating to predict expected match outcomes, then update the skill rating based on the outcome of a game. The classic method used for chess ratings is called the Elo method after Arpad Elo[20], which assumes an average skill and fixed variance of performances around that skill. Elo is a specific form of a generic paired comparison model known as the Bradley-Terry model [6].

A more advanced version class of Elo ranking systems use Bayesian estimation to achieve more accurate updates of the model. An algorithm called *TrueSkill©* is used

by Microsoft in the Xbox Live online gaming community[31]. Instead of carrying only a skill rating, to rank a player, it includes both a mean μ and standard deviation σ, both of which are updated by the outcome of a match. Glickman introduced the idea of a *dynamic* Bradley-Terry model for estimating time varying parameters via paired comparisons in [26]. Coulom created another form of Bayesian skill rating system based on the Bradley-Terry model called Whole History Rating [15], which incorporates information from the full history of match outcomes instead of incremental updates. This work will examine applying the Whole History Rating algorithm to ranking autonomous algorithms created by students in the competition.

1.3.5 Literature Gaps

Under the scope of applying a student robotics competition format to the ISS, the following gaps have been identified.

1.3.5.1 Student Robotics Competitions

The existing structure for many robotics competitions, including an annual season, a scoped challenge or game, and a model built around local mentors to guide teams, can be applied. The main gaps to address are:

Accessibility Many robotics competitions require entry fees and center around hardware-based designs with parts kits. For the higher tier competitions like *FIRST*, the startup, entry fee, and participation costs climb into the tens of thousands of dollars. There are many fewer robotics competitions with a focus on low cost and low startup time.

6DOF At the college level there are several instances of autonomous robot competitions using flying vehicles, but the only 6DOF competitions available to high school students are submarine based [45].

Simulation-Based Robotics While some competitions include experience with CAD-based robot designs, there are no secondary school competitions based on dy-

namic simulations (there are some for college students, such as RoboCup Rescue). More specifically, there are no competitions which prepare in simulation for a hardware-based competition.

Competition Robotics and Space There are no competition robotics programs involving space and the ISS.

1.3.5.2 ISS Outreach Programs

We wish to utilize the already established appeal of space and the ISS for attracting students to the competition while improving upon the following gaps:

Limited Interaction Outreach efforts usually manage to reach many students, but only a small number of finalists in outreach competitions (e.g. 8 total in YouTube Space Lab [72]) get a chance to run experiments in space.

Dynamics Experiments Many of the opportunities for the general public to perform experiments on the ISS are limited to constrained volumes that prevent interaction with microgravity dynamics. From a robotics perspective, this is one of the most compelling aspects of space.

1.3.5.3 Related Questions

Merging the two preceding sections to create a new robotics competition raises the following related questions to be addressed in this work:

- Starting from a brand new architecture for a robotics competition:
  - How do we build a broadly accessible platform usable by thousands or tens of thousands of users?
  - How do we maintain a meaningful tie to the ISS resource it represents?
- With a competition, based on SPHERES, students do not have direct access to the hardware platform:

- What are the robotics?
- How do students learn and interact?
- How do we make it fun and engaging?

- Given the high cost, and more importantly, high expectations of using an ISS resource:
  - What simulation and controls technologies are required to ensure non-expert users have some level of success during the allocated time?
  - How do we maximize the number of students that get to run experiments on the ISS?

1.4 Broad Access Platform Design Principles

At its core, Zero Robotics achieves broad access to the SPHERES facility through an online platform hosting a simulated representation of the satellites and a repeating robotics tournament. The effective implementation of this model is driven by several unifying design principles that extend the laboratory design principles described in [63] and incorporate many of the educational goals described by Honey et al. in [33]. They serve as the framework for answering the questions in the preceding section, and throughout this thesis, will be used as touchstones to generalize the lessons from creating and running the Zero Robotics program. The intent is to contributed both the framework of principles and concrete examples to guide the creation of other outreach efforts based on limited availability research facilities or other difficult to test dynamic robots.

Engage and Educate An effective platform creates an exciting challenge without sacrificing educational value. The platform should strive to draw students in with exciting problems and attractive awards while clearly identifying which skills are intended to be learned by participants and providing ways to acquire them. Feedback

from users and impact evaluations should be a significant guiding factor for improving the platform.

Accessibility An effective platform minimizes barriers to entry. Concerted efforts should be made to minimize entry costs and facilitate access by novice teams.

Incremental Difficulty An effective platform accepts a range of skill levels and progressively challenges all participants.

Efficient Inquiry An effective platform facilitates the process of asking questions and minimizes the time required to supply an answer. The platform should include multiple ways of evaluating performance and providing feedback, some driven by user inquiry and some provided automatically.

Authenticity An effective platform provides an accurate enough surrogate for the resource it represents. Assuming participants don't have access to the real hardware or iterative testing on the hardware is so difficult it impairs *Efficient Inquiry*, the platform should provide a model on which experiments can be performed. As noted in [43], *Authenticity*, is also a key factor in building self-efficacy In Zero Robotics, while only a limited number of teams can participate in the final championship, all teams use the online simulation environment to test programs and compete. Making the simulation tools as realistic as possible will improve the satisfaction with competing in the tournament and prepare finalist code for hardware testing.

1.5 Approach

The full approach is summarized in Figure 1.3 along with major elements. The following sections will elaborate each of these components.

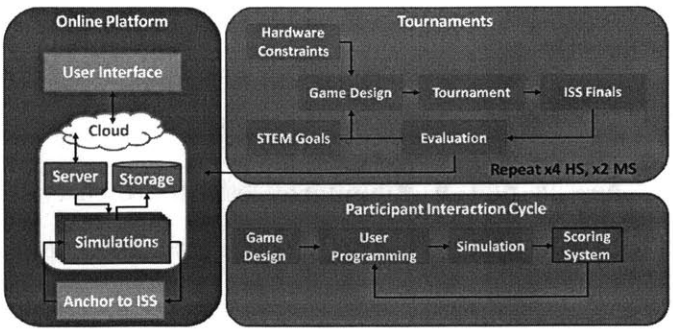


Figure 1.3: There are three main components of this study: an outreach tournament for students and a user interaction cycle, both tied together by an online platform.

1.5.1 STEM Outreach Program

1.5.1.1 Overview

Chapter 2 presents a structure for annual, nationwide tournaments to extend the experience of operating SPHERES on the ISS to high school and middle school students around the country. The main contribution of the chapter is a progressively challenging tournament structure for 6DOF robotics competitions, including a unique multi-week collaboration phase. Students investigate the physics of satellite motion, learn to program the satellites, and fine-tune their algorithms, all while vying for a place in the championship rounds that take place live from the ISS.

Successful robotics programs like *FIRST* have demonstrated incredible power to enhance student skills by directly involving them in cross-disciplinary problem-solving challenges and surrounding them with knowledgeable mentors to guide their solutions. Zero Robotics follows a similar approach, structuring the competition season around a challenging technical problem to solve, and the same mentor-based team structure. As a primarily software simulation competition for programming satellites, there are also several important differences:

1. The engineering process is entirely *model-based.* Students implement and test their solutions on virtual models before taking them to real hardware. This process parallels model-based design processes followed by engineers for many aerospace systems where complicated dynamics and a high cost of demonstra-

tion missions necessitate simulated study to achieve a high probability of success. This approach is not meant to compete with hands-on hardware competitions, rather it teaches a complementary skill set.[1]

2. There is a heavy focus on applying *math and physics.* On a day to day basis, students must program solutions using vector math, trigonometry, and basic calculus. The simulation environment can be used at a basic level to verify fundamental principles such as $F = ma$ and at the same time demonstrate non-idealities due to sensor noise and thruster disturbances.

3. Most if not all other robotics competitions use 2D drivable robots with three degrees of freedom. In many competitions, the drive systems are also heavily geared so dynamics are not as important in autonomous control. In Zero Robotics, the robots have second order dynamics and move with full motion in six degrees of freedom. This is particularly interesting because it introduces the challenge of learning about and controlling 3D rotations, a topic not typically covered in high school curricula.

In the taxonomy introduced in [49], Zero Robotics is a purely *autonomous,* head-to-head *opposition* competition, with a *new game* each year.

As a software competition, Zero Robotics also crosses over into the world of simulation and games. It sits in the middle of the spectrum described in [33]. As a simulation, the tools are quite advanced. Users have full control over the satellite's motion, and the simulation environment implements complex, accurate physics. However, in comparison to commercial games, the interaction is quite limited, the graphics are simple, and the development cycle shown in Figure 1.4 introduces a delay between creating a program and seeing the results. Nonetheless, as evidenced by the dedication of teams to participating in our initial seasons, the combination of the competition and the programming environment are sufficient to keep students engaged. This is an interesting development because combining a competition game

[1] A long-term goal of Zero Robotics should be to bring hardware and software worlds together. Students would then have an opportunity to model and simulate systems they have designed on their own.

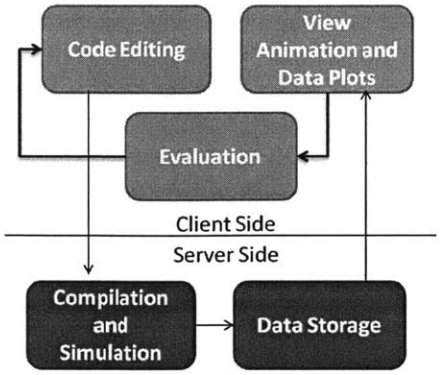


Figure 1.4: A typical software development cycle for a Zero Robotics user.

with a simulation allows us to invert a common pattern in games for education where learning effectiveness is sometimes sacrificed in favor of maintaining interest. While Zero Robotics games often contain some fictionalized game components to enhance their appeal, every step toward solving the challenge requires students to engage with fundamental concepts in programming, math, and science *before* they see the exciting results.

Achieving Broad Reach: Zero Robotics Tournaments Zero Robotics depends on a competitive structure to leverage the limited availability of time available to perform tests in space. A tournament format is a natural template for achieving all levels of the pyramid shape in Figure 1.2 because a broad base of competitors are drawn by the opportunity to make it to the top, and the narrowing of teams due to down-selection allows for a target experience (ISS championship) in the final stages.

Development for a Zero Robotics high school tournament consists of a series of successive phases that span the entire year. The tournament itself lasts from September to December with the ISS finals taking place at the end of December or in early January.

1.5.2 Zero Robotics Platform

Chapter 3 presents the central enabling component of Zero Robotics, an online platform for community building and hosting the annual tournament. The web environ-

ment consists of five primary modules:

1. A community site for hosting challenges, learning, sharing ideas, and tracking progress

2. A high fidelity SPHERES simulation for running program simulations

3. A simplified Application Programming Interface (API) to provide easy access to satellite controls and sensors

4. An Integrated Development Environment (IDE) for programming the SPHERES satellites

5. A visualization front end for viewing the results of simulations

These components address the principles of *Engage and Educate* by providing a tool for exploration of math, physics, and programming concepts; *Efficient Inquiry* by accelerating the process of writing and evaluating programs; *Accessibility* by making the tools easily available online, and *Incremental Difficulty* through a variety of ways to program the satellites. Combined, the components contribute a web-based architecture for writing, compiling, and simulating code for a dynamic robot.

1.5.2.1 SPHERES Simulation Back End

The ability to test and optimize code in simulation is critical to achieving reliable test sessions, and this is especially important with student-developed code. For development on a simulation platform to be possible it is important that the system as thoroughly as possible model:

1. The dynamics of the system. In the case of SPHERES, the entire system is accurately modeled down to the level of individual thruster firings and variations in thrust due to multiple thrusters being opened.

2. Sources of noise. In many cases it is either not possible or not worth the computational effort to model small variations from the basic model of the

system. In these cases, the simulation must appropriately compensate with additive noise to represent uncertainty in the dynamic model.

3. The operation of the onboard software and the way it interacts with the hardware system. The Zero Robotics simulation models the SPHERES software down to millisecond ticks of the internal clock, sufficient to model the fundamental time cycle of the internal software.

Instead of working to create a downloadable simulation package, where it must be either very compact for downloading at each use or pre-installed on the user's computer, it is advantageous to make the simulation run as a service on a web server that communicates with a user's web browser. As a service, the simulation can be easily upgraded without requiring redeployment. Users don't have to worry about configuring software to compile their code, and the data from simulation runs can be archived for later analysis. There is also no installation or startup time involved in the development cycle; users just log on to the website and begin programming.

Several enhancements to the compilation and simulation system help to ensure students produce successful code for hardware testing. SPHERES has an extremely limited amount of program memory for storing programs, so teams are given a tightly controlled allocation of space to use. Code is compiled by the exact Texas Instruments compiler used to build code for SPHERES, providing an accurate estimate of the total consumption of code space used by the program. During simulation, the user code is checked for illegal memory access outside of array boundaries, and other runtime checks help prevent errors during testing on the ISS. Integrated together these steps contribute a pattern for fast, highly detailed, web-based simulations.

1.5.2.2 Zero Robotics API

For the wide range of skill sets that will be interacting with the Zero Robotics programming environment, it is important to provide an a skill-appropriate set of tools for commanding the SPHERES. The Zero Robotics API contributes a set of commands that simplify the process of reading a satellite's positioning information

and commanding motion. For example, the API provides simplified commands like `setPositionTarget()` to instruct the satellite to move to a 3D position in the test volume. The API also includes a reduced attitude representation using a 3D unit vector as a pointing direction, allowing users to connect math they will already be learning for position control with attitude control. More advanced users can choose to access lower level functionality such as commanding the satellite with forces and torques or specifying attitude with quaternions.

The ZR API is also intended to be useful for Zero Robotics game developers. With multiple seasons to draw on as examples, common functions needed for implementing any Zero Robotics game have been folded into the API, and a framework has been established so developers can focus specifically on implementing the game logic, not reproducing utilities.

1.5.2.3 Integrated Development Environment

Daily interaction with the Zero Robotics website revolves around an online Integrated Development Environment (IDE). To be compatible with the SPHERES hardware, user programs are ultimately written in C++, but the environment offers two ways to create programs: a traditional text editor, and a block diagram graphical editor that converts to C++.

1.5.2.4 Visualization Tools

The visualization front end provides an animated 3D visual representation of the trajectories and data returned from the output of the simulation service. Users can control playback speed, change perspectives, and view game-specific scoring data. Information can also be plotted in a series of 2D line charts. Like the rest of the Zero Robotics tools, both components are accessed from a web browser.

1.5.3 Scoring Methods

Chapter 4 briefly reviews the history of scoring methods used for the Zero Robotics platform, then presents the most recent implementation of a continuous scoring system based on the Whole History Rating Algorithm [15]. A main contribution of this thesis is the application of the algorithm to improve student interaction with the website under the principle of *Efficient Inquiry*. This includes improvements to the algorithm for better stability and faster responsiveness along with experimental results from the 2012 season.

1.5.4 Collision Avoidance Algorithm

Chapter 5 contributes a low-level control algorithm for preventing collisions between players in Zero Robotics games. Originally developed for SPHERES as an always-on supervisory guard in close proximity formation flight, it has been part of all Zero Robotics challenges. The method is based on predicting the future closest point of approach of two vehicles and performing a correction maneuver if the trajectories will travel too close together. The implementation is lightweight and ideally suited for the computationally constrained environment of Zero Robotics and SPHERES.

Chapter 2

Zero Robotics Tournaments

2.1 Introduction

Zero Robotics tournaments are the fundamental structure for extending the experience of using the SPHERES satellite research facility to thousands of students. Each tournament consists of a scoped challenge, or *game*, designed prior to the start of the season, and a series of individual simulated competitions to select finalists for the championship round aboard the ISS.

This chapter begins by covering the design methodology for Zero Robotics games, specifically the components necessary for a game that moves between simulation and hardware. The next part examines the structure of the tournament season and the considerations for creating a fair and enjoyable experience. The final sections present an overview of all Zero Robotics tournaments to date and their associated design lessons.

2.1.1 Tournament Nomenclature

Throughout this discussion, a specific meaning has been assigned to the following terms:

Game The challenge created for each tournament season

Competition A single scored event during the tournament season

Tournament A collection of competitions that form the annual season based on a single game design.

Alliance A group of teams that collaboratively produce a single player in the latter part of the tournament season.

Tournaments are named according to the following guidelines:

ZRHSYYYY Zero Robotics High School Tournament

ZRMSYYYY Zero Robotics Middle School Tournament

ZROC# Zero Robotics Open Challenge

2.2 Game Design Methodology

The design of engaging and challenging games has been studied extensively as evidenced by the existence of several entire industries predicated on producing them. Zero Robotics has the unique situation of transferring a game that takes place mostly in a fictional environment to a real hardware platform. In keeping with principle of *Authenticity*, many of the design constraints of Zero Robotics games are dominated by the hardware requirements. Having completed the design of six separate challenges, the Zero Robotics program has accumulated enough experience to establish guidelines for creating tournament games. This section is not intended to supplant formal or informal methods of game design, many of which are directly applicable. Rather, the goal is to complement them with considerations for producing an exciting and educational virtual challenge while respecting constraints of a hardware platform.

2.2.1 Game Example: RetroSPHERES

To frame these guidelines with a motivating example, we will start with a brief overview of the 2012 high school tournament and the game RetroSPHERES. The

remaining notes about the game design will be provided in each of the following sections as examples of the design methodology, and the lessons from the design of other tournaments will be included in Section 2.4.

The 2012 game was motivated by the problem of cleaning up orbital space debris. Players competed by writing programs for fictional RetroSPHERES, specialized SPHERES satellites designed to de-orbit debris by releasing clouds of dust. The satellites raced through a virtual obstacle course divided into three zones shown in Figure 2.1. In the first zone, both SPHERES had the opportunity to release one or more dust clouds, produced by spinning in place. The second zone was a shared region containing virtual power-up items. The optional supply packs contained extra fuel and supplies to help reach the finish line. As a checkpoint in the mission, players were required to visit a virtual disabled satellite before proceeding into the the last zone. In the final zone, the two SPHERES switched sides and attempted to detect the dust clouds deployed by the other player while navigating to reach the finish line. Entering a dust cloud resulted in a significant reduction in velocity, consuming additional time and fuel. Players could use charge (obtained in the re-supply packs) to boost their ability to identify the location and size of a dust cloud or to shrink a dust cloud. The fastest player with the most fuel at the end of the match was declared the winner.

2.2.2 Recommended Components

This section will review the features of Zero Robotics games that have been present in all seasons or have been added based on lessons learned during the seasons. While not all of the items are strictly required, the following components are strongly recommended for all games.

2.2.2.1 Number of Satellites

All Zero Robotics games to date have been played with two satellites. The choice is firm enough that much of the online platform has been structured with the assumption that two satellites will always be used. Without significant re-design of the Zero

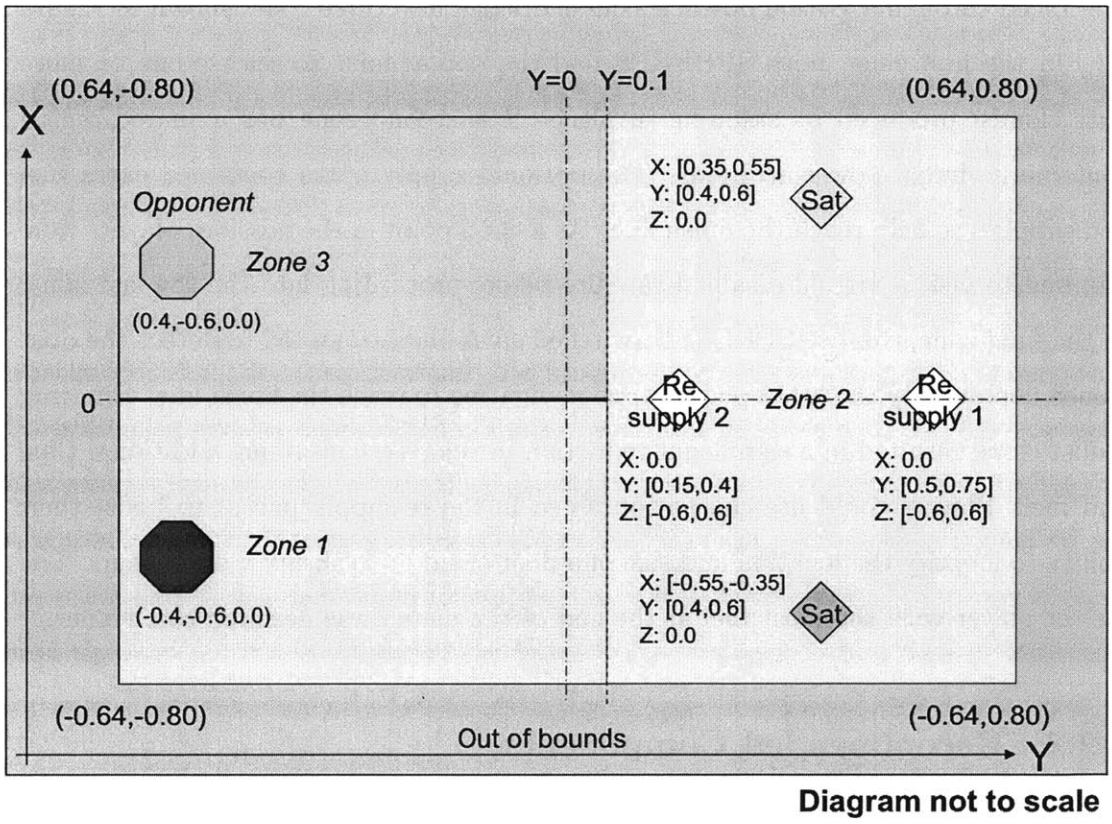


Figure 2.1: In the 2012 game RetroSPHERES, satellites started on opposite sides of the Y-Z plane and moved through 3 zones. In Zone 1, players created obstacles, moved to Zone 2 to pick up special Re-Supply Packs, then navigated through the opponent's obstacle field in Zone 3.

Robotics platform, two satellites will be required in all games.

It is important to note that there are several compelling ideas for other configurations. With three satellites, interesting tournament structures could be created to allow more teams to participate in the ISS finals. Games could also include more complicated interactions where teams have the possibility of briefly joining forces during a match. The third satellite could even behave as an independent actor to disrupt the game play.

Nonetheless, there are strong reasons for choosing to restrict the game to two players. Of primary concern is the ISS competition, where the game must run on the SPHERES hardware. During any given test run, the satellites consume battery charge and CO_2 propellant, and may deplete these resources at any point during the session. The propellant tanks are not replaced at the beginning of the session, so it is possible a tank may have be partially consumed before the competition starts. Satellites may also reset occasionally due to infrared interference. Combined, these factors result in a historical reliability of approximately 70%-90% for each test. All satellites must complete the test run to retrieve scoring information, so adding a third satellite can significantly increase the probability of a test failure if the individual satellite reliability is low. Other operational costs include the time necessary to bootload the third satellite, the added complexity of positioning three satellites at the beginning of the test run, and additional consumable changeout time.

To generalize these observations, for a competition involving n robots, assuming the probability of failure is independent of other robots

$$P(\text{match success}) = (1 - P(\text{sat fail}))^n . \tag{2.1}$$

The dependence on reliability introduces a trade between running competitions efficiently and increasing participation. Hardware competitions with high robot reliability can afford to involve more players in each match, while lower reliability competitions must decrease players unless time is available for match re-runs. Applying the principle of *Authenticity*, the virtual environment should keep the same number

of robots as the final hardware competition. This restriction will ease the transition between virtual and hardware phases since the game software will not have to be modified and re-tested for a different set of competitors.

In the Zero Robotics tournaments, two player games are also advantageous for the scoring system because they clearly distinguish the relative performance between the competitors through a win-loss outcome. In contrast, single player games can be judged against a common performance benchmark but lack the excitement of head to head competition, while multiplayer games enhance excitement at the cost of complex scoring methods. More detailed match scoring methods are covered in Chapter 4.

2.2.2.2 Symmetric Play

For all virtual and ISS competitions, players may be assigned to any satellite with no guarantee of playing the same number of matches in a satellite role. It is therefore essential that the game is symmetric to all players. Symmetry is typically achieved by making players equidistant from important game features such as items or power-ups at the beginning of the simulation. For example, in the 2012 game RetroSPHERES, players started on opposite sides of the Y-Z plane and moved towards "Supply Packs" located on the Y-Z plane (see Figure 2.1). Each pack could be reached by either player in the same amount of time.

2.2.2.3 Multiple Winning Strategies

Much of the excitement of competing in a Zero Robotics challenge comes from creating an innovative solution to the game. If a game has an obvious optimal solution strategy, it is very likely that the competing teams will discover it and the entire competition will converge to a single behavior. In [68], Sylvester labels this a *degenerate* strategy. This outcome can be both dull and frustrating as games with a single solution tend to be determined by small random variations in the simulation or slight differences in implementation.

The same experience can result from a game that is perfectly balanced with many possible strategies. Sylvester supplies the example of a 5-way Rock-Paper-Scissors-

(Spock-Lizard) game. Every one of the many potential options is exactly balanced by another strategy[68], but in truly simultaneous play, winning, losing, or ending in a draw is determined by random chance.

Ideally, in keeping with the principle of *Incremental Difficulty*, potential strategy options should span a range of skill levels with a higher payoff for greater skill. Ensuring the existence of multiple, interesting strategy options is part of the game balancing process covered in Section 2.2.5.

In the example game, RetroSPHERES multiple winning strategies were available with different paths to completing the race. Players could spend more time deploying dust clouds to make an opponent's navigation through the obstacle field more challenging, rush to complete the race very quickly and sneak past the opponent's clouds under creation, or focus on a mixed strategy of dust clouds and item retrieval. This approach successfully offered multiple winning options, but the rewards were very similar for all strategies. More details are discussed in Section 2.4.

2.2.2.4 Visual Elements

Much of the Zero Robotics competition takes place in an online environment where rich data visualization is possible during simulation playback, but the final ISS competition takes place using only the satellite hardware. The only visual feedback while a match runs is the motion of the satellites. As a result, it is necessary to make any virtual components of the game observable in some way through the satellite behavior. SPHERES have strong control authority over rotation, so indicators with short time spans tend to involve a change to the satellite rotation. For example, the completion of a game is usually indicated by inducing a rotation about one of the satellite axes. Velocity changes can be used to indicate spatial transitions such as running into an obstacle or a virtual wall.

When adding motion visualization to a game, it is usually better to make the desired behavior part of the game rules. In the 2011 game AsteroSPHERES, users retrieved special bonus items from the center of the playing volume by moving to the item locations. The items were retrieved by slowing below a specified velocity

limit, after which an external torque was applied to show a slight rotation. The item locations were clear from the satellite trajectories, but the unexpected rotational disturbance was viewed as an annoyance, especially if it disrupted a subsequent re-orientation maneuver. With the user controller occasionally fighting the disturbance torques the additional motion was sometimes difficult to observe in the ISS test session. For the following season, in RetroSPHERES, item pickup changed to requiring the users to perform a maneuver (turn at least 90 degrees from a starting condition) resulting in much clearer indicators and fewer complaints. Other visual elements in RetroSPHERES included reorienting the satellite to deploy a dust cloud, scanning for obstacles by pointing the satellite in different directions, slowing down when passing through obstacles, and performing a right angle trajectory turn and spin at the end of the match to signal completion.

2.2.2.5 No Ties and Scoring Continuity

All games should result in a win-loss outcome without the possibility of ties. During the highly time-constrained ISS final competition, all efforts must be made to avoid match replays. Constraining the game to produce a win or a loss guarantees a score will be available for a match, and replays can be saved for operational problems like exhaustion of consumables instead of breaking ties.

Eliminating ties is also important for the virtual component of the tournament. During the online competition phases, the Leaderboard scoring system introduced in 2012 ranks players by win-loss outcomes. While it is possible to incorporate penalties for ties into the scoring system or explicitly account for tie outcomes, results from the 2012 tournament indicate that it is best to avoid ties unless the likelihood of ties can be carefully modeled. See Chapter 4 for more details.

Another important consideration in the scoring design is the balance between continuous scoring values and one-shot bonuses. Games can be easier to understand and strategize for if the scoring system has a gradually changing, continuous score, especially if the value is monotonically increasing. On the other hand, discrete jumps, or bonuses can add significant excitement to the gameplay. Discrete jumps are best

saved for a high intensity moment or a difficult to achieve objective, while continuous scores are useful for scoring overall performance. For example, AsteroSPHERES awarded most of the match points for performing the main mission objective, but included additional bonus points for winning a finale race at the end of the game.

2.2.2.6 Code Size, Fuel, and Time Limits

Following the principle of *Authenticity*, games must adhere to several constraints imposed by the SPHERES hardware and the final ISS competition. First, Zero Robotics user programs, game implementation code, and the SPHERES operating system all share a flash memory space of approximately 230KB, of which approximately 64KB is available to be divided among all 9 user programs for a typical game. Game designs are implemented with careful monitoring of the code occupied by the game implementation. In some cases it is necessary to sacrifice game enhancements in favor of preserving space for the user implementation.

For general competitions following the Zero Robotics model, code size restrictions are not likely to play a strong role because storage space for robotics hardware has dramatically improved since SPHERES was deployed to ISS. However, from a pedagogical view under the principle of *Engage and Educate*, program size restrictions can encourage careful attention to what is truly necessary to include in the program. This skill is still relevant for development of embedded systems and increasingly so for modern web applications where entire micro-frameworks are transmitted when a user loads a web page.

To preserve consumable resources like propellant and batteries, the game design can include virtual limits as part of the game rules. Zero Robotics games typically have an upper fuel consumption limit of about 10% of a full propellant tank, or about 50 thruster-seconds of thruster firing time. Instead of a hard limit, the resource restrictions can also be incorporated as part of the game scoring system. In RetroSPHERES, teams were scored by the amount of propellant remaining when crossing the finish line to emphasize efficient motion.

When utilizing a remote laboratory like the ISS for a championship competition,

the time available may be tightly constrained. For Zero Robotics a full competition must fit into a specific block of time allocated for running the ISS finals, usually with no more than 2 hours of actual testing time. The main game design decision related to this constraint is choosing the duration of matches. Matches that are too short do not give the competitors enough time to perform meaningful actions in the game, while matches that are too long consume valuable time and can become tiresome to watch and analyze (*Efficient Inquiry*). For Zero Robotics, matches are usually 3 minutes, which gives enough time to traverse the volume in both directions with several additional actions along the way. With deployment of the satellites in the work volume, initial positioning, game time, and transition time between tests, a 3 minute match takes about 6 minutes per test run. This translates to about 20-24 maximum tests in a test session, including replays of failed matches. The overall number of matches available affects the format of the final competition, discussed in Section 2.3.8.

2.2.2.7 Collision Avoidance

While collisions between the hardware satellites will generally not cause damage, they can cause significant perturbations to the state estimation system, sometimes resulting in divergence of the state estimate. More importantly, collision dynamics are not modeled in the SPHERES simulation. In the virtual environment it is possible to produce unrealistic behaviors such as passing through an opponent's satellite. Unless the game specifically requires contact between the satellites, such as the docking demonstration during the Autonomous Space Capture Challenge, some means of preventing collisions should be implemented to adhere to the principle of *Authenticity.*

All games except the Capture Challenge have used the algorithm covered in Chapter 5, which runs as an always-on supervisory control layer to interrupt the user's program if a potential collision is detected. Collision avoidance can also be used to ensure exclusive physical access to a shared resource like an item pick-up. In Retro-SPHERES, the two shared re-supply items could not be picked up at the same time because collision avoidance prevented satellites from occupying the same space.

The algorithm can be incorporated into the game's scoring system to award or penalize collision events. When layering game rules on top of collision avoidance game designers must be wary of forcing the users to avoid the avoidance system as it may result in overly conservative trajectories. In all applications the user should have a way of knowing that the avoidance algorithm has activated on the previous control cycle so additional corrective action may be taken if necessary.

Adding an avoidance system protects against collision events, but it may also introduce uncertainties in the simulation and ISS test outcomes. Small changes in the initial conditions going into a avoidance event can lead to significantly different outcomes, and there may be situations where the algorithm activates in one simulation but not another due to random variations in the satellite trajectories. Though these situations are complex, they are still highly preferable to losing control of the vehicle. As an aid to analyzing collision avoidance scenarios, under the feedback side of *Efficient Inquiry*, users should have clear indications that the algorithm is active in both the game API and in the 3D game visualization.

2.2.2.8 Boundary Limits

Most robotics competitions are constrained to occur within a defined field of play, sometimes limited by a hard boundary such as a wall, or by a soft boundary such as a penalty for crossing the outer limit. For Zero Robotics, the playing field is physically limited to a roughly 2 m cube in the Japanese Pressurized Module (JPM), but collisions with the wall can disrupt the state estimate and must be carefully avoided. To give a clear indication of this constraint in the virtual competitions and prevent users from crashing into the walls on the ISS, the game usually implements a boundary limit in software. Like the collision avoidance system, the boundary limits impose a game constraint to prevent a potentially problematic behavior from occurring.

Two types of boundary limits have been used in competitions to date. The first is an active limit that partially overrides the user's controller. Instead of guiding the satellite back into the volume, the limit only attempts to prevent the user from

colliding with the wall. Assuming the boundary limits are specified with a global direction $\mathbf{e} = \begin{bmatrix} e_x & e_y & e_z \end{bmatrix}^T$, the following operations are applied to each component i of the user's force vector $\mathbf{f} = \begin{bmatrix} f_x & f_y & f_z \end{bmatrix}^T$ when they leave the boundary:

$$f_i = \begin{cases} f_i & f_i \cdot e_i < 0 \\ 0 & f_i \cdot e_i \geq 0 \end{cases}, i = \{x, y, z\} \tag{2.2}$$

$$f_i = \begin{cases} f_i & v_i \cdot e_i < 0 \\ f_i - K_d v_i & v_i \cdot e_i \geq 0 \end{cases}, i = \{x, y, z\}. \tag{2.3}$$

Equation 2.2 nulls any forces directed along the boundary limit, preventing an out-of-bounds player from continuing to accelerate away from the volume. Equation 2.3 applies a velocity controller to the forces to slow motion out of the volume. After the limits are applied, it is still the responsibility of the user to guide their satellite back into the volume.

Additional incentive to return to the volume can be applied by the second type of boundary limit: a scoring penalty for leaving the volume. Scoring penalties should be large relative to the total score, but not catastrophic if the user briefly exits the volume. One way to achieve this is to apply a penalty based on the total time the boundary conditions are violated.

Table 2.2.1 contains the boundary limits based on data from SPHERES Test Session 22, where the wall locations were determined by slowly moving the satellite until it collided with a wall or exited the volume in the indicated direction. The limits usually include a buffer region to implement the bounding behavior. Example values are also shown in Table 2.2.1, but are sometimes adjusted on a game-by-game basis after analysis of the boundary limit behavior. Note that the boundary limits are the same as those imposed in the RetroSPHERES game example in Figure 2.1.

Table 2.2.1: The boundary limits for Zero Robotics games should include a buffer around the wall limits to implement the boundary limit behavior. The recommended value is about $20\,cm$ on each side, but additional analysis may relax this limit.

Direction	Wall Limit	Recommended
x	$\pm 0.85\,m$	$\pm 0.64\,m$
y	$\pm 1.0\,m$	$\pm 0.8\,m$
z	$\pm 0.85\,m$	$\pm 0.64\,m$

2.2.2.9 Space Theme

In addition to running on satellites flying in microgravity aboard the ISS, Zero Robotics games usually draw upon a realistic motivation from space research as a theme. It is not essential to determine the theme at the beginning of the game design process, and in general the theme should not constrain the possibilities for interesting game dynamics. Nonetheless choosing a theme can supply interesting ideas for game components or behaviors. Like good science fiction, imagining a compelling scenario can stimulate creative, realistic implementations.

2.2.3 Commonly Featured Components

The following sections describe components of games that have been used in many or all competitions. They are provided as examples of interesting features that can be added to games.

2.2.3.1 Items

Many games have included special items to be picked up by the satellites from designated locations in the volume. An item retrieval usually involves moving to a location, then meeting a set of motion requirements to acquire the item. Items can be optional "power-ups" that add enhancements to the satellite's capabilities or required checkpoints. In RetroSPHERES, the two shared items in the middle of the playing field awarded different levels of virtual fuel and virtual charge and each could only be acquired by one of the satellites. The other items located outside of the Y-Z plane had to be obtained before proceeding to the final phase. Adding items to a game helps to

improve the visual features of the game because it is usually easy to recognize when an item is being acquired.

2.2.3.2 Randomization

Nearly all Zero Robotics games to date have incorporated an element of randomness in the challenge:

- HelioSPHERES: Random starting locations of the satellites and random location of a virtual solar panel.
- AsteroSPHERES: Randomized orientation of the asteroid competitors circled around or drilled on.
- RetroSPHERES: Placed power-up items in random, symmetric locations in a shared zone.

Random item locations or initial positions, help to emphasize strengths and weaknesses in the player programs by forcing the users to try different scenarios. With a wide variety of possibilities, users are encouraged to build more generalized approaches to solving the associated programming challenges, leading to better modularization of programs.

Random behaviors are also helpful for the Leaderboard scoring system described in Chapter 4 because the scoring algorithm assumes players win or lose with a certain probability based on their program's skill at solving the challenge. Instead of supplying many duplicate scenarios to the system, randomized challenges give the algorithm a more accurate picture of the program's skill in many different scenarios.

2.2.3.3 Endgame Finale

While it is perfectly valid to construct a game with a simple continuous scoring system that gradually accrues over the course of a game, adding a last-minute, high stakes action, or *finale*, at the end of the match can greatly improve the overall excitement of participating in and viewing a tournament. For maximum excitement,

the finale should ideally be both high value and difficult to solve. Unfortunately, heavily weighting a challenging problem can make the match results less repeatable if the solution is affected by random variations. The best designed finales should admit robust implementations, but pose a significant problem to write as computer program.

The finale can be a normal part of the game if the overall challenge is structured like a race. For the 2010 Summer of Innovation tournament, players raced from one end of the test volume to the other, and the first player to cross the line won the match. A race component can also be appended to a game with a different structure, like in AsteroSPHERES where a brief race at the end of the match awarded additional bonus points. The RetroSPHERES finish maneuver was similar to the Summer of Innovation race with a right angle turn at the finish to clearly indicate the satellites approaching the end of the match.

2.2.4 Implementing Decentralized Games

In contrast to other robotics competitions, a unique aspect of the Zero Robotics architecture is the fully decentralized nature of the game management software. SPHERES was originally designed to have a distributed processing system in which the ground station laptop primarily starts tests and initiates cycles of the time-division communication method. Beyond this basic synchronization method, all other game updates must be tracked independently on the individual robots. This section discusses and addresses several of the challenges introduced by the decentralized architecture.

2.2.4.1 Choosing Data to Transmit

If the players have any interactions in the game, such as the exclusive item pickups or user-defined obstacles in RetroSPHERES, a mechanism for sharing game data between satellites is required. Zero Robotics uses the SPHERES RF communication system which presents the additional difficulty of extremely limited bandwidth. All shared game information is limited to a maximum effective throughput of approxi-

mately 480 Bps. A recommended practice for choosing the data to transmit under the principle of *Authenticity* is to use only telemetry data to construct visualizations of simulation results. If a game feature cannot be visualized with the available data it is likely that additional information should be added to the telemetry packets.

In addition to game-specific information, it is highly recommended that all games transmit at least the following items:

- Commanded forces and torques sent to the actuators. These values are the lowest level representation of the commands requested by the users and can be used in a wide variety of analysis scenarios.
- Current score. Having a real time picture of the score is useful feedback in the visualization and can be used for debugging problems in the scoring system.
- Flag to indicate status of the collision avoidance algorithm (if used). Can be used to clearly display collision avoidance events in the visualization and for analysis purposes.

If a vacant space in the telemetry packets remains, adding a data version identifier can be helpful in case the format of the packets changes mid-season. When performing data analysis, the version identifier can be used to properly parse the values. In general, it is not a good idea to re-arrange the telemetry unless absolutely necessary because it complicates post-processing and season-wide data analysis.

2.2.4.2 Delay and Fault Tolerance

Game information packets transmitted during a control cycle cannot be acted upon until the following game update cycle. For Zero Robotics this means that any information that affects the behavior of the opponent satellite will have at best a 1 second delay. Furthermore, the SPHERES RF communication system may occasionally drop packets. All game implementations must assume both the delay and the unreliability of data transfer[1].

[1] If bandwidth allows, an acknowledged packet transmission system can greatly simplify the implementations described here.

A simple strategy for maintaining synchronization between satellites is to transmit all key features of the game state at each time step. If intermediate packets are lost, all data can be reconstructed by both satellites at a future time step. This approach greatly simplifies the implementation of animations based on the game telemetry because the visualization can be *stateless*. At any point in time, enough information is available to completely render a view of the game without keeping track of the game history.

One feature that is readily described using a continuously transmitted state is a synchronized event time. When a satellite achieves an objective, it transmits the game time of completion in a telemetry packet. If the objective can only be accomplished by one of the two satellites, such as picking up an exclusive power-up or winning a race, the time stamp can be used to decide which satellite achieved the objective first. In these situations it is important to make users aware of the possibility that due to time delays, the API may briefly indicate that the objective was achieved, then indicate otherwise.

In complicated games, compressing the entire game state into the available bandwidth can prove challenging. An alternate approach is to implement a partially stateless telemetry scheme. Instead of repeatedly transmitting the entire game state in every cycle, part of the information can be spread out and repeated over multiple time steps. In other words, it is slowly *streamed* with repetitions to ensure delivery. This method induces a lag in the updates because the full game state takes multiple cycles to transmit, and if a packet is dropped in the middle, it may take several cycles to restore.

Streaming information is best applied when the component of the state being distributed does not affect the game until later. For example, in RetroSPHERES, users could create up to 10 virtual obstacles with unique sizes and locations, far more information than could fit into a single set of data packets. As the users created the obstacles, the telemetry transmitted the position and size of the obstacle during and after its creation. The game took advantage of the time between creating obstacles to repeatedly send the final sizes and locations.

Ephemeral data transmissions between the players such as activating a weapon are more complicated because there is a chance for packet loss. The 2010 Summer of Innovation game used a simple acknowledgment system to confirm the message was received. The attacking player transmitted a number indicating which weapon to activate, and the opponent replied with the same number to indicate that the message had been received. In most cases this method worked, but in about 3 of 24 ISS matches, there were significant dropout delays between start of transmission and acknowledgment.

While the acknowledgment method works, it can introduce a non-deterministic delay between the start of the command and its effect on the opponent. Other games have simply used best effort delivery, relying on the user to transmit multiple times if the intended effect failed. This method should only be used if temporary data is expected to be transmitted on nearly every cycle and the overall effect of a single packet loss is minor. It is not a good approach if consistency with simulation results is a high priority.

2.2.5 Game Balancing

Ensuring the balance between potential strategic paths is vitally important to any game design. Even the most intriguing game concepts can become dull if only a single optimal strategy exists, and the nature of the game can rapidly shift away from the intended goals. As with any game design process, Zero Robotics game development includes a tuning phase where rules and scoring systems are adjusted to promote multiple, interesting strategic options. The balancing process is complicated by the fact that the game is fully autonomous and rigorous play-testing requires enough time to develop fully autonomous players. This section provides several steps toward achieving game balance both through simple numerical methods and hands-on testing.

2.2.5.1 Preliminary Parametric Analysis

The initial balance analysis should take place early in the design to allow sufficient time for game changes that might arise as a result of intently studying the gameplay. Before starting the balancing process it is important to identify the core components of the game that should remain distinguishing features of the game concept. These items can include special power-ups, puzzles to solve, or even fragments of potential winning strategies. Sylvester suggests turning up the influence of these elements as much as possible, then "locking" them in place to preserve the character of the game[68]. There can of course be further adjustment in later phases, but having several strongly influencing features will help diversify the strategy options.

The next step is to establish a rough scoring system as a basis for the remaining analysis. The game design usually includes several challenges to solve or outcomes that are intended to be expressed by the players during the game. A simple but effective guideline to follow is to place scoring emphasis on the desired behaviors. If it is expected to see players accomplishing a specific objective, there should be a positive effect on the score for completing it. Likewise, if a behavior is discouraged, adding a scoring penalty will make it less prominent. For examples of scoring systems from past Zero Robotics games, see Section 2.4.

Following the scoring process, it should be possible to construct the outlines of several complete strategies for solving the game. The strategies should span a range of skill levels from approaches to score "easy points" for beginners to very difficult but conceptually feasible ideas for advanced players. With the strategy outlines the game can be discretized into a set of actions, such as "move to location A," "pick up item," "move to location B." Each action is then parameterized by important design parameters such as the location of items, or the point value awarded for accomplishing a task. Combining the actions together into a sequence produces an estimate of the completion time for the strategy and the expected reward. Several branching alternatives might also be considered based on assumed actions of the opponent. Once several strategies have been decomposed in this way, the game designer can

adjust parameters and see a holistic picture of the changes.

When attempting shift emphasis in the game, it is important to remember both points and skill level can be adjusted. Scaling point values is easily accomplished, but changing skill level tends to involve modifications to the game mechanics. Difficulty can often be modulated by imposing constraints such as a fuel limit, or by adding an additional required task to complete. There should not necessarily be a smooth relationship between skill and point values since incremental refinement can be less satisfying than a leap to a much higher level. Sylvester describes this characteristic as a "strategic landscape" with "peaks of incredible effectiveness alongside deep troughs of failure" [68]. In other words, an engaging game will have several clearly separated local optima, with increasing reward for higher difficulty, as shown in Figure 2.2. These goals can sometimes be at odds with challenges strictly based on real-life engineering performance metrics like fuel consumption or tracking accuracy, which tend to have slowly varying improvements. Combining engineering metrics with heavily weighted fictional elements like weapons can introduce more significant variations between strategic options while preserving the educational content (*Engage and Educate*) and the ability to improve with better algorithms (*Incremental Difficulty*).

The final step is to group the strategy outlines by expected difficulty and adjust the parameters until the options match a desired strategy distribution. If the intended landscape cannot be achieved the game may require additional modifications. Throughout the process it is important to remember that the example strategies are only representatives of more complex combinations, some of which may break the assumptions of the initial analysis. The best way to avoid these problems is by creating real implementations.

2.2.5.2 Play-Testing with Autonomous Players

As the game matures, it is critical to write hand-coded players that complete the challenge objectives. Without playing the game first-hand it is difficult to find nuances of the rules that may be out of balance or unfair from parametric analysis alone. To start with, the players should be based on the strategies outlined in the preliminary

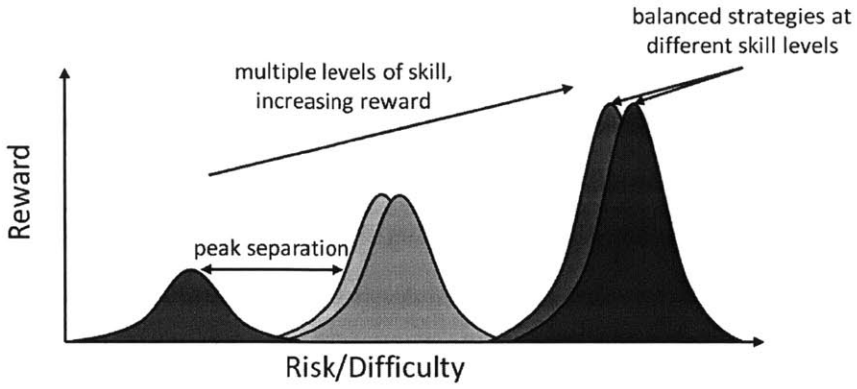


Figure 2.2: A well-designed game will not necessarily have a smooth relationship between skill and reward. Separation of the peaks of reward can encourage players to strive for making leaps in performance.

analysis, and after running through the game, the parametric analysis should be updated to reflect the true strategy performance. Keeping in mind the notes above about maintaining several strategies of varying difficulty, the game rules and scoring should be adjusted to balance the strategies.

For the purpose of benchmarking it is not always necessary to fully implement the strategies because the game developers can access the game's internal API functions. For example, in the second half of the RetroSPHERES tournament, the locations of virtual items in the volume were only available using distance measurements, but using the internal API it was still possible to access the full 3D item locations directly. These shortcuts can speed player development without compromising the broader theme of the strategy.

For the beginning of each competition phase, several of the benchmarking players should be refined into more competitive strategies. These implementations will become *standard players*, initial opponents released to all teams as generic examples of potential strategies. Contributing to the principle of *Accessibility*, supplying the players lowers the barrier to entry by eliminating the startup task of developing additional implementations for opponents.

2.2.5.3 Hardware Unit Tests

The last phase of game design involves testing on the final hardware platform. If testing time on the hardware platform is extremely limited as it is with SPHERES, a successful strategy has been to use hardware demonstrations to perform targeted unit tests to prove key features of the game. As with software unit tests, it is best practice to only exercise one game feature during each test, though with a pair of players it is often possible to perform two separate tests in parallel. The most appropriate tests are those that involve the physical motion of the robot and depend on the true dynamics of the system, such as rotating to pick up an item, testing boundary limiting behavior, or testing the effect of a "navigational disruptor" that changes the opponent's trajectory.

Though most of the session should focus on specific game elements, it is still important to run at least one head-to-head match to test the entire game sequence. In general the players developed for the game balancing phase should be used in the matches to anchor the simulation results, but it may be desirable to further customize the players to have thorough coverage of the game features in the limited time. If the session takes place during the competition season, it is also possible to use code from competitors in the tournament.

2.2.5.4 Addressing Imbalances and Game Problems During the Season

Despite the best of intentions, it is incredibly difficult to build a game without unforeseen strategies or hidden loopholes. The problem is compounded in the case of an annually re-designed game because only a short time is available for balancing. Initial play testing helps immensely but is inherently limited to the size of the team available to run through scenarios. When the game is suddenly subjected to study by thousands of bright students, weaknesses are often brought to the surface in short order. In many ways, a wide selection of unexpected strategies is exactly what is desired because it keeps the game exciting and challenging, but in some cases an unknown degenerate strategy can be introduced that completely changes the intended

character of the game. In these situations, it is helpful to have a formal mechanism that allows the game designers to correct the game in response to bugs or imbalances.

Any adjustments to the game must be approached with extreme caution. A hasty decision or overreaction risks offending competitors that may see changes as attacking their strategies, while letting an uncorrected vulnerability go unaddressed may result in widespread frustration. Each situation must be studied on a case-by-case basis, but useful generic preparations are possible. In advance of the tournament, the game design team must establish a set of guidelines to follow about updates to the game balance during the tournament. These guidelines should be made available to the participating teams to set expectations for the season. The following items are rules of thumb on which to base the guidelines:

- Establish an expectation that the game may change during the season based on observations of the competitions.

- In general, refrain from making any changes to the rules close to submission deadlines. Teams will have little time to react to any updates before the competition ends. The best time to make adjustments is between competitions when the game may already be changing due to the tournament structure.

- Immediately announce and detail any changes. A thorough justification must be provided for the change, supported by the guidelines established at the beginning of the season.

- Attempt to clearly establish the intent of each game rule in the manual. In the event of a contradiction between the intent of the game and the behavior of the game, clarify the rule and change the manual or code accordingly to keep the intent.

- Establish an expectation that teams report bugs in the game, where a bug is defined as a contradiction between the game manual and the game behavior or any action that allows a team to bypass a rule.

Constructing these rules will bind both the competitors and the game designers to a consistent set of steps for solving problems. If changes are necessary, they will be much easier to justify if they are traced directly to one of the guidelines. Just as important, changes outside the guidelines should be avoided at all cost and only considered if the issue at hand threatens the success of the ISS competition.

2.2.6 Game Manual

A detailed manual is released at the beginning of each tournament with a thorough guide to both the game and the tournament. An example of the manual format for Zero Robotics is described in A.1.5.

2.3 Tournament Design Methodology

Though considerably less flexible than the game design, the tournament season also requires several design decisions. This section will outline the standard tournament structure and highlights the decisions that must be made for preparing the season. Most of the components discussed here are based on the high school tournament. The final sections will discuss middle school and open challenges.

2.3.1 Season Timeline Overview

Building and running a Zero Robotics high school tournament is a year-long process that starts immediately after the completion of the previous tournament season. Prior to the start of the tournament the first months are dedicated to game design and testing, followed by the tournament sequence. Each tournament starts with a kickoff event followed by four phases: 2D competition, 3D competition, Alliance competition, and ISS championship competition. Table 2.3.1 summarizes the full sequence of events and dates presented in the following sections.

Table 2.3.1: Tournament Timeline

Dates	Event	Description
Jan-Aug	Game Development	Design and programming of the tournament game
Early- to Mid-April	Registration Opens	Launch of online registration and publicity efforts
Early Sept	Kickoff	Live webcast announcing the release of the game
Sept-Oct	2D Competition	First tournament round constrained to 2 dimensions
Oct-Nov	3D Competition	Second tournament round with full 6-DOF motion. First down-selection round.
Nov (1^{st} week)	Alliance Selection	Top 54 teams join into 18 alliances of 3 teams
Nov-Dec	Alliance Semi-Finals	Alliances compete for 9 ISS competition slots
Dec (2 weeks)	Finalist Code Prep	Winning alliances prepare code for ISS
Late Dec / Early Jan	ISS Finals	Live competition aboard ISS

2.3.2 Game Design

To maximize the time available for designing, balancing, and testing the game, development starts immediately after the completion of the ISS tournament. For Zero Robotics, much of the game development is performed by undergraduate researchers, so the schedule is centered around an academic calendar. The early part of the year (spring semester) is dedicated to brainstorming and concept exploration, followed by prototyping, and eventually the first rounds of game tuning. By the summer a prototype is completed, and the final refinements occur during the months prior to the tournament kickoff. Additional details of the specific process for Zero Robotics are summarized in Appendix A.1.6.

2.3.3 Registration

Each season opens with an initial registration phase. For Zero Robotics, teams interested in participating are required to register with basic team details including the

number of mentors available, the size of the team, and a short student essay. Since the tournament is free of charge, the registration form serves as a minimal filter to provide some assurance that the team is prepared to participate. While it slightly reduces *Accessibility*, establishing a local support base of dedicated mentors for the team is critical to the principle of *Engage and Educate*. The mentors help to keep the students involved with the project and provide lessons beyond what is learned from the platform.

2.3.4 Kickoff

The kickoff event marks the beginning of the official competition phase of the tournament. Each season starts with a live broadcast from MIT to unveil the season's game. Anticipation of the kickoff builds excitement for the season, and keeping the game details a secret starts all teams out on an even footing.

2.3.5 Competitions and Game Evolutions

A typical Zero Robotics season has four main competitions: 2D, 3D, Alliance Semi-Finals, and ISS Finals. Following the principle of *Incremental Difficulty,* at each change between competitions there are opportunities to update the game with new challenges. Introducing changes keeps the tournament from getting stale, and a gradual increase in difficulty helps rookie teams establish comfort with programming before the challenges become too complex. Updates can also be targeted at fixing balancing issues since scoring modifications fit naturally with the shift in game type.

Between the first two competitions, the transition from two dimensions to three dimensions may be a significant enough challenge to warrant only making slight adjustments to the rest of the game for balancing. For the alliance phase more significant changes can be introduced for two reasons. First, the teams have likely carefully honed their solutions over the course of the 2D and 3D competitions. Without making updates at this point the game can become a repeat of the 3D competition with little additional innovation. Second, with multiple collaborating teams it is desirable to

release a large challenge that motivates the alliance to distribute the work among all the members. This gives all of the teams a chance to contribute to the final program instead of replicating the lead alliance code.

Modifications of the game usually have a significant effect on game balance. Ideally, the game evolutions will be considered during the pre-season balancing activities, but it is sometimes necessary to perform the analysis for the next phase while a competition is running. A helpful strategy, employed for the 2012 season, is to focus initial balancing efforts on the 3D game, then adjust the game parameters for proper balance in 2D prior to the tournament launch. With the 3D game requiring only a small number of updates, efforts can be focused over the span of two full competitions to incorporate lessons from the 2D and 3D phases into the Alliance phase. Of course as much balancing as possible should be performed prior to the tournament start.

The RetroSPHERES game modifications between 2D and 3D mainly involved adjusting the game parameters to account for the additional dimension. Virtual obstacles were allowed to grow larger to fill the significantly increased space, and items were moved to maintain symmetry while also having a Z axis component. For the Alliance phase, more significant challenges were introduced. Items could only be located by using slightly inaccurate distance measurements, and a gravity field was added to the dust clouds to distort the trajectories of the satellites when moving through the third phase.

2.3.6 Tournament Scoring

Tournament scoring has two parts: competition scoring, and elimination scoring. Competition scoring is the process for evaluating team performance during an individual competition. Chapter 4 examines several options to rank and score teams in competitions.

Elimination scoring is the process for selecting teams for the Alliance phase and the alliances that will ultimately proceed to the ISS. For the Alliance phase, most Zero Robotics tournaments to date have used a weighted average of the 2D and 3D scores for determining the seeding rank going into the alliance selection process.

Due to differences in game difficulty and player skill between the phases, the absolute point totals received in each phase are not directly comparable. When weighting the competitions it is important to decouple the phases by normalizing the scores:

$$score_{normal} = \frac{score - score_{min}}{score_{max} - score_{min}}. \tag{2.4}$$

The values $score_{\{min,max\}}$ are the minimum and maximum scores in the competition. The normalization preserves the relative distribution of scores but re-scales it to a fixed range of $[0, 1]$. This way, teams are judged by how well they performed relative to the best player instead of by an absolute point scale. The final score is then a convex combination over the competitions under consideration with competition weights w_i:

$$\begin{aligned} score_{final} &= w_1 score_1 + w_2 score_2 + \ldots + w_n score_n \\ \sum_i^n w_i &= 1, \end{aligned}$$

which also guarantees that the final scores fall in the range $[0, 1]$.

2.3.7 Alliance Phase

2.3.7.1 Overview

Starting in the 2011 season, Zero Robotics introduced an Alliance phase inspired by the cooperative components in *FIRST*'s FRC and FTC competitions. Unlike *FIRST*, where multiple alliances are form temporarily during a single competition event, the Alliance phase in Zero Robotics is a large component of the tournament season, spanning approximately four weeks. During this period, the top 54 teams from the 2D and 3D competition phases form 18 alliances of 3 teams each and work collaboratively to improve their satellite programs. The work is facilitated by project sharing tools on the online platform, allowing teams with large geographic separations to work on the same program. Forming alliances also triples the number of teams that experience the ISS finals (*Accessibility*) and promotes useful cooperation skills

(*Engage and Educate*).

2.3.7.2 Selection Methods

Alliances have been selected with two approaches in the two seasons where they have been part of the tournament. In 2011, alliances were assigned with an automated selection algorithm under the assumption that it would be impractical to form the alliances through a live event. Figure 2.3 outlines the pairing algorithm. Teams are divided into three tiers, and each team creates a list of their desired partners from the tier below. (a) Based on the preference ranking the last team in the second tier is awarded first choice of a team in the third tier. The second to last team in the second tier is awarded their first *available* choice and so on proceeding up the tier. (b) Next, the first team in the first tier is awarded their first choice of a team in the second tier, forming an alliance of three. The selections proceed down the first tier until all alliances are formed. An example selection is shown in (c) if the teams rank their preferences in seed order.

Many teams found the automated alliance selection process to be too impersonal and resulted in selections far from the initial preference ranking. Teams also raised the concern that half of the top 18 teams were guaranteed not to attend the finals while lower ranked teams would be promoted by the ranking system. To address these concerns, the 2012 event used a live teleconference to pick alliances. The selection rules were modified to follow a modified *serpentine* selection pattern shown in Figure 2.4. This process is similar to the selection process used in the *FIRST* Robotics Competition, except that the top 9 teams are excluded from picking each other to spread out the skill levels more evenly. The steps are:

1. (a) Team Rank 1 selects their partner from anyone between Rank 10 and Rank 54.

2. Team Rank 2 selects their partner from the remaining Rank 10 – 54, proceeding until the first 18 pairs are created.

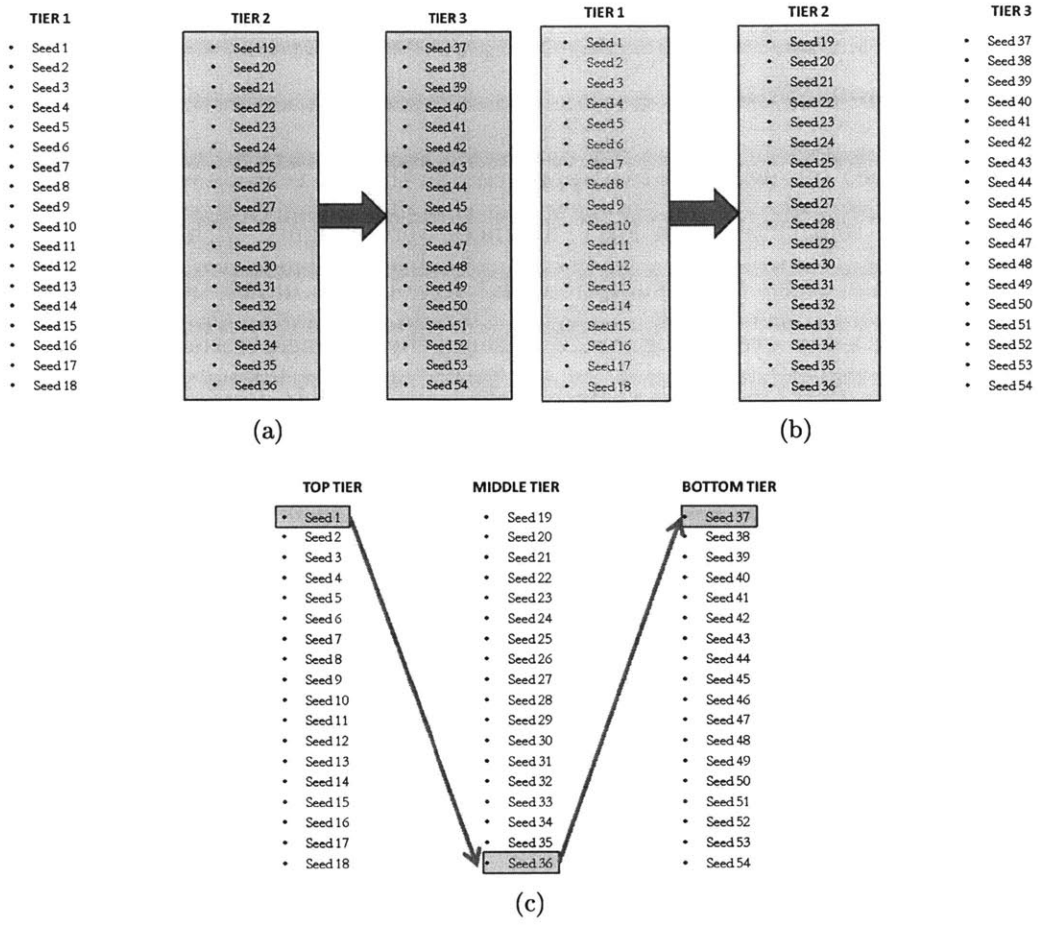


Figure 2.3: 2011 Alliance Selection Algorithm

3. A break takes place for the new pairs to discuss their selection for the 3rd Alliance team.

4. (b) The lowest ranked pair then selects their 3rd team from the remaining 18 teams.

5. The 2nd lowest rank pair make the next selection, proceeding until 18 alliances are formed (c).

Based on experience with the 2012 selection process, it is best practice to enforce a rule that only the top 18 teams may decline an invitation (in order to lead their own team). Allowing declines beyond the top 18 may result in a stalemate if a team intentionally declines to be picked by a higher ranked team. Instead, if a lower ranked

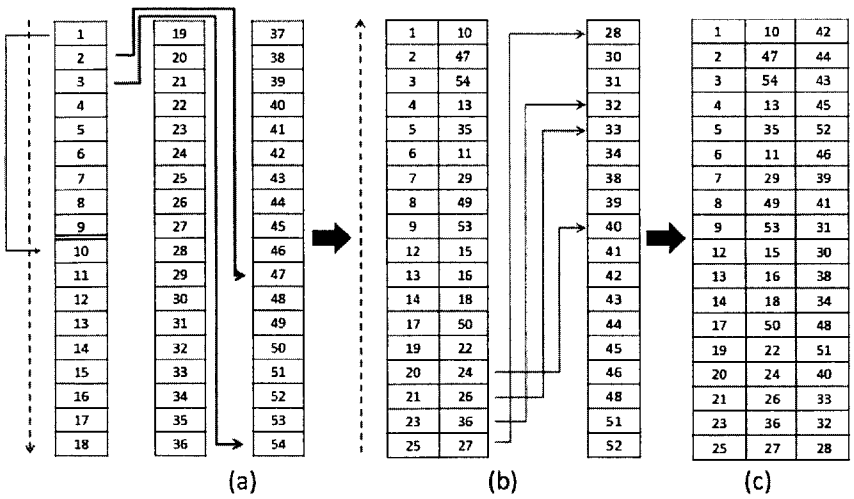


Figure 2.4: 2012 Alliance Selection Method

team declines an invitation, they lose the chance to participate in the tournament and an alternate is inserted into the ranking system. If tournament participation is lower than 54 teams, alliances of 4 teams are allowed to absorb the remainder of $n/3$ teams. This is preferable to making smaller alliances because there is a higher likelihood of more teams reaching the final ISS competition.

2.3.7.3 Challenges

An ongoing challenge for the Alliance phase is ensuring adequate participation by all the team members. The game evolutions from Section 2.3.5 were introduced in part to introduce enough of a change at the Alliance phase that all teams would be required to contribute to the solution. This approach introduces a motivation for collaboration, and based on the initial trial in 2012, it has been received favorably by the teams. However, the process for collaboration is still mostly unstructured. Beyond providing the collaborative programming tools, the Zero Robotics program does not impose additional constraints about how the collaboration should take place. If alliance participation remains a problem, it may be helpful to introduce specific tasks for each team to complete, though giving teams as much freedom as possible remains preferable.

2.3.8 ISS Finals

2.3.8.1 Event Priorities

The ISS final event is the most distinguishing feature of the Zero Robotics competition. In this last phase of the tournament, alliances finally have a chance to view their programs running on real satellite hardware in space. Achieving a successful ISS event requires extensive planning and a careful balance of priorities. Since Zero Robotics strives to give as many students as possible a chance to see their work tested in space, the tournament should focus heavily on ensuring all teams get at least one chance to run their code. For the 2011 and 2012 tournaments, the stated prioritization has been:

1. Running all submissions aboard the ISS at least once
2. Completing the tournament bracket
3. Running all submissions during live video

This arrangement ensures that all teams will have at least one match containing real ISS data. The live video priority drops below completing the tournament because matches are recorded during loss of signal periods.

Occasionally, due to time pressure during the tournament it is necessary to substitute live matches from the ISS with simulation results. A full round-robin tournament of simulation results should be prepared in advance of the live session with the same codebase that is sent to the ISS. If a simulation match is used, the corresponding match animation should be displayed to indicate the results.

2.3.8.2 Championship Formats

Several championship bracket formats have been used for the final competition. Each has the objective of selecting a champion while constraining the number of tests to the time allotted for the finals.

Single Elimination Standard elimination bracket with one loss leading to disqualification. Has the disadvantage of only running some teams once while running others several times in a row.

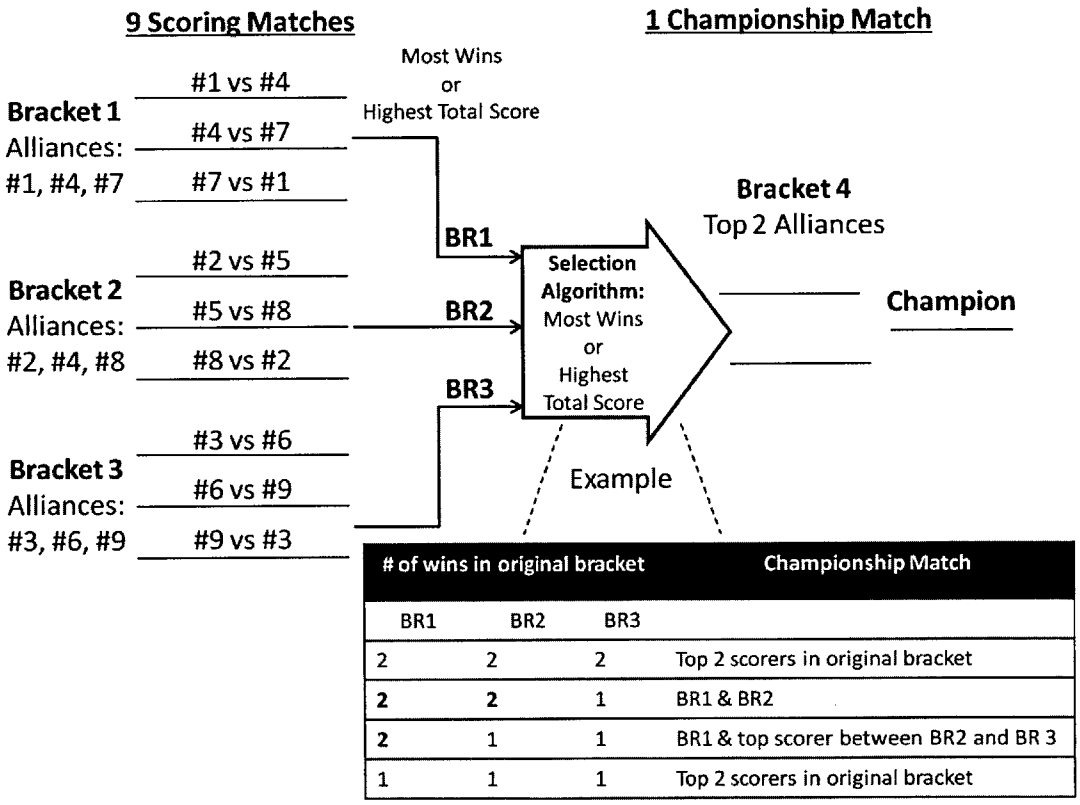
Modified Single Elimination A custom elimination structure used for the Summer of Innovation that guaranteed two live test runs. Adds an additional "loser" bracket to the single elimination format without consuming the 2N-1 matches for a double elimination format. Unfortunately, it is somewhat unfair because some teams have sudden death losses, while others have a double elimination. The format also requires 15 matches for 10 teams, which is too long for the time constraints.

Mini Round-Robin This format has been used in the 2011 and 2012 finals. Instead of running a full round robin, the teams are divided into 3 groups of 3 teams each. Each group runs a round robin of three matches, and the winner by number of matches proceeds onward. If all teams have the same number of wins, a tie breaker (such as score) is used. Figure 2.5 shows the 2012 bracket.

2.3.8.3 Final Competition Emphasis

With the extremely limited time available for final testing aboard the ISS, the Zero Robotics program has struggled with the balance between demonstrating code in space and ensuring a completely fair competition. In multiple final competitions, matches have gone undetected where the satellites exhausted their CO_2 supply, or time limits have required the substitution of simulation results for live test results. While simulation results tend to correspond very well with the general motion of the satellites, there can be mismatches between the scores in simulation and the scores on ISS. All of these events can make the final competition disappointing to the participants despite the unusual opportunity it represents.

Some of the challenges can be solved with better algorithms and tools for the students. The ability to detect low gas levels on the satellites is becoming a critical issue for the SPHERES program in general and must be addressed before the next



# of wins in original bracket			Championship Match
BR1	BR2	BR3	
2	2	2	Top 2 scorers in original bracket
2	**2**	1	BR1 & BR2
2	1	1	BR1 & top scorer between BR2 and BR 3
1	1	1	Top 2 scorers in original bracket

Figure 2.5: 2012 Final Competition Bracket

Zero Robotics finals. Mismatched simulation and ISS results are less problematic as long as they are explainable by real-life discrepancies. The simulation tools should be improved to better highlight sources of randomness and give users to explore a wider range of test cases.

Operational issues such as skipping scheduled matches and missing depleted tanks are more problematic because they are often due to time pressure. One solution is to reduce the number of matches in the tournament. A single elimination competition for 9 teams can be completed successfully with extra margin but usually only gives a single test run to 4 of the teams. With the extra time non-critical matches could be performed for additional demonstrations. Along the same line, it might be reasonable to shift the emphasis of the final competition to performance related metrics, such as how well the teams do against a standard player or award several performance prizes in addition to the championship.

2.3.8.4 Virtual Finals

Teams that don't qualify to compete on the ISS have the chance to compete in an alternate track of the tournament in the online environment. A champion is also selected from the virtual competition. Though previous seasons have kept this event separate from the ISS finals, the Virtual Finals could be combined with the ISS finals to build an even larger event.

2.3.9 Other Zero Robotics Tournaments

The preceding discussion has covered the design features of the Zero Robotics High School tournament based on four years of development. Pilot programs for two additional tournament types of tournaments that utilize the same platform have been executed during the same period. A discussion of the design lessons from these tournaments will be part of the tournament history in Section 2.4.

2.3.9.1 Middle School Tournaments

Two pilot programs have taken place for the creation of a Zero Robotics program for middle school students, each following a format heavily centered around a curriculum that introduces students to the necessary math, physics, and programming concepts to compete. At 5 weeks in length, the programs have been much shorter than a typical high school season and take place during the summer. Students spend 2-3 weeks programming the satellites, and the final tournament takes place several weeks after the completion of the curriculum. With the small scale of the pilot programs, students did not compete in virtual competitions, but future tournaments will use a single competition at the end of the competition period to perform a down-selection for ISS.

2.3.9.2 Open Challenges

The Zero Robotics platform has been designed with the intent of eventually opening the full capabilities of programming SPHERES to anyone. Zero Robotics Open Challenges are tournaments open to the general public targeted at solving a specific algorithmic problem relevant to satellite control research. The first and only open challenge to date is the Zero Robotics Autonomous Space Capture Challenge discussed in Section 2.4.5.

2.4 Tournament History

Zero Robotics has been directly shaped by its history. This section will review the game and tournament designs from each season along with the key lessons that have contributed to the program.

2.4.1 2009 Pilot

The Zero Robotics team was privileged to receive seed funding to create a pilot program during the fall of 2009. The pilot program consisted of two schools from

North Idaho: Bonners Ferry High School and Post Falls School District.

2.4.1.1 Game Design

Gameplay As the first experimental step toward creating a software interface for high school students to program SPHERES, the 2009 game was intentionally limited in complexity. The game involved a *helper* assistant, which must reach a goal and the other, a *blocker*, which tried to prevent the *helper* from reaching the goal. Students developed programs for both helper and blocker roles. During all maneuvers, the satellites conserved fuel to reach the target before exhausting out of a virtual fuel allocation. A collision avoidance algorithm aboard the Blocker satellite forced the Helper to move away if the satellites came in proximity. A major component of the challenge was determining how to use the avoidance algorithm for offense maneuvers.

Scoring The game score was awarded to the helper based on its performance in the game.

- *Goal Bonus (100 pts)* The Helper satellite received up to 100 points for reaching the goal zone before the match time limit expired as a percentage of the time remaining.

$$g = 100 \times \left(1 - \frac{t_{goal}}{t_{total}}\right)$$

- *Blocking (-100 pts)* During the game, the Helper satellite tracked each second it was blocked by the Blocker satellite (avoidance algorithm active) and divided this by the total elapsed time. The percent of time that it was blocked was subtracted from the score.

$$p = -100 \times \frac{t_{blocked}}{t_{total}} \tag{2.5}$$

- *Fuel (30 pts)* The helper was penalized 30 points for running out of fuel and

received a 30 point bonus if the blocker ran out of fuel.

$$f_{helper} = \begin{cases} -30 & \text{helper fuel exhausted} \\ 0 & \text{otherwise} \end{cases}$$

$$f_{blocker} = \begin{cases} +30 & \text{blocke fuel exhausted} \\ 0 & \text{otherwise} \end{cases}$$

- *Other Penalties* Teams were penalized 10 pts for running into the walls, 5pts for exiting and re-entering the volume, and disqualification for faulty software causing a test termination.

MIT Standard Players During the 2009 year, a competition interface was not available for the teams to compete online. To give the teams a sense for how their opponents were progressing strategically, the Zero Robotics team released standard helper and blocker players to both teams. The players were updated over several iterations to incorporate strategies from the competitors, thereby distributing information to the teams by a third party.

2.4.1.2 Tournament Design

The 2009 pilot established the initial template for the standard tournament structure presented in Section 2.3.1. There were several differences of note with the current tournament structure.

- With only two schools there were no elimination rounds and no alliances, but scores in each phase were kept for all phases to study the strategy for elimination in future competitions.

- Just following the kickoff, teams were provided with an introductory practice game prior to the launch of the tournament game. This was mostly driven by delays in the tournament implementation, but the slow ramp-up helped to establish a pattern of gradually increasing difficulty in the tournament.

- The initial 2D phase was implemented as a ground demonstration on the SPHERES flat floor facility with the intent that the 2D hardware phase could be used as a down-selection round. For the 2009 season, teams simply submitted their current strategies under development in the 3D simulation, and the Zero Robotics team restricted the motion to 2D. During the matches, the teams watched a live webcast of the flat floor matches.

- The ISS event did not use a bracket. Instead, all permutations of helper and blocker pairs were tested. A full analysis of the ISS test results is available in the SPHERES ISS Test Session 21 Report[51].

2.4.1.3 Lessons

Ground Demonstration Difficulties Despite attempts to provide a realistic environment for ground testing, feedback from the high school teams suggested that it was very difficult to use the results from 2D testing to extrapolate the 3D behavior of the satellites. The satellites were occasionally disrupted by friction effects and collisions between air carriages, and the 3D trajectories programmed by the satellites were only followed approximately. The flat floor testing gave students a realistic picture of ISS testing with downtime associated with changing consumables as well as a limited idea of how environmental disturbances affect the motion of the satellites.

Based on these results it was clear a change was necessary to successfully utilize the flat floor as an intermediate elimination round. The conclusion was to alter the structure of the initial phase of the competition in the same pattern as SPHERES research with a separate 2D implementation in simulation and hardware.

Game Balance Leading up to the final ISS competition, it became clear that with many strategies, the blocker could easily overpower the helper in the game. This was a strong initial indication that the Zero Robotics program would require carefully studied game designs for future seasons. The unintended imbalance also highlighted the pitfalls of relying exclusively on MIT-developed players for initial testing. After just a few weeks, the students were able to best the initial strategies released by the

Zero Robotics teams. This pattern has continued in subsequent seasons and is likely due to the significant amount of time the competitors spend analyzing and testing the games.

While the blocker player was indeed overly strong in the game, it was not completely unstoppable. For the final event, MIT specifically prepared a helper strategy to defeat each of the high school blockers based on knowledge of the teams' source code and successfully demonstrated them on-orbit. This situation is also important because it shows that just ensuring that a winning strategy is available is not always sufficient to ensure teams will find it.

Control Updates The API for the pilot was limited to supplying simple position target commands to move the satellite to an intended position. An internal PD controller moved the satellites to the target. Both teams found this interface too limited and implemented their own ways of modifying the targets to make the satellites go faster. This prompted the creation of a more detailed control API.

Key Programmatic Suggestions Several pieces of feedback from the pilot season proved critical to the subsequent design of Zero Robotics:

- *Online Interface*: Throughout the pilot season, students used an executable downloaded from a server. Posting any update required teams to re-download and re-install the tools. Even with two teams, it was difficult to ensure all participants were running the latest version. Teams strongly suggested centralizing the competition by moving the tools to an online interface.
- *Opportunities for Collaboration*: Even in small teams, competitors in the pilot found it difficult to collaborate on writing software. Future competitions should focus on facilitating collaboration within teams.

2.4.2 2010 SoI

The 2010 Summer of Innovation (SoI) tournament was the second Zero Robotics tournament and the first pilot of a Zero Robotics middle school program. Based

on suggestions from the pilot season and proposals made for DARPA's InSPIRE program, SoI debuted the first web-based prototype for programming SPHERES.

2.4.2.1 Game Design

Gameplay The SoI game was a fictional race constrained to a 2D plane shown in Figure 2.6 with both competitors stacked vertically in the test volume. The players started on one side of the volume and were required to race to a region on the other side, called the *dock zone* then return, taking a right angle turn to finish the game. Going toward the dock zone, virtual obstacles obstructed the path of the satellites, requiring trajectories without a straight paths. If a player collided with an obstacle or one of the walls, their satellite was forcefully returned to a known holding position, then released to continue onward. Returning from the dock zone, the obstacles disappeared for a higher speed return.

Three types of single-use power-up items were available by traversing the playing field in through certain regions. If the satellite passed through the +X side of the volume, it picked up a *magnet*, which could be activated to pull toward the player for several seconds. On the -X side of the volume, the player retrieved a *bomb*, which pushed the opponent away when activated. Picking up both the bomb and the magnet created an *EMP*, which temporarily disabled the opponent satellite and allowed it to drift freely.

Scoring The first player to cross the finish line won the match. The satellites synchronized finish times to ensure the correct winner was selected.

2.4.2.2 Tournament Design

Middle school participants from 10 Boston area schools spent five weeks learning to write programs for the satellites. As part of the program, a ground demonstration took place at the MIT flat floor, though the competition was not scored. The ISS final event used a modified single elimination bracket.

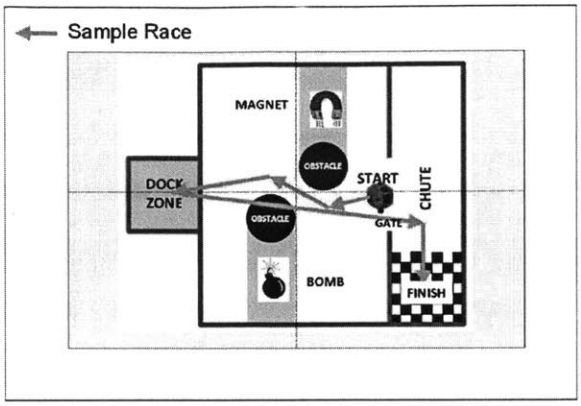


Figure 2.6: The 2010 Summer of Innovation game had a 2D layout with the teams competing in two layers. Competitors raced around two obstacles, picked up optional power-ups, then raced to the finish zone.

2.4.2.3 Lessons

Flat Floor Demonstration Of all the ground-based hardware events from the Zero Robotics seasons, the SoI demonstration best fulfilled the purpose of illustrating the differences between the online simulation environment and the hardware platform. Some of the factors contributing to the success were:

- Instead of running the satellites together on the same playing field, the two satellites were allocated physically separated regions on the flat floor in a side-by-side configuration. This allowed them to move about the volume without the issue of air carriage collisions. Internally, the Zero Robotics software virtually re-centered the satellites, so they appeared to one another to be using the same coordinate system.

- Students viewing the demonstration were present at the event instead of viewing via webcast. In person, students could watch both the real satellites moving on the floor and a real time virtual version of the game.

- The relatively simple game rules with limited interaction between the satellites and a race format were more clearly visible.

ISS Time Allocation The SoI final competition used a modified single elimination bracket with 15 matches for 10 teams. While all matches were completed, several were not completely successful because the satellites exhausted their fuel supplies before the end of the test. There was not enough time to re-run the matches, so the Zero Robotics team used the partial match result to declare a winner. This event motivated the need to prepare backup simulation results ahead of time in the final competition.

2.4.3 ZRHS2010: HelioSPHERES

The 2010 tournament was the first nationwide version of Zero Robotics, executed as a limited pilot program in preparation for future open registration tournaments. The online web platform initially developed under Summer of Innovation was retrofitted for higher user capacity and text-based code editing features. The game, HelioSPHERES began a tradition of naming the tournaments based on the theme of the game.

2.4.3.1 Game Design

Gameplay The background motivating theme for HelioSPHERES was an on-orbit assembly mission where a assembler satellite was tasked with maneuvering a large solar array to a space-based solar power station. Both satellites were initialized in the center of the volume at a random position along the perimeter of a circle shown in Figure 2.7. At the beginning of the match the location of the solar panel was partially unknown, requiring the competitors to scan for its position using a limited field of view sensor. Once the panel was located, the satellites performed a docking maneuver to attach themselves to the panel, then moved to the other side of the volume to deposit the panel at the power station.

As an antagonistic element, the satellites were provided with a *navigational disruptor*, capable of applying a strong force to an opponent along the vector between the two satellites. The disruptor required virtual *charge* to deploy, and the resource could only be replenished by pointing the back of the satellite away from the "sun"

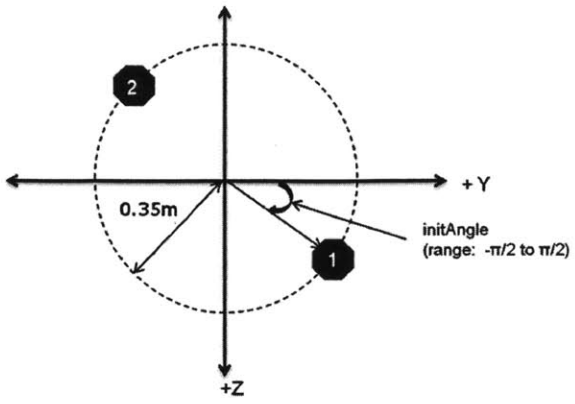


Figure 2.7: Players in HelioSPHERES started at random opposite positions around the perimeter of an initialization circle.

at the center of the volume.

If players exited the volume while carrying the panel, it was dropped close to the point of exit, and a new docking maneuver was required to pick it up again.

Scoring A HelioSPHERES game ended when one of the players docked with the power station, the match timeout expired, or both players expended their fuel allocations. The winner was determined by a prioritized set of rules to break any ties:

1. The first player to finish docking to the station won immediately.
2. If the game ended before either team docked to the station, the team with longest time holding the panel won.
3. If neither player docked with the panel, the player that discovered the panel first won.
4. If neither player found the panel, the player closest to the panel at the end of the match won.

2.4.3.2 Tournament Design

The HelioSPHERES season followed the same pattern as a standard Zero Robotics tournament, but the 2D competition was conducted a live event from the MIT flat

floor facility. At the time, Zero Robotics ground events were still being considered as a potential way to down-select teams. To run the competition a live double elimination bracket competition was performed for three regions via webcast on three separate days. In addition to the live video feed, telemetry from the satellites was streamed to a modified version of the simulation's 3D visualization tool with the objective of replicating the positive flat-floor experience from the Summer of Innovation tournament.

The 2D competition also featured an experimental hybrid co-simulation model where the satellite hardware on the flat floor provided X and Y position states, while an onboard simulator modeled the Z axis and attitude dynamics. From the live video view one could view the satellite position in a slice of the playing volume, while the 3D animation showed the game view of the tournament. In this way students did not have to develop a separate 2D version of the program for the 2D phase.

The 3D phase took place in simulation, ending in a round-robin tournament. Out of the 24 participating teams, 10 were selected for the ISS finals with a weighted score, weighting simulation results 60% and ground results 40%.

2.4.3.3 Lessons

Attitude Representation The 2010 tournament was the first to allow control of the satellite's orientation in three dimensions. A simplified attitude representation based on a unit vector pointing direction helped to make controlling attitude accessible to the student competitors. Teams were able to successfully scan for the panel location, point the disruptor at opponents, and configure the satellite in the correct orientation to dock with panel. Details of the representation are discussed in details of the Zero Robotics API under Section 3.4.3.

Ground Demonstration While the 3D co-simulation tools and live view of the animation added additional depth to the live ground demonstration, the Zero Robotics team still struggled to make the results meaningfully reflect the simulations. Though SPHERES researchers have quite successfully used the 2D air bearing facilities to

perform research and prepare for ISS testing, experiments are usually carefully tailored to working with the irregularities of the system, and experiments often take 10s of iterations before an algorithm is adequately demonstrated. In addition, the task of preparing and executing a live broadcast in the middle of the season puts a significant strain on the team. From team evaluation surveys, only 27% of 240 students surveyed found the ground demonstration to be essential or thought that it contributed to their ZR experience, and 38% didn't view the demonstrations at all. Without additional control layers such as adaptive friction compensation and an immense amount of testing to ensure better repeatability, flat floor demonstrations are best left out of Zero Robotics tournaments.

Game Balance One of the strongest lessons from HelioSPHERES involved an imbalance in both the the game dynamics and the scoring rules. The navigational disruptor tool supplied by default to all teams was originally designed to have limited effectiveness due to the need to recharge from the virtual sun at the center of the volume. However, by positioning the satellite between the sun and the opponent it was possible to replenish charge fast enough to nearly continuously repel an opponent. A team could spend most of the time during a match repelling until an opponent's fuel was exhausted, then win by triggering one of the secondary tie breakers without completing the mission. During the ISS finals only one of the competitors completed the full docking scenario.

The result from HelioSPHERES was a clear indication that game designers must be cognizant priority inversions, especially due to tie breaking. The solution of allowing but discouraging ties by applying a penalty was attempted during RetroSPHERES, but led to difficulties with the competition scoring system. Future games can avoid repeating the balance problems by making easy win strategies less reliable and attempt to counter powerful game elements with others of similar strength.

2.4.4 ZRMS2011 and ZRHS2011: AsteroSPHERES

The 2011 tournament was the first open-registration national tournament and the first international tournament. 113 teams in the US and 13 teams from the EU participated in the program. During the game development phase, an early version of the game was used for a second small-scale middle school pilot program.

2.4.4.1 Game Design

Gameplay Based on the aggressive nature of the 2010 competition, the 2011 competition attempted to introduce a component of collaboration into the tournament structure. For the game AsteroSPHERES, teams were tasked with collaboratively extracting minerals from virtual asteroids. In many of the game objectives, more points could be achieved by cooperation between the players.

In the game scenario teams worked to extract Helium-3 from the surface and interior of two asteroids, Indigens and Opulens, during three phases of 60 seconds each. During the first phase, competitors acquired power-up items to assist in the remaining steps: one of two lasers to melt ice on one of the asteroids, a disruptor upgrade to enhance repelling and attracting, and a shield to protect against an opponent's disruptor. In the second phase, SPHERES could mine the asteroids by revolving around the asteroid to perform *surface collection* or spin at the location of the asteroid to gain ore by *drilling*. The plane of revolution and axis of rotation were randomized for each match. Opulens, contained a richer ore deposit but started the game covered in a thick layer of ice. Players could cooperate in the second phase to melt the ice for a point bonus and also expose the high value deposit. The ice disappeared in the final phase.

For the final 60 seconds of the match, the satellites could continue mining or race to independent mining stations to deposit the collected ore. Mining stations appeared in the last 10 seconds of the match. Using a laser the teams could signal completion of the mission to earth ending the match up to 10 seconds early. Matches could also end early if both players ran out of virtual fuel.

Scoring

- *Melting Ice Sheet:* Each time both satellites succeeded in hitting the ice sheet on Opulens, both satellites were awarded 0.1 points. (Max 1.5 pts)

- *Drilling and Surface Collection:* Points for mining were maximized by spinning or revolving closest to a target angular velocity, linearly decreasing away from the target. Spinning at $30\,°/s$ on Indigens awarded the maximum of $0.06\,pts/s$, while revolving at a radius of 10-40 cm with an angular velocity of $8\,°/s$ resulted in $0.066\,pts/s$. Mining Opulens multiplied revolving and rotating scores by 1.3.

- *Cooperation Bonus:* If the teams simultaneously revolved and rotated about the same asteroid, the point acquisition rate was doubled.

- *Race Bonus*: The first satellite to reach one of the mining stations received up to 4 bonus points. If the second satellite reached the mining station, the second satellite received 2 bonus points and the first satellite to finish received an additional 2 bonus points.

- *Penalties:* Points were deducted at a rate of $0.06\,pts/s$ for leaving the interaction zone. If the collision avoidance algorithm activated within 15 cm of any mining station during Phase 3, both satellites lost 1 point per second. This penalty was intended to prevent teams from pushing each other off the mining zone locations before the final 10 second period when the stations appeared.

2.4.4.2 Lessons

Game Balance During development, the intent of the game was to give both mining approaches equal chances to win albeit with different strategies. Revolving, because it required trajectory planning and more fuel, was awarded significantly more points. A spinning satellite could win by leaving the match early to gain bonus points in the race phase, potentially using the disruptor to delay the other satellite. Two major issues arose at the start of the season:

1. There was a significant imbalance between the spinning and revolving points. The team that managed to revolve effectively was nearly guaranteed to win the match.

2. The strong emphasis on collaboration led teams to assume that the imbalance was intentional. Teams quickly began to organize ways of trading turns at spinning and revolving using the side of the playing field the satellites were initialized on as a lightweight way to make the decision. The competition focus shifted to performing the best possible rotations and revolutions.

The first item was the result of limited parametric analysis of the scoring system during game testing, which mainly focused on the creation of standard players. For the remainder of the season, the game design team attempted to bring the strategies back into balance by de-emphasizing revolving and increasing the final race bonus. Unfortunately, in light of the second issue, the adjustments began to weaken existing strategies and were perceived negatively by some of the competitors.

To avoid this situation, the design team should have either:

1. Made *more* aggressive changes to the game parameters to set up a broader range of strategies. Too much concern was paid to making as small of a change as possible to the game rules and making the strategy alternatives nearly equally balanced. The main alternative of leaving drilling or surface collection early to race to the mining station provided nearly the same points as revolving with much higher risk. Teams did not even consider antagonistic strategies because there was little incentive to follow them and also risked losing points in the collaborative part of the game.

2. Directly embraced the performance challenge that the teams latched to at the beginning of the tournament. Instead of attempting to guide the game back toward the initial intent of a mixed competition and collaboration in each match, the competitive aspect could have been pushed even more heavily into the competition scoring system with additional challenges to make the collaboration

component harder. Part of achieving a successful tournament season is realizing that the game can be shaped by the competitors as well as the game designers.

Better Processes for Bug Management During the season two notable bugs resulted in lessons for future tournaments. The first bug, eventually labeled "instamelt," allowed a team to instantaneously melt the ice layer around Opulens. The bug was discovered after the official 2D competition results were released when a single team made use of the issue. The final results of the competition were not modified because the game manual did not explicitly contradict the game behavior, and the team had believed the bug was a hidden strategy. Other teams found this decision to be unfair.

The second major bug was reported privately just a few hours before the final submission deadline for the ISS phase. It involved an inconsistency between the behavior of the game code and the rules described in the manual. The game design team chose to leave the game code in place because making a change to the program would introduce risks for the ISS finals and would not give teams much time to react to changes in a behavior they had been testing against throughout the season. Nonetheless, the reporting team expressed concern that reporting the bug to the rest of the competitors indirectly broadcast a hidden strategy and represented a last minute change of the game rules.

Both incidents highlighted the need for an official policy for addressing bugs in the game code. The game manual guidelines for changes to the game under Section 2.2.5 were established for this purpose, along with a code freeze deadline for submitting bug reports. The Leaderboard scoring system in Chapter 4 has also helped to bring bugs to light earlier in the competition period and prevent the last minute changes that tend to least to the most contentious outcomes.

Alliance Phase Two significant issues from the alliance phase had an important effect on future seasons. First, some teams found the automated pairing algorithm for assigning alliances unfair. There was no option to decline participation in the

finals, and some teams simply dropped out of the competition without notifying their partners.

Second, the challenge remained exactly the same between 3D and Alliance phases except for small adjustments to the game balance. Without a new challenge to solve, many alliances replicated the led team's best-performing code and ceased additional development work. This outcome motivated the need for game evolutions throughout the tournament season, especially at the Alliance phase.

Collaboration and Competition in Zero Robotics An examination of the 2011 season was performed by Nag in [53] from a broader perspective of combining collaboration and competition to achieve an objective through crowdsourcing. In the current context of tournament and game design, it is important to be aware that adding a cooperative element to the game may induce a coupling with the competition and tournament scoring systems. At a match level, cooperation is not meaningful unless it confers a benefit to both of the competitors. In AsteroSPHERES, cooperating teams were awarded higher scores, which in turn resulted in the need for a competition structure that recognized teams with high scores. The coupling drastically changed the way the game was played because teams attempted to optimize cumulative points, not necessarily strategies that outperformed opponents in each match. For future tournaments, game designers must be aware that adding a cooperative element at the competition level will require additional effort balancing the game rules with the competition scoring system.

There are approaches to introducing cooperation with a limited dependence on the overall scoring system. A cooperative element can be introduced to scale the complexity of the challenge. If the teams choose not to cooperate during a limited portion of the match, the rest of the game is more difficult and it becomes harder to gain a competitive edge. This does not link to the competition scoring system beyond which team wins or loses, and it opens up the strategy space. Another option, sometimes used in *FIRST* Robotics Competitions, is to award a small additional bonus in the overall scoring system to teams that accomplish an easily identified

cooperative task. In the FRC 2012 game Rebound Rumble[SM], teams were ranked in qualification rounds by number of wins plus bonus points for each match where teams cooperated at the end of a match by balancing on a bridge at the center of the field [24]. This method strongly encouraged teams to complete the cooperative task while preserving the incentive to win.

2.4.5 ZROC #1 Zero Robotics Autonomous Space Capture challenge

The Zero Robotics Autonomous Space Capture Challenge (ZRASCC) was launched as an experiment in crowdsourcing for the development of spacecraft control algorithms. The Zero Robotics platform was opened to the general public for the first time, and additional enhancements were added to the Zero Robotics API to access lower levels of the SPHERES control system.

2.4.5.1 Challenge Design

Challenge The Autonomous Space Capture Challenge consisted of synchronizing rotational and translational motion of a spacecraft, or *Tender*, with a tumbling space object, or *POD*, thereby setting up the conditions to "capture" it. The challenge specifically focused on producing a control algorithm to minimize the propellant cost to capture the object. Competitors were tasked with identifying the most challenging docking conditions by specifying several parameters of the space object's motion. To complete the challenge, the Tender was required to:

1. Maneuver to a *Capture Zone* located 25 ± 1 cm along the -X axis in the 7° *Approach Cone* of the space object (see Figure 2.8).

2. Align for capture by orienting the -X axis of your satellite within $\pm 2.5°$ of the space object's -X axis (see Figure 2.9)

3. Stay within the capture zone for 5 seconds with a relative velocity of less than 5 mm/s

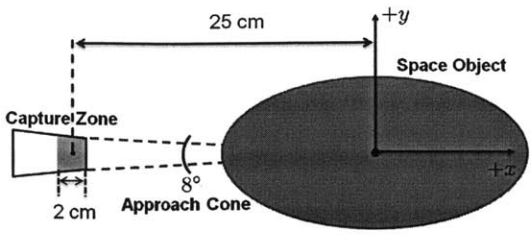


Figure 2.8: ZRASCC Capture Zone Positioning

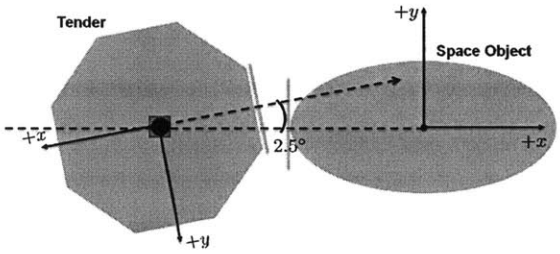


Figure 2.9: ZRASCC Capture Zone Alignment

while avoiding the following constraints:

1. The Tender must maintain a 30 cm collision avoidance distance from the center of the space object except when in the approach cone (see Figure 2.10). The approach cone ends at the boundary of the capture zone at 24 cm from the object.

2. Docking must occur while the centers of both the Tender and the space object are within the Object Capture Area. The boundaries are shown in Figure 2.11. It is important to note that the absolute position of the tender within the test

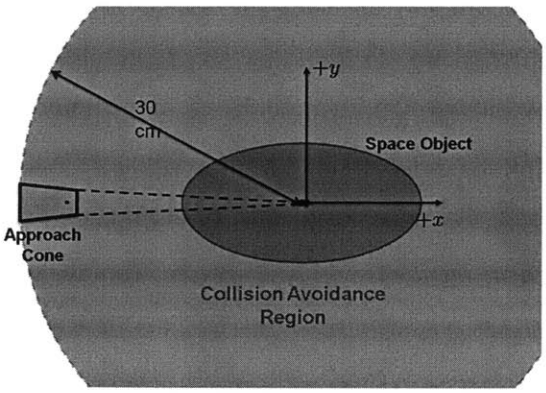


Figure 2.10: ZRASCC Collision Avoidance Region and Avoidance Cone

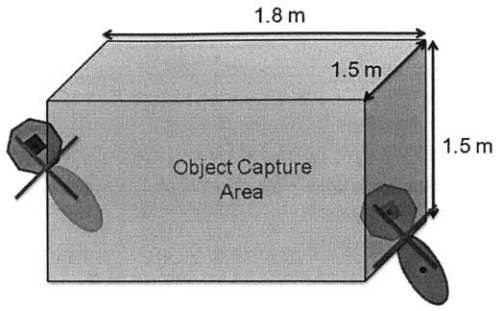


Figure 2.11: ZRASCC Capture Area

volume will have a high uncertainty.

3. The Tender must complete the challenge without running out of a virtual tank of propellant. Each time the Tender fires a thruster, a counter records the total time it is open. An allocation of 30 thruster-seconds is allowed for completing the challenge. The total propellant remaining in thruster-seconds is available through the API function *ACGetFuelRemaining()* and is displayed in the visualization.

4. The Tender must complete the capture maneuver within a time period of 210 seconds.

Scoring During the scoring process, submissions ran in head-to-head matches against the top performing projects on a competition leaderboard (see Chapter 4). For each pairing, the scoring system executed matches with the players in the roles of both SPH1 and SPH2, and both players used the space object parameters specified by SPH1. If SPH1 did not specify parameters, the parameters from SPH2 were used, and SPH1 was not scored. Both competitors were initialized in the same positions and performed the same capture challenge with the same object parameters simultaneously. The final score for the match was the *difference* in propellant consumed between the two players.

$$score_1 = propUsed_2 - propUsed_1$$

$$score_2 = propUsed_1 - propUsed_2$$

If only one competitor completed the challenge in the allotted time without violating constraints, the score for the successful tender (+) was automatically set to the maximum 5 points, and the score for the unsuccessful tender (-) was set to -5.

$$\begin{aligned} score_{+} &= 5 \\ score_{-} &= -5 \end{aligned}$$

As an extra incentive for attempting to complete the challenge, if the unsuccessful tender managed to reach the capture zone for at least one second, and the relative fuel consumption between the satellites is within 1 unit, the unsuccessful tender received 0 points instead of -5.

$$\begin{aligned} score_{+} &= \text{as above} \\ score_{-} &= 0 \end{aligned}$$

If neither satellite completed the challenge their scores were both be set to 0.

2.4.5.2 Tournament Design

Due to the open challenge nature of ZRASCC, the tournament followed a different structure than other Zero Robotics tournaments. The tournament took place over the course of 4 weeks, with each week representing its own mini competition called a *milestone*. Competitors made submissions to the leaderboard system, and at the end of the week, the top ranked player on the leaderboard was selected as a finalist. The code from the winning team was released publicly for all teams to use in the next phase of the tournament with the objective of raising the collective performance of all competitors in the algorithmic challenge. Modifications to increase the difficulty of the challenge were also added at the end of each milestone. The incremental challenge updates were incorporated into the 2012 high school tournament and will become a regular part of the high school tournament (see Competitions and Game Evolutions in Section 2.3.5).

2.4.5.3 Lessons

Test Session 33 completed the final phase of the Zero Robotics Autonomous Space Capture Challenge with a live ISS demonstration. From the online competition and the ISS demonstration, there are a number of important conclusions for the capture problem and also for future algorithmic challenges. Full analysis of the results specific to the algorithmic challenge are covered in the SPHERES Test Session 33 Report [52].

As a trial of crowdsourcing algorithms for spacecraft, ZRASCC highlighted important considerations for future open tournaments. Crowdsourcing caters well to general software development challenges because there is a relatively large community of professional and amateur programmers capable of writing functional software. In ZRASCC, drawing on the same community to robustly solve challenging control problems proved more difficult. Competitors were able to produce algorithms that achieved the docking objectives in isolated cases, but the competition did not produce robust solutions to a wide variety of scenarios as intended. Part of the problem is that on the time scale of a short competition, participants mainly have a chance to focus on algorithm *sequencing*, or piecing together and adjusting parameters of existing algorithms, as opposed to algorithm development. Crowdsourcing applications like protein folding have been successful at translating or at least comparing sequences of high level actions to state of the art algorithms [41], but the results of ZRASCC were not competitive with potential solutions from the literature.

Given that building new control algorithms requires a specialized knowledge base, producing a high performing solution may not be accessible to many of the competitors in an algorithm development challenge. There are two alternatives that are directions of research for future competitions:

1. Instead of relying on teams to independently implement a solution, create an open source challenge where participating teams work together to create the solution. This reduces the competitive motivation for the tournament, but the draw of an ISS event at the conclusion may be sufficient to draw many to participate. The teams could also be divided into multiple large conglomerates

that still share code but try different approaches.

2. Further develop the Zero Robotics API in a way that allows users to build complex control approaches with relatively simple building blocks along with tools to ensure the applicability of the algorithms.

A second reason for low performance in the challenge was a mix between low participation and a lack of tools for thoroughly testing programs. ZRASCC was designed to encourage robust solutions by pitting competitors against a wide variety of scenarios posed by opponents. Though nearly 100 teams registered for the competition, at most 15 made submissions at the milestone deadlines. Without a large number of opponents, teams did not experience many variations in the missions scenarios. In addition, in absence of opponents to test against, the teams did not have access to tools to run batch simulations over a variety of parameter values and analyze the resulting performance. This deficiency has motivated the development of the Monte Carlo tools covered in Appendix C.

2.4.6 ZRHS2012 RetroSPHERES

2.4.6.1 Game and Tournament Design

The game and tournament design for RetroSPHERES has been covered throughout the preceding sections, but there are several more items to note from the season:

- The 2012 tournament introduced a continuous scoring system called the Leaderboard, covered in detail in Chapter 4.

- This tournament removed the ground demonstration completely based on feedback from the 2011 season that the demonstration videos were not of much benefit.

- The alliance selection phase used a live teleconference for virtually gathering the teams, which proved to be a favorable improvement over the automated selection method from the 2011 season.

2.4.6.2 Lessons

Game Balance Prior to the season a strong effort was made to balance the game with both parametric and standard player approaches, and there were relatively few problems with balance during the season aside from small bug fixes. However, related to balance, many of the teams converged to very similar approaches. The game had been explicitly designed with at least 4 major strategies:

- *Rush:* Don't create any obstacles, pick up required item, dash for finish before opponent can make it there.
- *Mixed:* Create at least one obstacle, pick up at least one additional item to assist with the obstacle field.
- *Builder:* Focus on creating many obstacles, then pick up the required item and finish.
- *Hoarder:* No obstacles, attempt to pick up all items.

Most players chose either the mixed strategy with at most one obstacle or the rush strategy. Despite attempts at making obstacle creation more attractive, few teams chose to make more than one. In this case, the game may have been *too* balanced–not enough separation between the strategy options or insufficient incentive to choose alternative solutions, as illustrated in Figure 2.12. Given the problems during AsteroSPHERES with a single dominant strategy emerging early in the competition, there was a heavy focus on making strategies in RetroSPHERES have almost exactly the same expected performance. In light of the more nuanced view of a strategy landscape from [68] discussed in Section 2.2.2, trying to make the strategies so close may have forced teams to choose the strategy that was easiest to implement and gave any small performance advantage over the others. The resulting lesson is that is acceptable to have different levels of payoff for strategies as long as higher scoring approaches also entail higher risk or difficulty. For instance, if creating obstacles could have almost completely blocked off all paths to finishing the race but required extreme conserva-

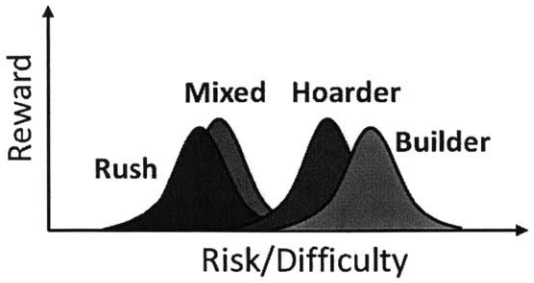


Figure 2.12: The strategy options in RetroSPHERES were well balanced, but increasing difficulty did not result in a clear reward payoff. As a resutl, most competitors chose to pick a reliable strategy over a more risky approach.

tion of fuel to make it to the end, more teams may have gravitated to the builder approach.

Game Evolutions The 2012 tournament was the first season to experiment with game updates at each of the main competition phases. The additional challenges were generally received positively by the competing teams, and in contrast to the previous season kept teams actively solving problems throughout the alliance phase. The only item of concern was that the additional challenge of locating items with only noisy distance information may have been too difficult to solve. Specifically, teams commented that the noisy measurements made it difficult to apply known algorithms to the problem, and the solutions were beyond the background knowledge that could be expected from high school level students. Most teams managed to solve the problem in one way or another, so the example is mainly provided as a caution for future tournaments to be sensitive to the difficulty of the game evolutions. The new features do not always have to make the game significantly harder, just provide enough of a challenge to require multiple teams to solve at once.

Execution Time Limit Some solutions to the item location challenge resulted in long computation times that exceeded the capabilities of the SPHERES processor. The problem was not discovered until late in the tournament when a test session was performed on the ISS using student-developed code. Large gaps in telemetry were traced to computational overruns in each cycle of the user program. The issue led

to hasty development of a profiling tool for measuring computational performance in a single cycle of the code. All future tournaments should include code profiling in preparation for the ISS phase, and it is recommended that teams get a sense for computational limits early in the season, though requiring code profiling early in the season is probably over restrictive. See Section 3.3.8 for details of the profiling tool.

Leaderboard This season was the first high school season to use a live leaderboard scoring system. A full analysis of the leaderboard and its programmatic effects are discussed in Chapter 4.

2.4.7 Evaluation

Periodic impact evaluations and feedback surveys are essential to the principle of *Engage and Educate* because they help to determine if the program is meeting its educational objectives, highlight potential problems, and help to guide future development. Most of the Zero Robotics evaluations to date have focused on soliciting platform-specific feedback from competitors to better enhance the experience of participating in the tournament. Many of the lessons from the tournament summaries in the preceding sections have been communicated by users from short answer survey questions. Formal studies for quantitatively measuring the impact of Zero Robotics on targeted STEM subject areas remain as future work (see Section 6.3.3), but the sections below provide some initial observations.

2.4.7.1 Participation and Attrition

A coarse view of team participation statistics between years and during the tournament season helps to track program growth and identify potential problems. Table 2.4.1 shows team information across all four high school tournament seasons of Zero Robotics. The first column shows overall team registration for the tournament measured by the total number of teams were created upon registration approval. The second column tracks the number of returning teams as a percentage of the previous year's registered teams. The third column indicates the number of teams that cre-

Year	Teams Registered	Returning	Created Proj.	2D	3D
ZRHS 2009	2	-	2	-	-
ZRHS 2010	24	2	24 (100%)	22 (92%)	22 (92%)
ZRHS 2011	147 (125 US / 22 EU)	16 (67%)	135 (92 %)	87 (59%)	91 (62%)
ZRHS 2012	143 (96 US / 47 EU)	51 US (42%)	137 (96%)	94 (66%)	88 (62%)

Table 2.4.1: Team participation for the four years of the Zero Robotics High School tournament. Percentages are with respect to the total number of teams registered. Even though nearly all teams create at least one project, there is a high attrition rate at the first submission deadline.

ated at least one project during the season, and the fourth and fifth columns show the number of teams that entered a submission for the 2D and 3D phases.

Despite significant growth between the closed pilot program and the initial open registration tournament in 2011, the overall registration remained flat between 2011 and 2012. The number of returning teams suggests that a low retention rate is a significant problem. In 2012, 45 new teams were created in the US, but 74 teams from 2011 did not register for the new season[2].

Attrition is also high during the season. In both the open registration years, only about 60% of the teams made a submission in the 3D phase before the first elimination round. Although it appears most teams that register create at least one project, the steep drop in participation occurs at the first submission. Nearly all of those that remain are able to make the next submission. This indicates that the biggest hurdle is difficulty getting started with the program.

Both statistics show that Zero Robotics is able to attract new teams to participate but needs careful attention to supporting new teams and retaining old ones. Retention can be improved by staying in contact with old teams, and targeted surveys to find out why teams are departing. Additional support for both groups comes from additional attention to *Accessibility*, especially improving learning resources like tutorials.

[2]Returning team information was not available for the EU teams. The percentage in the table is calculated with respect to the number of US teams in 2011.

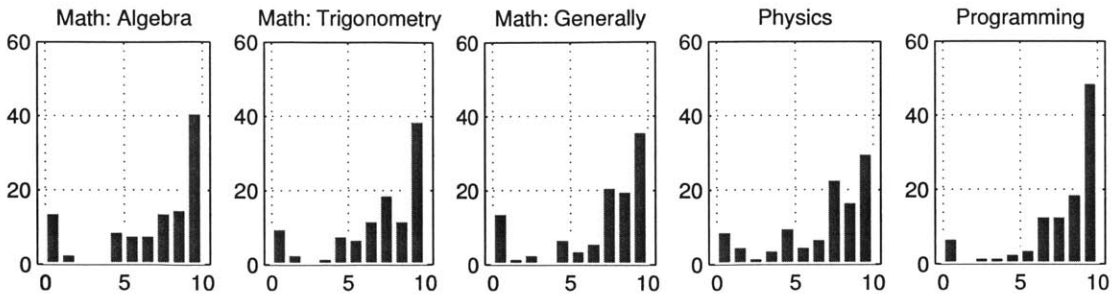


Figure 2.13: Histogram of survey responses for: *On a scale of 0-10, rate your confidence that Zero Robotics has prepared you for further study in the following subjects.* Among 113 surveyed students from 36 teams, at least 90% indicated a benefit for each subject and at least 60% indicated a strong benefit (8, 9, or 10).

2.4.7.2 Subject Area Preparation Survey

As part of the end of tournament feedback survey in 2012, students were asked to rate how confident they felt that Zero Robotics prepared them for future study in math, physics, and programming. The results, shown in Figure 2.13 suggest that most of the 113 surveyed students strongly believe the program prepares them for the subjects, especially in the area of programming. These initial findings are a positive sign that the program is having the desired impact and motivates more detailed study.

2.5 Summary

This chapter has presented frameworks for creating the two central features of a recurring robotics challenge: the game and the structure of the season. In both cases the guidelines have been specialized for the case of running a competition without physical hardware until the final event. In the game design, under the principle of *Authenticity*, hardware constraints are transferred into the virtual environment, and in some cases virtual limits are imposed to protect against difficult to simulate events. As with all game design, strategy balance is a critical consideration, but the short time for development and high complexity of the autonomous player solutions warrants extra attention to balancing efforts. A mechanism for continuing adjustments during the season helps to compensate when inevitable bugs surface.

Tournament design also prepares for hardware, following the principle of *Incremental Difficulty*, by gradually ramping up the challenge. The multi-week Alliance phase, unique among robotics competitions, gives students an extended experience with collaborative software design and is made possible by the virtual nature of the main competition season.For the crowning event of the season, the ISS finals, much has been learned about structuring time efficiently for a highly constrained time window, especially in appropriately scoping the expectations for the event.

From the participation among 6 challenges, and initial quantitative analysis, Zero Robotics is succeeding at bringing thousands of students to solve interesting problems while building confidence in STEM-related skills. Challenges remain to keep teams participating through the season and between seasons. The remaining chapters will examine how teams interact with the platform and several tools aimed at improving the Zero Robotics experience.

Chapter 3

The Zero Robotics Platform

3.1 Introduction

This chapter will present the design of the online platform created to host and run the Zero Robotics program. The platform consists of three main components:

1. A detailed simulation of the SPHERES satellites and internal software. It serves as the robot for most of the tournament.

2. A software Application Programming Interface (API) for simplified control of SPHERES and standardized implementation of Zero Robotics games, called the Zero Robotics API.

3. A web-based infrastructure for writing, compiling, simulating, and reviewing SPHERES programs.

The first component is central to carrying out the principle of *Authenticity*. The high fidelity model gives participants a realistic experience of working with the satellite hardware and helps to ensure their programs will work correctly on the real SPHERES. The second component bridges *Accessibility*, *Efficient Inquiry*, and *Incremental Difficulty* with a set of functions to control SPHERES at a wide range of skill levels. Users can begin with telling the satellite where to move and which direction to point, and proceed all the way to commanding applied forces and torques. The

last component centers on achieving *Accessibility* by making the platform available to any team with a modern web browser.

3.1.1 Contributions of Industry Partners

Many components of the current production architecture for Zero Robotics were developed in collaboration with industry partners TopCoder and Aurora Flight Sciences.. This section details the roles of the two partners.[1]

Aurora Flight Sciences

Aurora has served as a subcontractor to the Zero Robotics program since the ZRMS2010 Summer of Innovation challenge. Their primary responsibility has been to develop the graphical editing mode of the Zero Robotics Integrated Development Environment detailed in Section 3.6 and the 3D visualization in Section 3.7. For the 2010 nationwide pilot, Aurora worked with MIT to develop specifications for the graphical editor then delivered the editor as a standalone library, after which it was integrated into the website by the author. The 3D visualization was delivered as a prototype and was later enhanced by MIT with additional playback features.

TopCoder

TopCoder played an instrumental role in creating a scalable, cloud-based architecture based on the 2010 platform prototype. TopCoder's main approach to developing software is through a crowdsourcing model, where software developers compete to submit components at each phase of the software development cycle. The process began with MIT specifying a high level description of the desired website functionality, then proceeded through many competitions covering wireframe mockups of the website, themes, detailed software architecture, and finally, assembly of individual components. At the wireframe and theming level, MIT would define a set of desired layouts and general color descriptions, then choose among the submissions. During

[1]Components not explicitly noted in these sections were contributed by the author or others where noted.

the initial production site development, the architecture and assembly levels would usually flow down from the initial requirements, though the MIT reviewed most of the code produced and occasionally requested additional functionality. Later, as additional features like the Leaderboard scoring system from Chapter 4 were added, MIT would either directly implement the algorithm in the production codebase or launch a competition at the architecture and assembly levels to translate an initial specification to an implementation. The code was then reviewed and deployed to the production website by MIT.

3.1.2 SPHERES Software Architecture

This section presents a brief overview of components of the SPHERES software architecture that will be referenced throughout the chapter. The complete design of the software architecture is covered in [63].

Figure 3.1 summarizes the three main layers of the SPHERES software architecture. The lowest layer is a real-time proprietary kernel developed by Texas Instruments called DSP/BIOS[69]. It provides basic operating system features like hardware and software interrupts, threading, scheduling and prioritization, and concurrency constructs. There are three main priority levels, starting with hardware interrupts, triggered by pin-based hardware signals; software interrupts posted by hardware interrupts or the software scheduler; and tasks, computationally intensive procedures with less stringent real-time requirements.

The next level, known as SPHERES Core, is the satellite's operating system built on top of the generic DSP/BIOS services. SPHERES Core implements the generic routines for accessing all the satellite sensors and actuators and communicating with other satellites. A major component of SPHERES Core is the SPHERES standard estimator [55], which uses measurements from Inertial Measurement Unit (IMU) and ultrasound sensors to produce state estimates. It also serves as a buffer between DSP/BIOS and the higher levels of the software stack, exposing simplified methods for controlling the satellite through the Guest Scientist Program (GSP) API .

The final layer is dedicated to custom implementations provided on a test-by-test

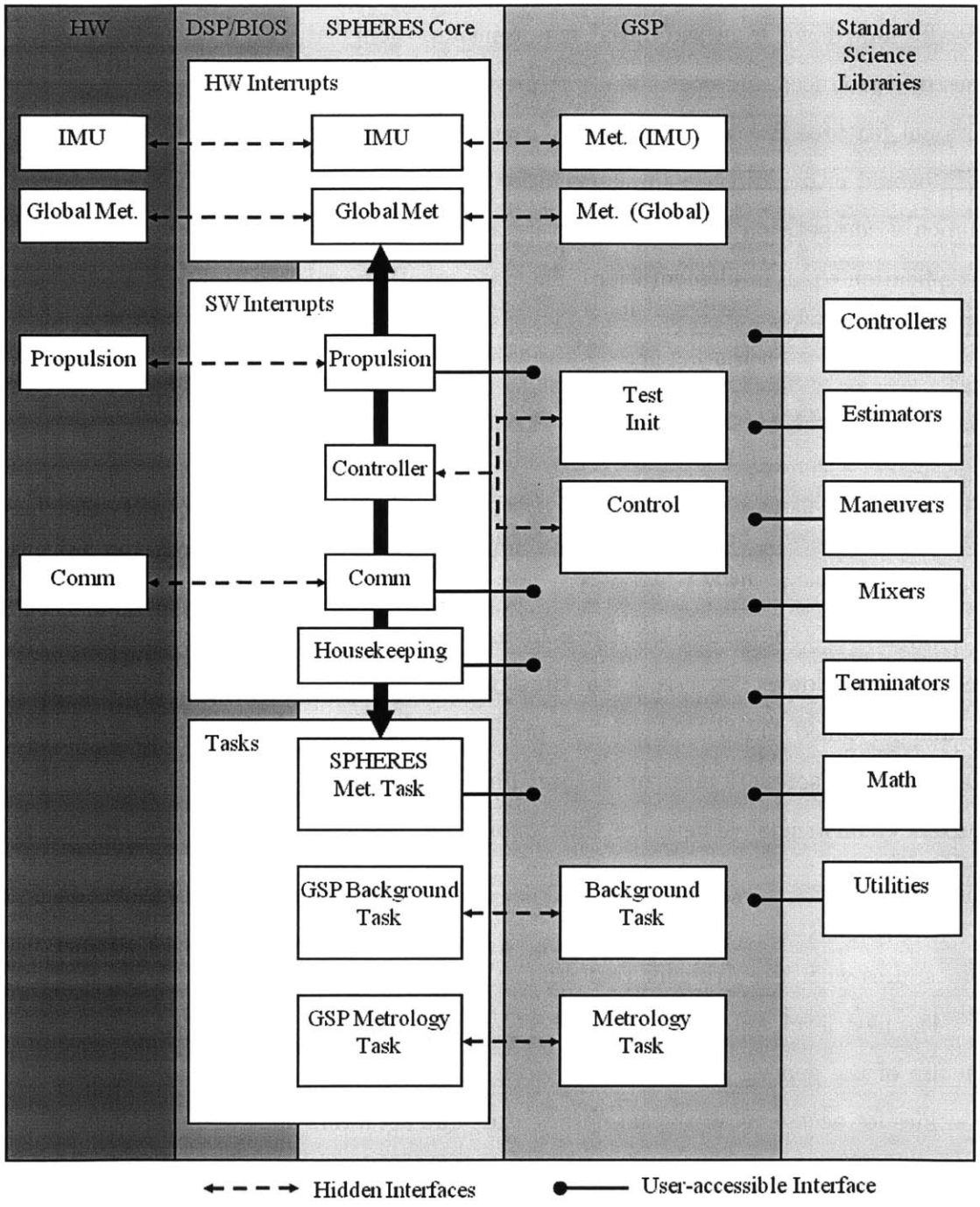


Figure 3.1: SPHERES has three main software layers: (1) a low level real-time operating system from Texas Instruments called DSP/BIOS, (2) a layer implementing the common routines and resources that run the satellite called SPHERES Core, and (3) a layer of test-dependent functions implemented by the guest scientist with the Guest Scientist Program (GSP) API. Image from [63].

basis by guest scientists using the GSP API. The main entry points to a SPHERES program are callbacks fired by SPHERES Core:

gspInitProgram() Called when the satellite first turns on

gspInitTest() Called when a test starts

gspControl() Called at each user-defined control period (typically 1 second).

Additional sections can be added to process high speed inertial sensor data and ultrasound measurements. The GSP API also includes a library of functions for performing common matrix/vector math operations and control laws. During and between control cycles, programs use a *maneuver number* as the state in a state machine to set behaviors at different phases of execution. The satellite also automatically tracks the time elapsed since the last maneuver change and the total elapsed test time. At the end of each control cycle, the GSP implementation makes final calls back to SPHERES Core layer to actuate thrusters and trigger additional cycles of the state estimation system.

3.2 Spheres Simulation History

The SPHERES simulation component of Zero Robotics is the product of many years of refinement, first driven by the requirements of the SPHERES graduate research team, then by the needs of the Zero Robotics platform. Under Zero Robotics, the simulation has had two major revisions, preceded by at least at least five versions of the SPHERES research simulation with varying levels of fidelity. Each iteration of the design has contributed design features to the current version used in the Zero Robotics program.

3.2.1 Common Features

All versions of the SPHERES simulation have shared several basic components. The high level data interfaces between these components are listed below and shown in Figure 3.2.

SPHERES Software Models the satellite's onboard processor and software, including SPHERES Core and the GSP API. The software component receives sensor measurements in the form of register values (a software model of an FPGA connected to analog sensors) and produces a set of thruster commands in the form of solenoid command bits. Separately, the software component represents communications through the SPHERES RF communication stack with a stream of communication packets, denoted by $\mathbf{x} \in \mathbb{R}^3$, $\dot{\mathbf{x}} \in \mathbb{R}^3$, $\mathbf{q} \in \mathbb{R}^4$, and $\omega \in \mathbb{R}^3$ respectively.

Dynamics Engine Simulates thrusters and rigid body dynamics. Upon activation of individual solenoids, the thruster model applies forces to the satellite body, resulting in linear and angular acceleration. The dynamics are integrated to produce a 13-element state vector containing position, velocity, quaternion, and angular rate states.

Sensor Model Simulates the onboard inertial measurement unit including 3 gyros and 3 accelerometers to produce $\mathbf{a}_{meas}$ and ω_{meas}, the true inertial measurements. The global metrology model simulates the transmission and reception of ultrasound pulses from fixed beacons in the testing area.

Timing Simulations timing models usually fall somewhere in the spectrum between discrete events dispatched by a scheduling engine and a series of continuous steps incremented by a simulation clock. Both ends have been used in versions of the simulation. The most recent version of the simulation is based on a fixed time step.

3.2.2 GSS, C GSP, and MATLAB Simulations

Three implementations of the SPHERES simulation were completed by other researchers. During the design and construction of the SPHERES flight hardware, the GFLOPS SPHERES Simulator (GSS) designed by Radcliffe [58] was built upon the real-time simulation framework GFLOPS developed by Enright [21]. The architec-

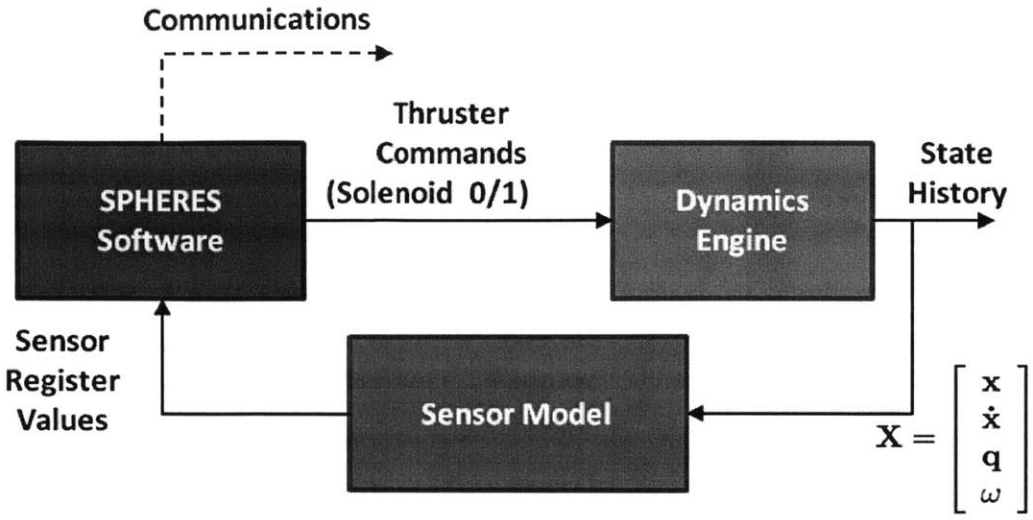


Figure 3.2: The SPHERES simulation has three main components: a model of the SPHERES internal software, a dynamics model, and a sensor model.

ture included multiple single-board computers running a real-time operating system, connected through an ethernet network. For each satellite SPHERES code was compiled into a wrapper and loaded into its own module. Additional modules provided dynamics, sensor, and communications components. See Figure 3.3 for an overview of the architecture.

While GSS provided a high fidelity model of the SPHERES distributed computing environment, its main application was the validation of flight code and required use of laboratory hardware. Overlapping with GSS and following completion of the SPHERES flight design, the Guest Scientist Program was created to make SPHERES more broadly accessible to the scientific community. Hilstad developed a simulation for personal computers to complement the GSP with the goal of making early algorithm development independent of the SPHERES team [32].

In this version, SPHERES code was again wrapped with a set of supporting functions, and each satellite executed in an independent process connected to a central server process. Shared signals sent between satellites for ultrasound metrology updates and communications passed through the central server using interprocess communication pipes. In contrast to GSS, dynamics were simulated separately in each SPHERES process. After a centralized simulation tick command from the server,

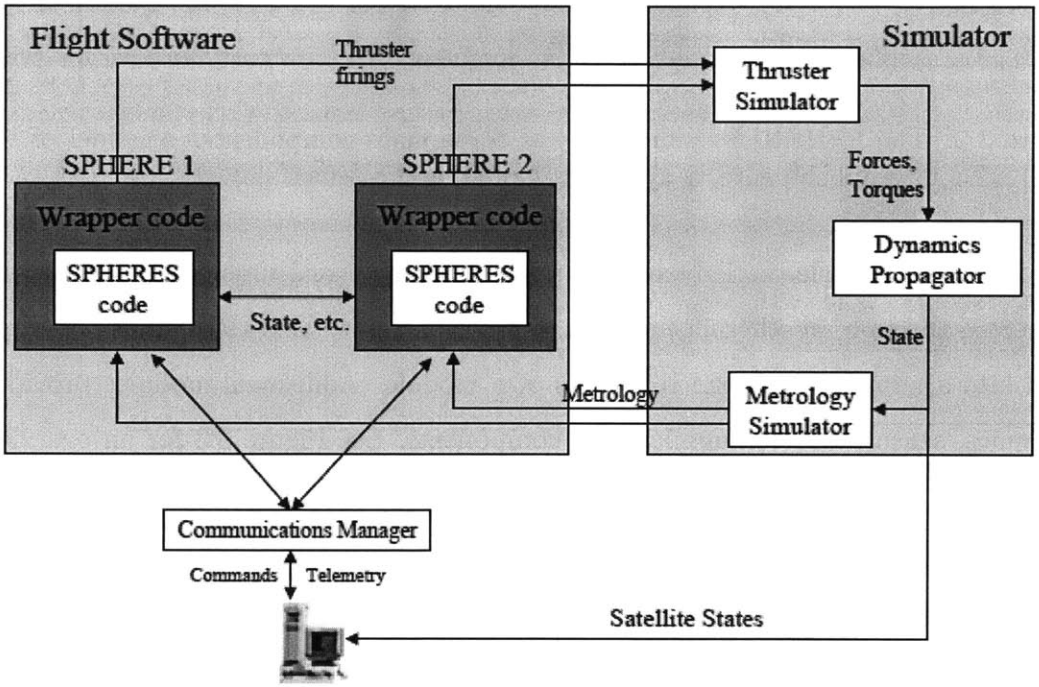


Figure 3.3: The GFLOPS SPHERES Simulator Architecture (GSS) used dedicated real-time computing hardware to model the satellite code and dynamics on independently executing modules. Graphic from Radcliffe [58].

each client advanced its state by one millisecond, then sent relevant status messages back to the server. This version of the simulation had the desirable features of running nearly the entire SPHERES Core implementation along with user code in its flight configuration while also being capable of running on personal computers. Data analysis took place after running simulations by parsing telemetry outputs from the simulation.

Following initial release of the GSP, extensive development and troubleshooting took place on the SPHERES metrology system, requiring detailed analysis of each operation in the onboard Extended Kalman Filter. For this effort, Nolet [55] designed a new version of the simulation in MATLAB [46], which modeled the metrology system in detail and approximated the remaining components with MATLAB-based functions. With the ability inspect states or variables of interest within SPHERES Core, extend the simulation with m-file scripts, and the overall accessibility of MATLAB, this version became the standard tool for preparing tests during ISS operations.

3.2.3 SWARM Simulation

Though the MATLAB simulation helped accelerate early prototyping, there were several major limitations. Most importantly, the simulation lost its Software-In-the-Loop capability, requiring a hand translation step from m-code to C prior to laboratory and flight tests. Mistakes in the translation were only evident during ground testing or in the worst case during ISS demonstrations. Any modifications to SPHERES Core required corresponding updates to the MATLAB supporting files, and the modified code could not be tested in simulation prior to use on the hardware. Overall, runtime performance of the simulation was also marginal, approximately one-to-one when running a full simulation of the estimator.

For the 2008/2009 Self-assembly Wireless Autonomous Reconfigurable Modules (SWARM) program, a ground-based demonstration of docking and assembly of flexible space structures [40], it was necessary to create a new simulation incorporating flexible dynamics into the simulation. With a heavy focus on laboratory testing, a key objective of the new simulation was the ability to move rapidly between simulation

testing and hardware. Dynamics and control logic were implemented in Simulink, and the automatic code generation capabilities of the Simulink Coder (formerly the Real Time Workshop) were used to translate control diagrams to hardware-ready C code. By reintroducing a Software-In-the-Loop capability, this approach dramatically improved turnaround time. The Simulink implementation also greatly improved the ease of modeling and switching between several concurrent configurations of the flexible system.

While it met the requirements for SWARM, the simulation had a significant limitation in the way it modeled SPHERES Core and the GSP, related to several mismatches between the SPHERES Core implementation and Simulink diagrams. The first limitation relates to the way concurrency is represented in Simulink. When running on the satellite DSP, SPHERES Core consists of several independent hardware and software interrupts as well as long-running tasks dispatched by a DSP/Bios. Modeling concurrent execution in Simulink usually involves placing several block diagram elements called *subsystems* at the same level in the diagram hierarchy and assigning each an inherent sample time. Unless deployed to a real-time operating system and specifically configured to respond to run concurrently, the Simulink engine runs from a single thread of execution. Calls to the subsystems are dispatched one at a time when a base simulation timer reaches a multiple of the subsystem's sample time. Algorithms and their associated code or block diagrams that run at different rates therefore *must* be placed in different locations of the diagram.

At the same time, as with most software APIs, SPHERES Core and the GSP are designed with a large library of function calls available, many of which access variables computed in different components of the software stack, some of which execute at different rates. In C this works well through the use of mutator functions or less desirably through shared global variables all accessible from the user's code. Simulink, on the other hand, assumes a somewhat rigid pre-definition of inputs and outputs at each level of the block diagram. To route information from one spot to another in the diagram requires either linking the subsystems by a signal connection or communicating through global datastore memory, both of which require, at a

minimum, loose specifications of data types and signal sizes. This approach benefits code generation applications because the data exchange interface is tightly controlled, but for APIs with many external calls, quickly becomes an impractical challenge of building a signal for every possible input and output required. Furthermore, since the generated code assumes an input-output format, a wrapper must be built that pre-populates the inputs with the API calls and reads the outputs at the end, also impractical for a large number of possible functions.

There are, of course, several ways to avoid the outcomes above, but most require a shortcut that bypasses the Simulink environment. Each method has its own limitations:

Re-Implement in Embedded MATLAB MATLAB/Simulink's code generation suite [71] includes the ability to write functions in a subset of the MATLAB language called *Embedded MATLAB*. Standalone library functions such as math routines or control algorithms can usually be translated to m-code functions and called from special user-defined MATLAB function blocks. Under normal execution, the simulation engine will execute the code in interpreted mode, and the Simulink Coder will generate C representations of the functions when the diagram is autocoded. It is also possible to make calls to functions located in C source code from the Embedded MATLAB blocks, but the calls can only be made after the source code has been generated. This approach stays within the Simulink hierarchy, but functions that require data transfers to or from other locations in the diagram still need a signal connection to carry the information.

Communicate Through the MATLAB Workspace Embedded MATLAB allows users to define *extrinsic functions,* explicitly indicated function calls that are only executed with the MATLAB engine. As illustrated in Figure 3.4, it is possible to store and retrieve variables from multiple places in the diagram with a set of extrinsic functions that replicate the desired API. However, because extrinsic functions cannot be autocoded, an API using only extrinsic functions will break the ability to generate code from the diagram. The solution is to implement

all simulated API functions with a switch that calls the extrinsic function when running under the MATLAB interpreter and calls the C API function when the diagram has been code generated. This dual-purpose API was used in the SWARM simulation to enable code generation and simulation under Simulink. This API eventually became part of the MATLAB-based implementation first used in Zero Robotics.

Communicate Through Shared Memory The last shortcut involves passing information through shared memory in an externally loaded library. Both MATLAB and Simulink support loading of C/C++ shared libraries. In Simulink, the shared libraries are called *S-Functions*, which are attached to special blocks in the Simulink diagram to implement low-level functionality. A single instance of the S-Function library is shared between all instances of the block in the diagram, and predefined gateway functions are called at model load, test start, and at each block sample time. Inside the library a global variable, singleton class, or shared memory region can serve as an conduit for data between areas of the diagram. Simulink Coder's Target Language Compiler (TLC) can also convert S-Functions to custom C code during code generation, and there is a utility called the Legacy Code Tool for wrapping existing C/C++ code with the S-Function gateway functions. While quite versatile, the main limitation of this approach is that it is quite tedious to implement. It would be particularly difficult to create a unique S-Function for every function in an API. More practically, like the MATLAB workspace method, the best approach is to use a single interface function as a wrapper around all code that needs to access the API. More details about this approach will be covered in Section 3.3.

Following the MATLAB workspace approach, the SWARM simulation created a full library of GSP and SPHERES Core API functions implemented in Embedded MATLAB with the ability to call their C counterparts when generated into C. Though the simulation was not used beyond the SWARM program, these libraries formed the foundation of the next iteration of the simulation.

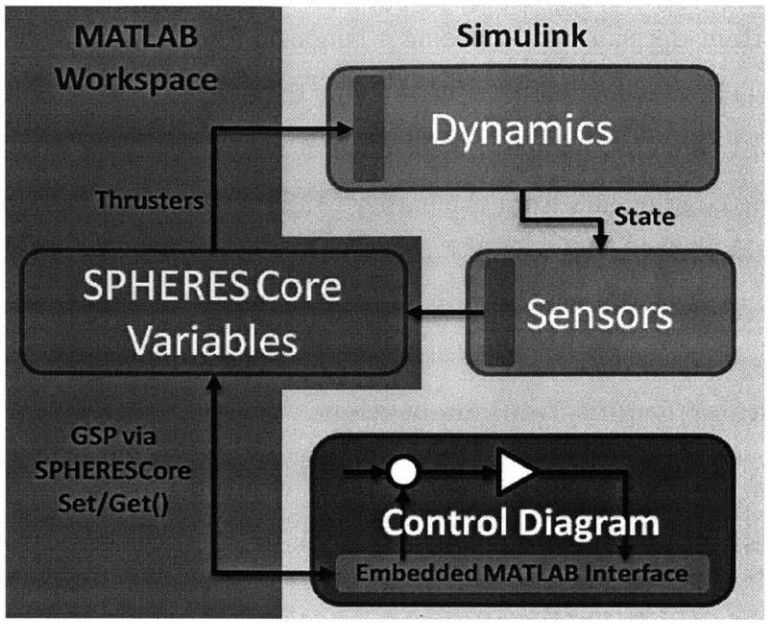


Figure 3.4: The 2008/2009 SWARM simulation utilized Simulink to append flexible dynamics to the SPHERES rigid body model. Due to inherent limitations of Simulink, calls to the GSP API passed through the MATLAB workspace.

3.2.4 v2009 MATLAB Engine

In 2009, a new effort began to combine the desirable features of previous simulation versions into a new implementation for general use by the SPHERES team and external researchers. From 2009-2012, this version was used by both the SPHERES team and the Zero Robotics platform for simulations. Going into the design process, the main objectives were:

- modularize the simulation components to promote extensibility,
- improve runtime performance to faster than real-time,
- re-introduce a Software-In-the-Loop pathway to testing code from the old GSP simulation,
- allow users to write programs for SPHERES in C or MATLAB, and
- add a fast 3D visualization for qualitative evaluation of performance.

The first important decision was choosing a language for the main engine to execute the simulation. The decision fell between reviving the GSP C/C++ simulation, adapting the SWARM Simulink simulation for general use, or building a new MATLAB simulation using components from the earlier MATLAB implementation. MATLAB was selected over C/C++ for broader accessibility to researchers and over Simulink to avoid many of the limitations discussed above.

3.2.4.1 Modular Engine Implementation

The simulation was implemented with an object-oriented discrete event framework with several basic components from which all other parts of the simulation extended:

Engine The core object linking all modules in the simulation. Manages the simulation time, dispatches simulation events, and collects common information for logging or transfer between modules.

Schedulable An object that can be added to the engine's schedule to receive a simulation event. Each schedulable object defines a list of events that it can respond to and their associated function callbacks.

Event A named signal triggered by the engine at scheduled times. Each event contains a reference to an instance of a Schedulable object on which to trigger the event.

To execute the simulation, an implementation of Engine is first instantiated and populated with instances of Schedulable objects. The clock begins when an event is posted to the Engine, which adds them to a priority queue sorted by the time of the event. The engine then executes the following loop:

1. Poll the event queue for any remaining events. If no events remain, terminate.
2. Check the time of the retrieved event, and advance the simulation time to this point.
3. Trigger the event on the specified Schedulable object.

4. Check for any simulation termination conditions, then loop back to beginning

In a simulation involving continuous dynamics, the discrete framework must be augmented to propagate the continuous equations of motion between discrete event times in Step 2.

In the SPHERES simulation, the main Schedulable objects were:

Dynamics Modeled the thrusters and 6-DOF rigid body motions of the satellite. The simulation engine expected an implementation of the Dynamics object to propagate continuous states.

Sphere Centralized data object for all parts of the satellite. Contained all components of the satellite including current state, variables for SPHERES Core, and communications information. Individual events and callbacks were defined for each of the basic SPHERES Core interrupts.

Beacons Modeled the SPHERES global metrology system. Events corresponded to triggering the ultrasound estimation system and the measurement receive times.

Animation Base object for implementing visualizations. Events triggered display refreshes with new data from the simulation, allowing visualization during execution.

During the implementation it became apparent that achieving satisfactory runtime performance would still be problematic, and several helpful optimizations were introduced. First, the priority queue at the heart of the engine was particularly slow due to adding, removing, and sorting operations, all slow when implemented in the MATLAB engine. Much better performance was achieved by moving this functionality to a Java library, then loading the library into a MATLAB wrapper. Next, to avoid fine-grained updates to the continuous dynamics, events were further classified into *normal* and *dynamic* events. Normal events did not require the most recent satellite state, such as sending a communication packet or triggering the start of the global metrology cycle. Dynamic events indicated to the engine to propagate the dynamics forward to the latest event.

While many bottlenecks were eliminated, the most difficult component to simplify was the model of the SPHERES estimator. To accurately model the operation of the estimator, the satellite must receive IMU updates at 50 ms intervals as well as ultrasound measurements every 200 ms for 9 beacons spaced at 20 ms intervals. The combination of propagating the simulation to each of these dynamic events, and the computational tasks performed at each event slowed the simulation considerably. To meet the objective of improving runtime performance, a switch was added to the simulation to enable a "fast mode" where the satellite's estimated state was replaced with the true state directly from the dynamics propagation. Basic algorithmic testing could be performed in fast mode, then checked out for ISS testing using the estimator model. The addition of this feature proved to be critical for meeting *Efficient Inquiry* objectives in the early stages of Zero Robotics, though it came at the cost of reducing the authenticity of the simulation available to participants.

3.2.4.2 SIL Implementation

The most important feature of this version of the simulation was the reintroduction of a Software-In-the-Loop approach to testing SPHERES projects. The simulation design included two pathways for moving between SPHERES flight code and simulation without modifying algorithms: generating code from MATLAB or loading a standard C GSP code template. Figure 3.5 gives an overview of the implementation. For either pathway, when the appropriate SPHERES Core interrupts fired in the schedule they triggered a special engine command for calling the researcher code, callGspMain(). Based on the currently configured mode, m-code or compiled C code would be executed for the standard GSP routine identified by a function identifier.

The MATLAB code generation pathway utilized the GSP and SPHERES Core m-code libraries developed for the SWARM Simulation. Instead of representing the researcher's algorithm as a block diagram in Simulink, Embedded MATLAB functions for each of the standard GSP gateway functions were created. The functions ran in interpreted mode while executing in the MATLAB environment but could be autocoded into hardware-ready code with the MATLAB Coder.

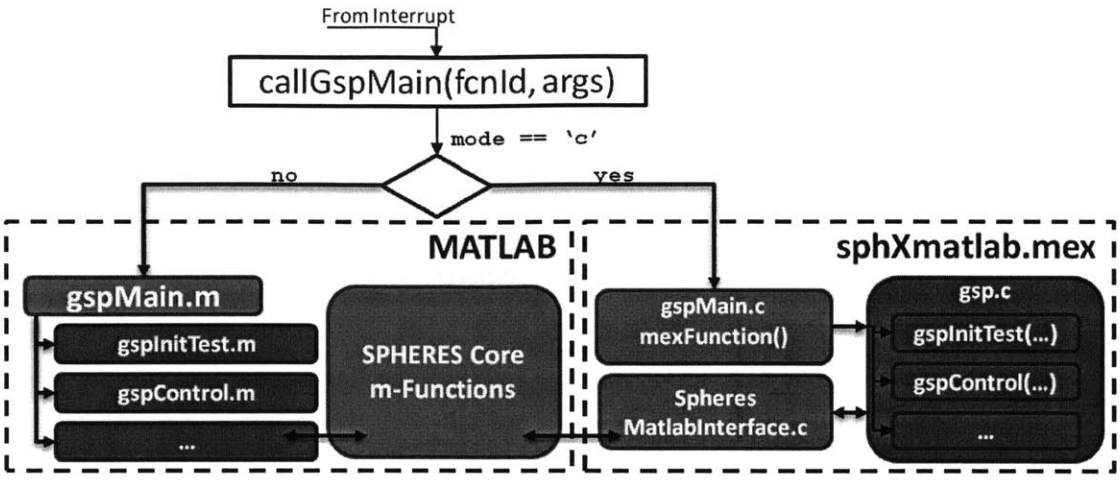


Figure 3.5: The 2009 MATLAB simulation had two pathways for Software-In-the-Loop testing that could produce flight code. The MATLAB option used a set of GSP m-functions with direct calls to SPHERES Core while the C option communicated with MATLAB through a wrapper.

To make C code callable from the MATLAB engine required a MATLAB *MEX function*, another form of shared library designed to mimic a call to an m-function with C/C++ . MEX functions have a single gateway, mexFunction(), activated when a MATLAB command is issued with the same name as the MEX file. For the 2009 MATLAB simulation, a MEX interface followed the same pattern as gspMain.m function to dispatch calls based on function identifier to the correct GSP function residing in the compiled gsp.c. A second interface, SpheresMatlabInterface.c, replicated most the of the SPHERES Core API as a set of wrapper functions around calls back to the MATLAB implementation.

3.2.4.3 Limitations

Despite significantly improving the process of developing flight code for SPHERES, the 2009 MATLAB simulation had several significant limitations:

Speed As indicated previously, running the simulation with a complete model of the estimation system resulted in very slow performance. For use in Zero Robotics, the performance was slow enough to favor running the simulation without the estimator in the online interface to improve *Efficient Inquiry*. SPHERES re-

searchers also rarely used the estimation model in the simulation.

SPHERES Core Simulation While the SIL upgrades enabled researcher code to execute in a flight-like configuration, SPHERES Core remained a simulated component in the MATLAB environment. For the sake of execution time or for ease of implementation, small differences existed between the C implementation and the MATLAB implementation. For the best simulation fidelity, it would have been best to model the complete satellite software in C.

Code Generation Inefficiency While the capability existed to generate flight code from MATLAB implementations, the functionality was only used in one ISS test session [50]. Compared to hand-written code, MATLAB's code generation tools can generally produce *faster* executing algorithms, but code size tends to be much larger. Just as it is a dominant theme in Zero Robotics games, the extremely limited Flash program memory available on SPHERES (approximately 230KB total) limits the efficacy of code generation. This is particularly true when implementing algorithms containing a large number of basic matrix and vector operations because each operation is expanded into a series of *for* loops[2]. While it is possible to teach a user to write m-code that generates a more efficient C code, users tended to gravitate toward using C code from the start of the implementation.

3.2.5 Overall Lessons

Table 3.2.1 summarizes the chronological design of the SPHERES simulation from the GSS simulation through the current implementation. The arc of simulation implementations started with very high fidelity models in C/C++, moved to simplified versions in MATLAB, and has returned to high fidelity models and C/C++ code with more accessible interfaces and better performance. The development path highlights several important lessons for the design of easily accessible research simulations:

[2]More modern versions of the code generation tools allow the replacement of vector and matrix operations with custom replacement functions to address some of the inefficiencies.

Version	Architecture	Timing	User Code	SPHERES Dynamics Core	
GSS	Independently executing modules on real-time OS	Discrete interrupts, propagated to most recent thruster edge	C/C++	C with C++ wrapper	C++
GSP	Server and individual satellite clients	1 ms time step, commanded by server	C/C++	C with C++ wrapper	C++
MATLAB	MATLAB m-file scripts	Discrete, pre-configured schedule, discrete dynamics	MATLAB	MATLAB	MATLAB
SWARM	Simulink diagram, autocoded controller	1 ms simulation step	Simulink / Embedded MATLAB	Calls via MATLAB Workspace	Simulink
v2009	MATLAB m-file scripts with dynamic loading	Discrete events with dynamic schedule	C	MATLAB via C wrapper to	MATLAB
v2012	Multithreaded C++ library commanded by MATLAB wrapper	1 ms time step	C/C++	C/C++	Simulink generated to C++

Table 3.2.1: Chronological view of SPHERES simulations. More recent versions have switched from MATLAB to a primarily C++ implementation for performance advantages.

- Modeling the true real-time behavior of the satellites in GSS came at the cost of requiring a highly specific laboratory configuration. The GSS version would not have been suitable for a highly accessible platform unless many instances of the workbench could have been connected to the online platform. For the purpose of a broadly accessible platform, a simulation should be designed to run on general purpose computing platforms, and ideally across operating systems.

- The simplified MATLAB implementation of the simulation became favored over the GSP simulation for ease of use but at the ultimate cost of a SIL capability and a significant decrease in fidelity. For research simulations, especially involving complicated code bases, simplified interfaces to analysis tools must be maintained along with efficient ways to change the model structure to adapt to new requirements. The SWARM simulation used Simulink block diagrams, and the 2009 simulation used an object-oriented framework in MATLAB.

- Some form of SIL capability is essential for eliminating mistakes in the translation of code. As shown by the SWARM simulation can significantly improve the efficiency of iterative laboratory testing.

- Simulation speed and accuracy are closely linked and introduce a tradeoff between *Efficient Inquiry* and *Authenticity*. For example, first implementation of the Zero Robotics simulation (based on v2009) focused on capturing the main dynamic behavior of the satellites but eliminated the estimation model in favor of speed.

The final major iteration of the simulation incorporates these lessons for a fast and accurate model of SPHERES.

3.3 Detailed Design of Current Simulation

The most recent implementation of the SPHERES simulation addresses speed and SPHERES Core simulation limitations of the previous version by moving all satellite

flight code to C/C++-based libraries. This change drastically improves the speed of the estimator and allows it to run with the exact same code as the hardware satellites. The simulation has also returned to a design based on a Simulink block diagram which improves execution performance and allows the entire simulation to be generated into independent C/C++ source code, further accelerating performance and enabling many options for distribution. The following sections cover the detailed design of the simulation and how it resolves many of the issues from previous versions of the simulation.

3.3.1 Top Level Block Diagram Layout

Figure 3.6 illustrates the top level layout of the simulation block diagram model. Starting from the left side of the model, the Global Metrology module simulates timing information for the ultrasound global positioning system, covered in Section 3.3.4.3. The Satellites & Payloads section consists of 3 duplicate satellite models connected to 3 payload systems. The payloads can create external forces and torques on the satellite dynamics an supply baud-limited UART data to the satellites. Each of the duplicate models can be switched on or off to simulate from 1 to 3 satellites simultaneously.

As with many of the previous versions of the simulation, the block internals of the satellite subsystems are divided into the three main components shown in Figure 3.2. The dynamics and measurement models are both implemented with Simulink block diagrams, but the SPHERES software is executed entirely in C/C++.

In the Termination Conditions section, the outputs from all satellites are monitored for errors or test termination signals, and the Termination block will end the simulation if any of the conditions for test end are detected. The Simulation Outputs section can be optionally enabled to record high frequency information for verification of the simulation, and the output port produces simulation data available to external programs calling the simulation.

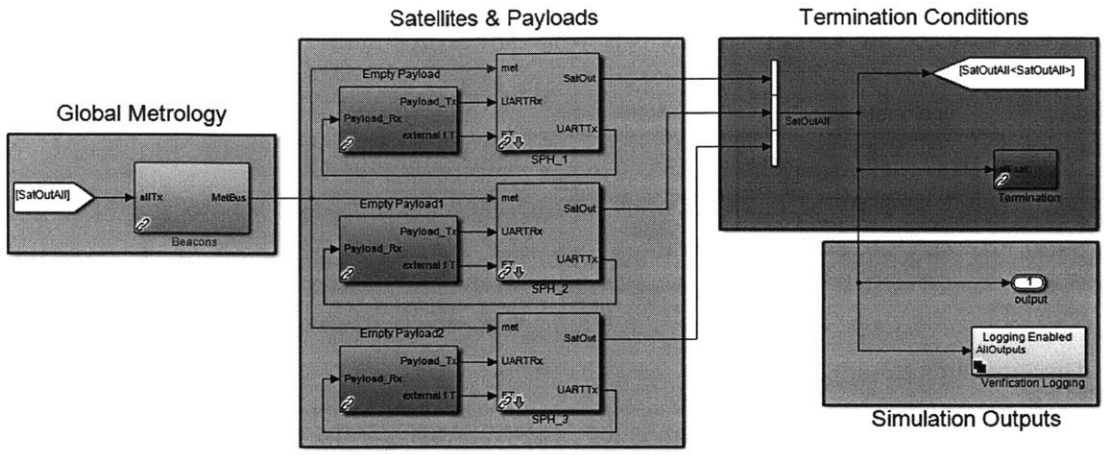


Figure 3.6: At the top level, the SPHERES simulation consists of a timing module for ultrasound global metrology, a set of duplicated systems representing each satellite, and subsystems for detecting termination conditions.

3.3.2 Repeatable Seeding of Random Variables

Simulations often contain many random sources to realistically model sensor noise and perturbations to dynamics. Random sources are usually based on a pseudo-random number generator initialized with a unique seed that determines the sequence of numbers produced by the generator. To explore new random realizations at each simulation run the seeds are usually generated by another random source. In other cases, it is sometimes necessary to exactly reproduce a simulation using the same sequence of random numbers, most frequently when locating bugs in user code that may be dependent on an exact state of the simulation. The SPHERES simulation uses an efficient method to recover the state of the random number generators by storing the seeding information along with the telemetry produced by the simulation. Instead of storing a long list of seeds for the generators, the simulation uses a single 32-bit integer seed for each satellite to generate a list of additional 32-bit seeds. A configuration file keeps track of which seeds in the list are allocated to corresponding random number number generators. New random elements added to the simulation simply append additional seeds to the list for generation. Once seeded, Simulink ensures that the random number generation runs consistently across platforms by

generating the random number generation algorithm along with the code for the diagram.

Under a single environment such as MATLAB, generating a list of random integers for the seeds can be performed in a repeatable way as long as the same algorithm is used. However, in other environments, random number implementations tend to vary wildly, even under the same general algorithm name. Since the Zero Robotics version of the simulation is intended to be used outside of MATLAB, and future versions of the SPHERES simulation may be distributed separately from MATLAB, a simple, repeatable algorithm for generating the seeds is used instead of relying on an inconsistent generation method. The algorithm is a Linear Congruential Generator (LCG) , which has the general form [57]

$$I_{j+1} = (aI_j + c) \mod m \tag{3.1}$$

where a is a multiplier of the initial seed I_j and c increments the result, followed by a modulus operation. LCGs tend to have poor properties when used for applications like Monte Carlo integration, but for generating other seeds, it is extremely simple to implement correctly in many languages. For 32-bit math, with the natural modulus of $m = 2^{32}$, [57] recommends $a = 1664525$ and $c = 1013904223$. Therefore to generate seeds for the simulation, the following steps can be performed:

1. For each satellite, select an initial 32-bit seed I_0 using a native random number generator. This is the only value that needs to be stored to reproduce the simulation.

2. Generate element j in the seed list iteratively using $I_j = 1664525 * I_{j-1} + 1013904223 \,\%\, 2^{32}$

3. Repeat until the algorithm generates the number of seeds specified in the configuration file.

In C the modulus operation is free due to the natural overflow behavior of integer math. The process has been tested for consistent generation in Java, JavaScript,

C/C++, Python, and C# and should have analogs in nearly every language. This means that any wrapper interface serving as a front end for generated code from the simulation should be able to repeatably generate random seeds for the simulation.

3.3.3 Dynamics

The simulation operates at a 1 ms time step to match the frequency of the main clock tick inside of SPHERES Core and the fastest rate at which commands to the thrusters can change value. Since the time step of $h = 0.001$ is fixed and very short, a 4th order fixed-step (Runge-Kutta) solver with an accumulated error of $O(h^4)$ is used. The time step and integration are also more than sufficient to integrate quaternions (see Equation 3.12) without accumulating significant errors due to re-normalization [1].

The satellites are individually modeled as six degree of freedom rigid bodies propelled by 12 individual thrusters. Thruster commands enter the dynamics module as a vector $\mathbf{u}$ where each element

$$u^{(i)} = \{0, 1\} i = 1, \ldots, 12 \tag{3.2}$$

is a binary value indicating if the thruster is on or off. In most simulations, it has been assumed that the thrusters instantaneously reach their full thrust after a configurable delay with a negligible transient[3]. More detailed models may be appropriate for future work, as discussed in Section A.2.3.

Converting the thruster on/off values into forces and torques is a two step process. First, the binary vector is scaled to account for the total number of thrusters activated. According to Chen in [9], there is approximately a 6% drop for each additional thruster

[3]The delay must be at least 1 ms to break an algebraic loop from the dynamics to the sensor modules through SPHERES Core and back to the dynamics. The one step delay is appropriate here because there is some delay between commanding the thrusters to open and the thrusters reaching full force.

opened after the first:

$$\mathbf{u}_{scaled} = \mathbf{u} \cdot 0.94^{n-1}. \tag{3.3}$$

where n is the total number of thrusters opened. To model random variations in thrust, additive noise is applied with a uniform distribution. A multiplicative, uniform random perturbation of $\pm 5\%$ of the nominal thrust value is applied as suggested by [55] in the first MATLAB simulation. The noise factor q_{thrust} is only changed once each time an individual thruster transitions from closed to open.

$$q_{thrust} = 1 + U(-0.05, 0.05) \tag{3.4}$$

$$u^{(i)}_{noisy} = u^{(i)}_{scaled} \cdot q_{thrust} \tag{3.5}$$

At this point, the thrust vector has been scaled to represent a ratio of its nominal value. The next step converts the values into forces and torques by multiplying them with a thruster matrix $\mathbf{T}$ where each column $\mathbf{t}^{(j)}$ is defined

$$\mathbf{f}^{(j)} = \begin{bmatrix} f^{(j)}_x & f^{(j)}_y & f^{(j)}_z \end{bmatrix}^T \tag{3.6}$$

$$\tau^{(j)} = \mathbf{r}^{(j)}_{cm} \times \mathbf{f}^{(j)} \tag{3.7}$$

$$\mathbf{t}^{(j)} = \begin{bmatrix} \mathbf{f}^{(j)} \\ \tau^{(j)} \end{bmatrix} \tag{3.8}$$

where $f^{(j)}_{\{x,y,z\}}$ are the body-frame thrust direction and magnitude of the j^{th} thruster, and $\mathbf{r}^{(j)}_{cm}$ is the location of the thruster with respect to the center of mass. In the SPHERES dynamics model, the thruster location includes the ability to define an offset pointing from the center of mass to the geometric center, denoted by $\mathbf{r}_{gc}$. The complete thruster location is defined as

$$\mathbf{r}_{cm} = \mathbf{r}_{gc} + \mathbf{r}_t. \tag{3.9}$$

The term $\mathbf{r}_t$ is the location of the thrusters with respect to the geometric center,

and $\mathbf{r}_{cm}$ is the location with respect to the center of mass. For normal SPHERES operations, the geometric center is assumed to be co-located with center of mass. A table of thruster locations is located in B.1.

Multiplying $\mathbf{T}$ with $\mathbf{u}_{noisy}$ produces

$$\begin{bmatrix} \mathbf{f} \\ \tau \end{bmatrix} = \mathbf{T}\mathbf{u}_{noisy} \tag{3.10}$$

a vector of forces and torques acting on the center of mass of the satellite body. From here the equations of motion can be integrated as shown below.

$$\dot{\omega} = \mathbf{J}^{-1}\left(\tau - \omega \times \mathbf{J}\omega\right) \tag{3.11}$$

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{\Omega}(\omega)\mathbf{q} \tag{3.12}$$

$$\ddot{\mathbf{x}} = \mathbf{R}(\mathbf{q})\frac{\mathbf{f}}{m} \tag{3.13}$$

This formulation assumes a quaternion representation of attitude of the form $\mathbf{q} = \begin{bmatrix} \mathbf{e}\cos\frac{\theta}{2}, & \sin\frac{\theta}{2} \end{bmatrix}$, with the scalar part as the fourth element. Equation 3.11 is Euler's equation of motion for propagating body frame angular velocities with inertia matrix $\mathbf{J}$. Equation 3.12 describes the quaternion propagation equations where $\mathbf{\Omega}(\omega)$ is the skew-symmetric matrix

$$\mathbf{\Omega}(\omega) = \begin{bmatrix} 0 & \omega_z & -\omega_y & \omega_x \\ -\omega_z & 0 & \omega_x & \omega_y \\ \omega_y & -\omega_x & 0 & \omega_z \\ -\omega_x & -\omega_y & -\omega_z & 0 \end{bmatrix}. \tag{3.14}$$

Finally, Equation 3.13 represents the satellite's double integrator translational dynamics expressed in the global frame. The rotation matrix $\mathbf{R}(\mathbf{q})$ uses the current

quaternion to translate body frame thruster forces into the global (inertial) frame.

$$\mathbf{R(q)} = \begin{bmatrix} q_4q_4 + q_1q_1 - q_2q_2 - q_3q_3 & 2(q_1q_2 - q_3q_4) & 2(q_1q_3 + q_2q_4) \\ 2(q_1q_2 + q_3q_4) & q_4q_4 - q_1q_1 + q_2q_2 - q_3q_3 & 2(q_2q_3 - q_1q_4) \\ 2(q_1q_3 - q_2q_4) & 2(q_2q_3 + q_1q_4) & q_4q_4 - q_1q_1 - q_2q_2 + q_3q_3 \end{bmatrix} \tag{3.15}$$

where $q_i\, i = 1, \ldots, 4$ denote the components of the quaternion $\mathbf{q}$.

To complete the dynamics calculations, the Equations 3.11 and 3.12 are passed through integrators, and Equation 3.13 is double integrated to produce the true states of the satellite. Assuming a given thruster configuration, mass, and inertia properties are correct, the dynamics are an exact (to numerical accuracy) representation of the satellites driven by noisy thrusters. Unmodeled components include:

- Aerodynamic forces due to airflow in the ISS test volume and plume impingement from other satellite thrusters. These effects are usually considered to be negligible compared to the thruster strength unless the satellite is freely drifting where at least plume impingement can have a significant effect on the satellite motion.

- The transient in thrust levels when a solenoid valve opens and closes. Several notes on this behavior are included in the Future Work under section A.2.3.

3.3.4 Sensors

This section presents the measurement and noise models of the gyros, accelerometers, and ultrasound metrology system. The models are presented with generic parameters, and Table 3.3.1 summarizes the assumed values used in the simulation.

3.3.4.1 Gyros

SPHERES satellites contain three rate gyros aligned with each of the body axes. To simulate the measurements, the gyro model first applies additive zero-mean Gaussian

noise to the simulation's true body rates.

$$\tilde{\omega} = \omega + \mathcal{N}(0, \sigma_\omega^2) \tag{3.16}$$

To model the analog to digital conversion of the SPHERES FPGA, the measurements are scaled by k_ω, then biased in units of counts by b_ω. The bias term represents both the center value of the ADC and the bias of the gyro. To represent an unanticipated gyro bias, the bias term is simply changed to a different value than the expected number in SPHERES Core. Following conversion to counts, the gyro samples are saturated to 12 bits (4095), corresponding to roughly $80\frac{deg}{s}$, then converted to unsigned 32-bit integers $\mathbf{z}_\omega$for transfer to the virtual FPGA.

$$\begin{aligned} \mathbf{z}_\omega &= \frac{\tilde{\omega}}{k_\omega} + b_\omega \\ \mathbf{z}_\omega &= (\texttt{uint32}) \begin{cases} \mathbf{z}_\omega & 0 \leq \mathbf{z}_\omega \leq 4095 \\ 4095 & \mathbf{z}_\omega > 4095 \\ 0 & \mathbf{z}_\omega < 0 \end{cases} \end{aligned} \tag{3.17}$$

The gyro model captures both discretization error and random noise and provides the SPHERES Core model with the same data values as the real satellite hardware. The model does not include a known high frequency ringing mode present in the gyro hardware or the frequency domain response characteristics. The ringing mode is filtered out with a notch filter and measurements are aggregated over a 50 ms period within SPHERES core, so there is a relatively small approximation error. Additional fidelity for future work might be required if simulations involving high speed inertial data are required. The only other potential inaccuracy is modeling misalignment of the gyro with the body axes of the satellite, but this error is assumed to be very small compared to the rotation rates for SPHERES.

3.3.4.2 Accelerometers

The accelerometer model has several additional considerations. First, the SPHERES accelerometers are not located at the center of mass, so there is a rotational coupling between acceleration and angular velocity. In vector form, the acceleration experienced by accelerometer i at radius $\mathbf{r}_i$ from the center of mass is

$$\mathbf{a}_i = \mathbf{a}_{sat} - \alpha \times \mathbf{r}_i - \omega \times \omega \times \mathbf{r}_i. \tag{3.18}$$

where $\mathbf{a}_{sat}$ is the acceleration of the satellite in the body frame, and α is the angular acceleration. Since each accelerometer only measures one axis, the final measurement is obtained by dotting the acceleration with a sensitivity direction $\hat{\mathbf{s}}_i$,

$$\mathbf{a}_{i,meas} = \mathbf{a}_i \cdot \hat{\mathbf{s}}_i. \tag{3.19}$$

The noise model also requires modification because there is a thruster-induced ringing each time a thruster opens or closes, as shown in Figure 3.7. This is modeled in the simulation as high variance random noise multiplied by a decaying exponential envelope, reinitialized each time a thruster changes value.

$$\sigma_a = \begin{cases} \sigma_{a,\text{on}} + \sigma_{a,ring} e^{-(t-t_0)/\tau_{ring}} & \text{thrusters on} \\ \sigma_{a,\text{off}} & \text{thruster off} \end{cases} \tag{3.20}$$

$$\tilde{\mathbf{a}}_{i,meas} = \mathbf{a}_{i,meas} + \mathcal{N}(0, \sigma_a^2) \tag{3.21}$$

The initial time of the decay envelope, t_0, is reset to the current simulation time each time a thruster changes value. The resulting model is a good fit to the observed sensor measurements, as shown in Figure 3.8.

As with the gyro model, the final measurement is scaled, biased, and saturated to model the ADC. The final vector $\mathbf{z}_a$ contains unsigned 32 bit integers for transfer to the virtual FPGA.

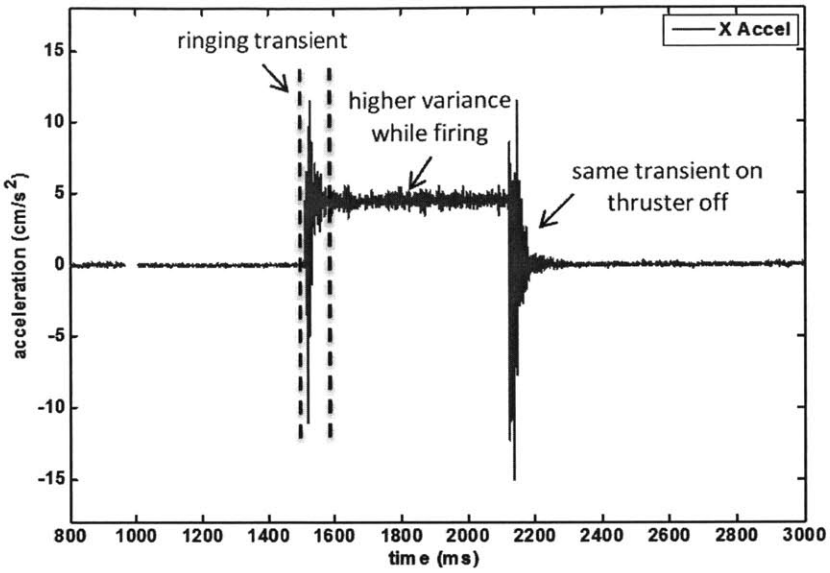


Figure 3.7: The SPHERES accelerometers have a noise response coupled to the firing of thrusters.

$$
\begin{aligned}
\mathbf{z}_a &= \frac{\tilde{\mathbf{a}}_{i,meas}}{k_a} + b_a \\
\mathbf{z}_a &= (\texttt{uint32}) \begin{cases} \mathbf{z}_a & 0 \leq \mathbf{z}_a \leq 4095 \\ 4095 & \mathbf{z}_a > 4095 \\ 0 & \mathbf{z}_a < 0 \end{cases}
\end{aligned} \tag{3.22}
$$

Like the gyro model, the accelerometer model presents correctly discretized measurement values to the SPHERES Core model, and the sensitivity and sensor location parameters give the model sufficient flexibility to represent sensor misalignments. The ringing noise model is only a rough approximation but should be sufficient for developing controllers with high speed inertial feedback based on the comparisons with flight data.

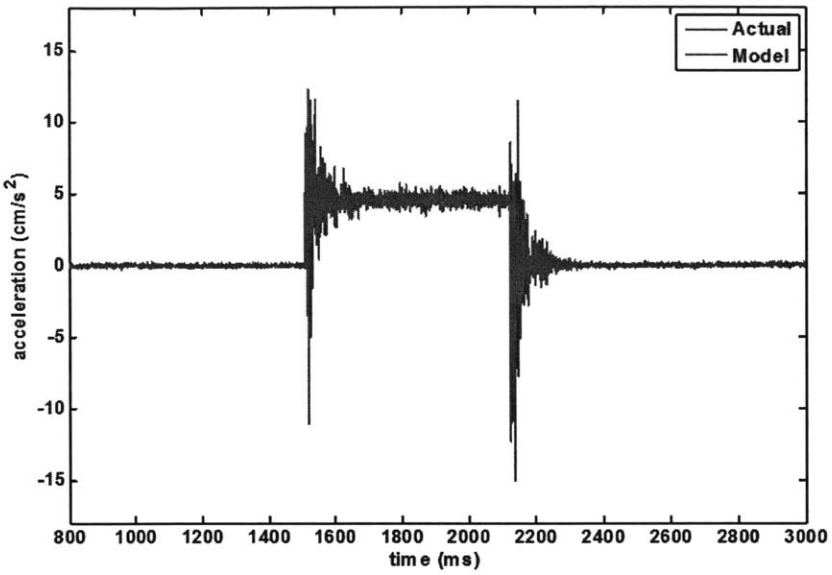


Figure 3.8: The accelerometer noise model uses a large noise magnitude multiplied by a decaying exponential to model the thruster transient.

3.3.4.3 Ultrasound Global Metrology

The SPHERES global metrology system consists of up to 9 wall-mounted beacons and 24 receivers attached to the faces of each satellite. A synchronizing pulse of infrared (IR) activates a schedule of ultrasound pulses, one per beacon, and the satellites process the measurements as they are received. The timing for the sequence of pulses follows the pattern shown in Figure 3.9. Each beacon is uniquely configured with an identifier that selects the time to issue the pulse relative to the initial IR flash. After a 10 ms hold, beacons transmit once every 20 ms, and the FPGA records receiver times during the first 10 ms after each pulse. Based on a speed of sound of approximately $343\frac{m}{s}$ this limits the range of the ultrasound system to at most $3.43m$.

In the simulation, the metrology system is modeled in two parts, one for transmitting, and one for receiving. On the transmission side, the logic follows the sequence described in Algorithm 3.1 where each iteration through the outer *while(true)* loop represents a 1 ms step of the simulation. When the $irTx$ transmission flag is set by any of the satellites, the counter *cycleCount* ticks off the initial 10 ms hold time, then the 20 ms periods between beacon transmissions. When *cycleCount* reaches zero at

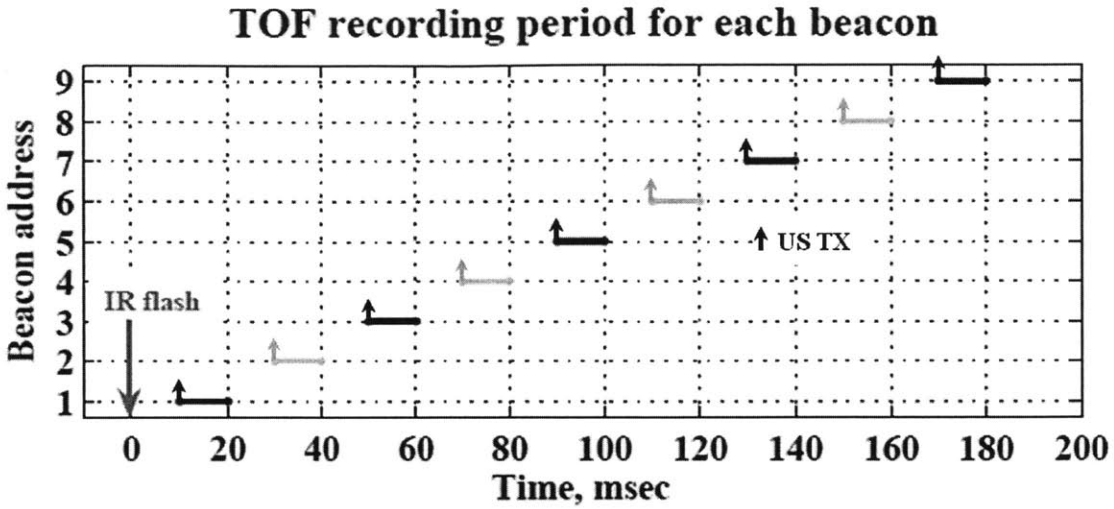


Figure 3.9: Ultrasound pulses in the global metrology system start 10 ms after the initial IR flash and repeat every 20 ms. The FPGA records Time of Flight measurements during the 10 ms period following each pulse, then flags the DSP to retrieve the measurements. Adapted from [55].

Line 9, the transmitter model produces two signals: $bcnNum$, the number of the currently transmitting beacon, and $usTx$, a flag set to 1 each time a beacon transmits. The process continues until the 9^{th} beacon completes its transmission then ends until the next IR pulse.

The receiver side of the model executes independently for each of the satellites in the simulation. Algorithm 3.2 summarizes the actions checked at each 1 ms step of the simulation. At Line 4, when the incoming $usTx$ flag from the transmitter module is set by an ultrasound transmission event, the receiver model immediately calculates the future time step at which the measurement will be received, t_{rx}. Since the satellite is not expected to move significantly over the 10 ms maximum receive window, *calculateRxTime()* on Line 5 estimates the reception time with the satellite position at the time of transmission, $\mathbf{r}(t_{tx})$.

$$t_{rx} = \frac{\|\mathbf{r}_{sat}(t_{tx}) - \mathbf{r}_{bcn}\|}{a} + t_{rx} \tag{3.23}$$

The beacon position $\mathbf{r}_{bcn}$ is fixed according to the beacon deployment in the virtual test volume, and a is the speed of sound.

Algorithm 3.1 Beacon Transmission Timing

```
1: started ← 0
2: while true do
3:    usTx ← 0
4:    if not started and irTx then
5:       started ← true
6:       bcnNum ← 0
7:       cycleCount ← 10
8:    if started then
9:       if cycleCount = 0 then
10:         bcnNum ← bcnNum + 1
11:         usTx ← 1
12:         cycleCount ← 20
13:      if bcnNum = 10 then
14:         started ← false
15:      cycleCount ← cycleCount − 1
```

Algorithm 3.2 Ultrasound Receiver Measurement

```
1: cycleCount ← 0
2: t_rx ← ∞
3: while true do
4:    if usTx then
5:       t_rx ← calculateRxTime()
6:       cycleCount ← 0
7:    if t ≥ t_rx then
8:       for all z_tof ∈ distVec do
9:          z_tof ← calculateTOF()
10:      t_rx ← ∞
11:   if cycleCount = 10 then
12:      usFlag ← 1
13:   cycleCount ← cycleCount + 1
```

On the time step where the simulation time, t, exceeds t_{rx}, the receiver model triggers the calculation of the beacon receiver vector, $distVec$. $distVec$ contains receiver measurements, d, for each of the satellite ultrasound receivers, typically 24 for a standard SPHERES satellite. The function *calculateTOF()* on Line 9 computes the time of flight from the beacon to the receiver and appends the receiver noise model. The noise model, originally developed and validated by Nolet [55], is implemented with the steps described in Algorithm 3.3 and described below. A diagram of the relevant transmitter and receiver geometry is shown in Figure 3.10.

Random Measurement Loss Approximately 3% of ultrasound measurements are corrupted or lost, registering 0 in the FPGA. On Line 1, the model generates measurement losses by comparing a randomly drawn number from the range [0,1] to the measurement loss probability.

Receiver Angle Bias Measurements have an angle-dependent bias, $b_{\theta,rx}$, based on the angle, θ_{rx}, between the relative vector, $\mathbf{r}_{rel}$, and the receiver normal, $\hat{\mathbf{n}}_{rx}$. The bias increases with the relative angle. Lines 4-7 calculate the receiver angle bias by rotating the body-frame receiver normals into the global frame, finding the receiver angle, then applying the bias. An additional random bias is either added or subtracted with equal probability based on the bias sign term, s_{rx}.

Transmitter Angle Bias On Lines 8-10, measurements are also biased by the transmission angle θ_{tx}. The bias, $b_{\theta,tx}$, is calculated in a similar manner to the receiver angle bias, this time using the angle between the beacon normal, $\hat{\mathbf{n}}_{tx}$, and $\mathbf{r}_{rel}$.

Distance Bias Lines 11-12 apply a bias base don the distance from the satellite to the receiver. The bias includes a 4^{th} order polynomial, b_{dist}, combined with a uniform random noise term, b_{noise}, increasing with distance. The polynomial is a fit to laboratory data collected from the receivers at increasing distances, and the noise term accounts for an envelope around the data. Coefficients for the polynomial are listed in Table 3.3.1. See [55], Appendix B for more details.

Random Noise After adding the measurement biases, an overall zero-mean Gaus-

Algorithm 3.3 Time of Flight Calculation

1: **if** $U(0,1) > P_{rx}$ **then**

2: $\quad \mathbf{r}_{rel} \leftarrow \mathbf{r}_{bcn} - (\mathbf{r}_{sat} + \mathbf{R}(\mathbf{q})\mathbf{r}_{rx,body})$

3: $\quad d \leftarrow \|\mathbf{r}_{rel}\|$

4: $\quad \hat{\mathbf{n}}_{rx} \leftarrow \mathbf{R}(\mathbf{q})\hat{\mathbf{n}}_{rx,body}$

5: $\quad \theta_{rx} \leftarrow \arccos \frac{\mathbf{r}_{rel} \cdot \hat{\mathbf{n}}_{rx}}{d}$

6: $\quad s_{rx} \leftarrow \operatorname{sgn} U(-1,1)$

7: $\quad b_{\theta,rx} \leftarrow \begin{cases} 0.007 + 0.001 s_{rx}\, m & |\theta_{rx}| > 35^o \\ 0.004 + 0.0005 s_{rx}\, m & 25^o < |\theta_{rx}| \leq 35^o \\ 0\, m & |\theta_{rx}| \leq 25^o \end{cases}$

8: $\quad \theta_{tx} \leftarrow \arccos \frac{-\mathbf{r}_{rel} \cdot \hat{\mathbf{n}}_{tx}}{d}$

9: $\quad s_{tx} \leftarrow \operatorname{sgn} U(-1,1)$

10: $\quad b_{\theta,tx} \leftarrow \begin{cases} 0.011 + 0.002 s_{tx}\, m & |\theta_{tx}| > 35^o \\ 0.005 + 0.001 s_{tx}\, m & 25^o < |\theta_{tx}| \leq 35^o \\ 0.0015 + 0.001 s_{tx}\, m & 15^o < |\theta_{tx}| \leq 25^o \\ 0\, m & |\theta_{tx}| \leq 15^o \end{cases}$

11: $\quad b_{dist} \leftarrow c_4 d^4 + c_3 d^3 + c_2 d^2 + c_1 d + c_0$

12: $\quad b_{noise} \leftarrow \begin{cases} 0.004 \cdot U(-1,1)\, m & d > 2 \\ 0.003 \cdot U(-1,1)\, m & 1 < d \leq 2 \\ 0.002 \cdot U(-1,1)\, m & 0.5 < d \leq 1 \\ 0.001 \cdot U(-1,1)\, m & 0 < d \leq 0.5 \end{cases}$

13: $\quad d' \leftarrow d + b_{\theta,tx} + b_{\theta,rx} + b_{dist} + b_{noise} + \mathcal{N}(0,\, \sigma_d^2)$

14: $\quad z_{tof} \leftarrow (\texttt{uint32})\, \frac{d'}{a} \cdot k_{tof}$

15: **else**

16: $\quad z_{tof} \leftarrow (\texttt{uint32})\, 0$

sian random noise term with standard deviation σ_d is applied to to the receiver measurement on Line 13.

The final step converts the biased and noisy distance measurement into FPGA counts, stored in the value z_{tof}.

3.3.5 SPHERES Software Simulation

The SPHERES Software component of the simulation models the execution of software running on the satellite's Texas Instruments TMSC6701 Digital Signal Processor (DSP). As discussed in Section 3.2, the software has been modeled at varying levels of detail, from duplication on a real-time operating system, to broad approximation

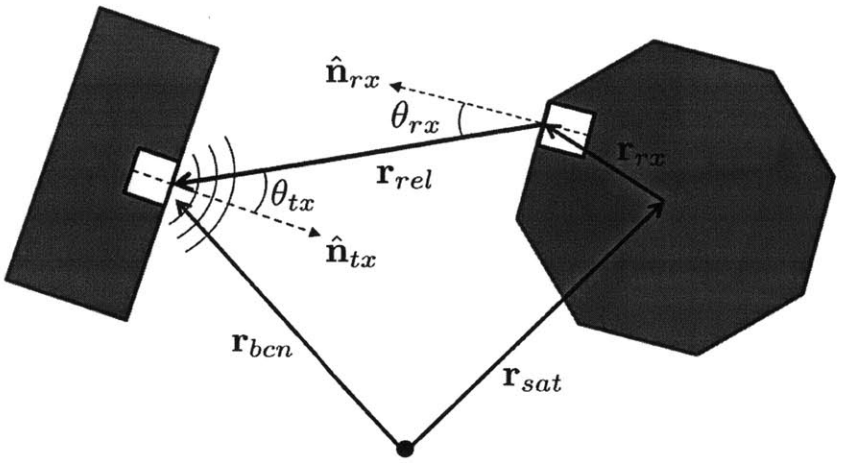


Figure 3.10: Ultrasound measurements are calculated from the time of flight to traverse the vector $\mathbf{r}_{rel}$. The receiver noise model uses the transmitter angle, θ_{tx}, and the receiver angle, θ_{rx}, to apply biases to the measurement.

Variable	Description	Typical
σ_ω	Gyro noise standard deviation	$0.003\,\frac{rad}{s}$
k_ω	Gyro scaling term (specific to each gyro)	$0.70799e-3\,\frac{rad}{s-count}$
b_w	Gyro bias term (specific to each gyro)	$2026\,counts$
$\sigma_{a,\text{off}}$	Steady state accelerometer standard deviation when thrusters are off	$0.0008\,\frac{m}{s^2}$
$\sigma_{a,\text{on}}$	Steady state accelerometer standard deviation when thrusters are on	$0.003\,\frac{m}{s^2}$
$\sigma_{a,ring}$	Maximum magnitude of noise during thruster transient	$0.05\,\frac{m}{s^2}$
τ_{ring}	Accelerometer ringing transient time constant	$0.0467\,s$
k_a	Accelerometer scaling term (specific to each accel)	$0.1152e-3\frac{m}{s^2-count}$
b_a	Accelerometer bias term (specific to each accel)	$2418\,counts$
P_{rx}	Probability of null ultrasound measurements	0.03
$c_{\{4,3,2,1,0\}}$	Polynomial coefficients for ultrasound receiver bias (function of distance between transmitter and receiver)	0.0004633 −0.0003565 −0.006537 0.01937 0.01051
σ_d	Ultrasound receiver noise standard deviation	$0.0033\,m$
k_{tof}	Ultrasound time of flight scale factor	$25\times10^6\,\frac{counts}{s}$

Table 3.3.1: Typical values for simulation noise and bias parameters.

in MATLAB scripts. One of the major contributions of the current version of the simulation is to strike a balance between these two extremes that preserves both high fidelity to the SPHERES software model and high performance.

The best possible software model would be to exactly replicate the operation of the SPHERES DSP in software. While cycle-accurate simulations of the C6701 DSP exist, execution time is slow enough that it is not practical to use in a simulation without severely limiting the platform principle of *Efficient Inquiry*. The next possible level of approximation is to attempt to simulate the behavior at an operating system level. In the current simulation, the onboard software is modeled down to basic operating system features such as tasks (threads), interrupts, and synchronization constructs, at which point simulated, platform-specific libraries are used instead of DSP/BIOS. At this level of detail the primary differences between satellite hardware and simulation have been reduced to the total execution time and the relative timing between concurrent threads. Considerations for both differences have been studied to close the remaining gap. The approach to modeling concurrent execution is described in Section 3.3.5.3, and execution time is discussed in Section 3.3.8. This approach reaches the lowest practical level of simulation accuracy, and the remaining discussion addresses achieving high performance with the selected implementation.

To enable the GSP and SPHERES Core APIs to communicate information between concurrent tasks as if they were executing on the satellite hardware, most of the actions related to SPHERES Software are modeled in C++ instead of in the Simulink block diagram. This is an implementation of the shared memory approach noted in Section 3.2.3. There are three main layers in the simulated software model:

S-Function Interface Interface layer between the Simulink model and the satellite code. This is the only layer that coordinates interactions between all the satellites, including communications and IR interrupts. On each simulation step, inputs and outputs in the form of pre-defined C data structures pass through the interface to the lower layers. As shown in Figure 3.11, a single interface is shared between all satellite instances. The interface has an internal library loader responsible for loading a shared object containing the code for each satel-

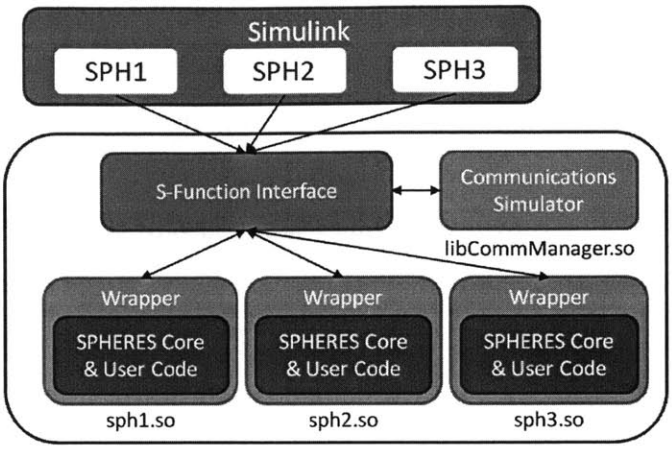


Figure 3.11: There are three layers in the software model. The S-Function Interface communicates with Simulink and coordinates software interactions between satellites, the Wrapper replaces low-level software interfaces with simulation equivalents, and SPHERES Core & User Code is flight-compatible software. The communications simulator is loaded separately from the satellites and models the flow of communication packets.

lite. It also loads a separate communications simulator module to model the ground station transmitter and flow of packets between satellites. More details about the dynamic loader are discussed in Section 3.3.6.

SPHERES Core Wrapper Emulates the hardware and low-level software interfaces expected by SPHERES Core. This layer contains memory regions to model the SPHERES flash memory and FPGA registers as well as simulated replacements for basic features of the SPHERES operating system. The wrapper has convenient access to all SPHERES Core and GSP functions for setting and retrieving internal values.

SPHERES Core and User Code Contains flight-compatible software for use in simulation and on satellite hardware.

3.3.5.1 S-Function Interface

Each of the satellites are configured with a set of parameters that are static for the duration of the simulation. They are used by the S-Function interface and lower layers to model components of the satellite that are not part of SPHERES Core but

are accessed by the software. For a detailed example of the interface for the SPHERES Simulation, see Appendix A.2.

3.3.5.2 SPHERES Core Wrapper and DSP/BIOS Model

Since it is not possible to use the proprietary, platform-specific DSP/BIOS execution environment in the simulation, several commonly used components are mapped to constructs in the C++ library Boost[18]. Boost is a well-established, cross-platform utility library and is also serving as a reference for many of the components in the C++11 standard. An important advantage of Boost is the ability to compile the same code on multiple operating systems with minimal (if any) platform-specific customizations. This feature has been critical for Zero Robotics to enable easy development on Windows and Linux systems, and ultimately it will be useful for potential releases of the simulation as a downloadable library.

The following section describes DSP/BIOS components used by SPHERES and their counterparts in the simulation:

Semaphores In SPHERES Core, Semaphores serve the dual purpose of mutually exclusive locks and condition variables for event notification. A calling task can *SEM_post* to a Semaphore to atomically increment its counting variable, *SEM_pend* until the count is greater than 0, check the current value with *SEM_count*, or *SEM_reset* to set the semaphore to a specific value. Since Boost does not have a semaphore construct, a custom Semaphore class combines a condition variable, a mutex, and a counting variable to replicate the behavior.

Mailboxes Mailboxes are concurrent queues containing fixed-length messages. Like semaphores they have *pend* and *post* methods, but the *post* operation adds a message to the queue while *pend* waits for a new message. The simulation implements Mailbox by extending Semaphore, using the internal mutex to protect a queue of messages. Just as in DSP/BIOS, the queue has a user-specified maximum length.

Flash Memory and FPGA Registers In the SPHERES Core header files, the

memory addresses for flash storage and FPGA registers are redirected to regions allocated by instances of the main satellite class. When components inside of SPHERES Core write to the addresses, the data can be easily read and acted upon following completion of the routine. The main challenge is performing actions that normally occur on writes such as sending serial data or triggering an IR pulse. For IR pulses, the value of the register is read after the SPHERES Core update tick completes. Serial port and communications data are intercepted with simulation-specific functions before reaching

Tasks DSP/BIOS tasks are concurrently executing threads dispatched by an internal scheduler. Typically in SPHERES Core, tasks run at the lowest level of execution priority and have the most computationally intensive tasks. Tasks are replaced in the simulation with Boost threads started when the satellite library loads at the beginning of the simulation. Careful attention must be paid to synchronizing the threads with the main simulation thread. See Section 3.3.5.3.

Software Interrupts Typically posted by SPHERES Core in response to a hardware-based event, these interrupts run at slightly higher priority than Tasks, but they may be delayed by higher priority hardware interrupts. In the simulation, software interrupts are modeled with direct function calls from the main simulation thread instead of a separate scheduled thread of execution. In general this is a reasonable model because most software interrupts are dispatched within microseconds of being posted and are expected to complete within a 1 ms time step. This assumption is not always valid, such as when the user control interrupt, SWI_Controller(), involves significant computation time.

Hardware Interrupts As with software interrupts, hardware interrupts are triggered with direct function calls from the main simulation thread. Most hardware interrupts run on each 1 ms time step of the simulation. This introduces a very slight inaccuracy because the thruster timing interrupt and TDMA communications manager interrupts are driven by a hardware timer with a period of 1.0078 ms [63]. The only interrupt not driven by a simulation tick is the IR re-

ceive interrupt, which is triggered for all satellites at the end of SPHERES Core execution if any single satellite sets the transmit flag in the FPGA memory.

For most of the components, a set of C++ gateway functions function with the same call interfaces replace the DSP/BIOS functions. Objects like Semaphores are referred to by a handle, which in the simulation implementation is simply a pointer to an array index that can be used to look up the requested object in a global instance of a class representing a SPHERE. For example, for a call to the *SEM_pend()* function the following steps take place:

1. SPHERES Core calls the function *SEM_pend(handle, timeout)*. In the simulation this function is implemented in C++.

2. The C++ *SEM_pend()* function dereferences the pointer *handle*, which returns an array index.

3. *SEM_pend()* looks up the Semaphore instance at the specified index in the global satellite instance and calls the *pend()* member function.

3.3.5.3 Thread Synchronization

On modern multi-core computing hardware, the concurrent threads of the simulation have the potential of running at very different relative rates compared to their behavior on the SPHERES hardware. When running faster than real time, or in any situation where the threads are not monitored by a scheduler, it is important to ensure the main thread does not run many steps ahead in the simulation while a parallel thread is making more laborious computations. While it is not practical to model the exact relative timing, it is possible to use a coarse model of execution to make sure the threads stay in sync.

Background tasks in SPHERES Core follow a consistent pattern with an infinite outer loop broken by a pause point to wait for new data, typically a *SEM_pend()* or *MBX_pend()*. Assuming this structure, each iteration through the loop can be synchronized with the main simulation thread with the following steps. Figure 3.12 illustrates the steps with satellite's state estimation thread as an example.

1. The process begins when SPHERES Core or GSP flight code initiate a call to one of the DSP/BIOS synchronization constructs. In the example, new IMU or global metrology data in the PADS hardware interrupt triggers an *MBX_post* to the estimator mailbox.

2. In the SPHERES Core Wrapper, the simulated implementation of the DSP/BIOS function checks incoming events to see if they are bound for one of the threads requiring synchronization. Before posting the signal or message to wake up the target thread, the simulation records a future synchronization time based on the expected total computation time, `delta`. The synchronization time is stored in a map based on the target thread's unique identifier.

3. The main simulation thread continues normal execution, potentially completing multiple time steps before reaching the synchronization time. At the end of each time step, the *waitForSync()* operation checks to see if there are any threads due to complete at the current simulation time. If a thread has not completed but is due to do so at the current time step, the main thread blocks for a signal.

4. The target thread wakes up in response to the message and begins executing in parallel with the main simulation thread.

5. At the end of the execution block, the thread calls a special simulation macro *SIM_SYNC_RELEASE()*. Using the calling thread's unique identifier, this action looks up the thread in the synchronization map, resets the synchronization time to infinity, and signals the waiting main thread to wake up.

The approach works well to keep the simulation synchronized and could be generalized for other simulations of hardware with multiple threads or asynchronous events. There are also several important limitations:

- The parallel task must provide some form of signal that can be intercepted by the main simulation thread to signal synchronization. In the SPHERES simulation, this requires modifying flight code to add synchronization annotations. When

compiled for hardware, the *SIM_SYNC_RELEASE()* macro is automatically redefined to empty code.

- An accessible pause point must be present to initialize the synchronization process. So far the required synchronization points in the SPHERES simulation have always provided a natural pause point, but future applications may require additional annotation macros to be added to the flight code, such as a *SIM_SYNC_START()* at the beginning of an execution block.

- Between the pause point and synchronization point there is no guarantee of relative execution timing. If a parallel task is given several simulation steps to complete its execution, the relative execution speed between the two tasks will almost certainly not match the relative timing on the hardware.

- Profiling on real hardware is required to set accurate computation time.

- The synchronization approach is "optimistic." The parallel thread begins computation as soon as the message or signal arrives and may complete before the synchronization point is reached. Any simulated time information the thread accesses from the simulation may be earlier than on the real hardware. An alternative "pessimistic" approach would be to wait until the main thread reaches the synchronization point, execute the parallel thread, then resume the main thread. This would result in slower, mostly single-threaded execution, but the timing would always model the worst case execution time.

For all of the points, a detailed knowledge of the interaction between the threads is important to correctly implementing their simulated behavior.

3.3.6 Dynamic Loader for Satellite Libraries

Following the principle of *Efficient Inquiry* it has been critical to reduce the time required for the sequence of compiling, running, and evaluating simulations. Since user code represents only a small part of the overall simulation source code, it is excessive

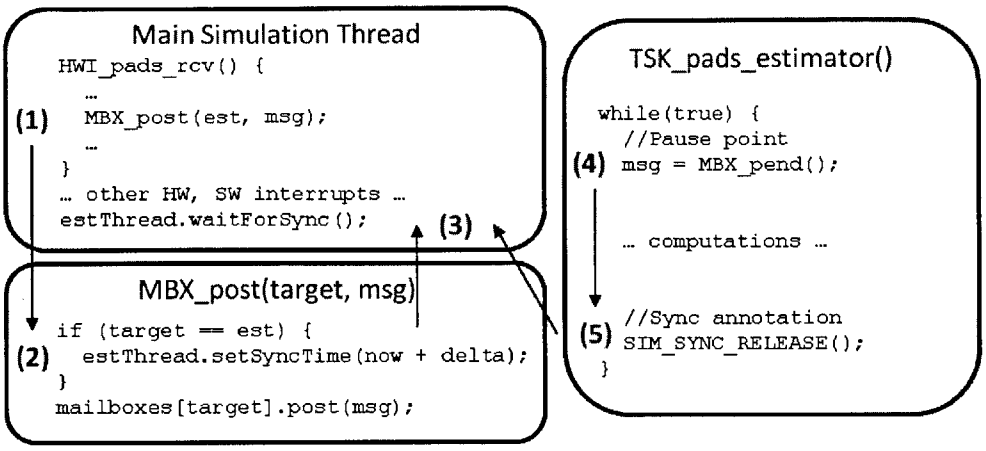


Figure 3.12: (1) The thread synchronization process starts with a signal or data from the main thread. (2) The simulated synchronization construct recognizes the target and sets an expected finish time based on the current simulation time. (3) The main simulation thread executes until the synchronization time is reached, then blocks if the other thread is still running. (4) The target thread wakes up with new data, performs, computations, then (5) releases the waiting main thread.

to recompile the entire simulation for each code update. In addition, since the simulation dynamic model changes much less frequently than satellite software or user code, it is desirable to independently deploy the simulation from the satellite code. The SPHERES simulation separates the simulation environment from the satellite software by dynamically loading the satellite code as shared libraries at runtime.

The loading process is similar to traditional systems for loading shared library plugins:

1. For each satellite in the simulation, a special library loader in the SPHERES Software component of the simulation is passed a full path to a user library.

2. The loader makes an operating-specific call to load the library into memory (`dlopen` for *nix and Mac, `LoadLibrary` on Windows).

3. The loader looks up a special function in the satellite library that can be used to retrieve a pointer to the SPHERES Core wrapper. The pointer is stored in a list containing entries for each satellite.

4. Using the SPHERES Core wrapper as an interface, the simulation can exchange

data with the satellite or call methods to advance the simulation state

The loader can also look up global variable names and write directly to their memory locations with new values. This ability is used extensively in Zero Robotics for variables that change from run to run such as randomly placed game elements. Since variables compiled in the code can be referenced by name, the game code does not require an additional simulation-specific interface for modifying variables, and most importantly, it can be pre-compiled for all simulation runs, further improving turnaround time.

3.3.7 Code Generation Capability

A major reason for returning to a Simulink simulation implementation is the ability to generate a fully C++-based version of the block diagram with identical outputs to the interpreted block diagram. The compiled C++ simulation runs significantly faster (2-3x) than the block diagram version and can be transferred as source code and compiled into standalone executables on other computers and operating systems. Simulink provides special utilities for creating interfaces between the C++ code running in the S-Function interface and the Simulink simulation so that the interface works in both the generated simulation and under the Simulink environment. This is particularly useful for initial simulation verification because the model can be easily modified in the block diagram format then converted to C++ for distribution with no additional modifications.

3.3.8 SPHERES Code Profiler

With the significant difference in computational power between modern x86 personal computers and the SPHERES DSP it is often possible to implement a program that runs very fast in simulation but would be infeasible to execute on the hardware platform. To prevent this situation from occurring, the Zero Robotics platform includes a code profiling tool based on a cycle accurate simulator of the C6701 DSP. As noted above, the simulator is too slow to run use in a complete dynamic simulation of

SPHERES, but it is possible to limit the scope to running a single iteration of the user's control loop. The code is compiled into an image for the DSP and launched in the simulator for a single cycle. The user has access to timing routines that print the total time elapsed at any point in the code, and the tool reports the overall execution time with warnings if the predicted time could result in problems. The approach does require the cooperation of the users in configuring their code to identify the worst case execution scenarios. Based on the 2012 RetroSPHERES tournament, the first to raise the issue of computational complexity, users were able to effectively locate and modify problematic code. All teams that proceeded to the ISS phase were able to reduce execution times to within acceptable limits within a week of using the profiling tool.

3.4 Zero Robotics API

The Zero Robotics API extends the GSP API with an additional layer of software interfaces for simplifying the control of 6DOF satellites for student users and for easily implementing Zero Robotics games.

3.4.1 History

The first Zero Robotics API was specifically configured for the pilot game. As described in Section 2.4.1, the primary action in the game was to steer the *helper* or *blocker* satellite to a position within the volume based on the motion of the opponent. The relatively simple game could be played with a single interface API function:

```
void setTarget(float *myState, float *otherState, float time, float *targetOut);
```

To play the game, competitors implemented their code in the body of the *setTarget* function. The incoming arrays *myState* and *otherState* were the first attempts at providing a simplified state representation for the high school students. The arrays contained position, velocity, and a single attitude angle, representing the rotation

around the satellite's z axis,

$$myState, otherState = \begin{bmatrix} x & y & z & v_x & v_y & v_z & \theta_z & \omega_z \end{bmatrix}^T. \tag{3.24}$$

For the implementation of time-triggered maneuvers, the *time* variable contained the time in seconds since the beginning of the match. The last argument, *targetOut,* provided a length 4 array,

$$targetOut = \begin{bmatrix} x & y & z & \theta_z \end{bmatrix}^T \tag{3.25}$$

for the user to command position and attitude targets. Following execution of the user code, targets were passed to a standard SPHERES PD controller for position and attitude. In this season alone, a PID controller was used when errors dropped below 10 cm or below 35^0. This behavior was later changed to consistently using a PD controller for position and a PID controller for attitude for more predictable responses. Only two utility functions were available, tailored to the game-specific objectives of reaching the target while minimizing fuel consumption. No additional utilities were available except for standard ANSI C math functions.

Development for the next Zero Robotics events, the 2010 Summer of Innovation Tournament (SOI) targeted at Middle School students, and the 2010 High School Tournament: HelioSPHERES started in parallel. Based on lessons from the 2009 pilot and the need to develop two games at once, the API began taking on a more generic format. Two entry-point functions `ZRInit()` and `ZRUser()` were created to separately initialize and update user code, and the user control options became more expressive with the ability to command forces and torques. For the first time users were also supplied with control over 3D satellite attitude using the representation covered in Section 3.4.3.3. The 2010 API persisted into the 2011 season where it was used in the AsteroSPHERES tournament. During this tournament a more formal internal template was developed for creating new games.

With the higher complexity control algorithms of the 2012 ZRASCC tournament, several new features were added to give users full control over the satellite from po-

sition control down to forces and torques as well as controlling the PD and PID gains of the internal controllers. For the first time a mechanism was added to the architecture for implementing the full range of SPHERES Guest Scientist Program functions, though it has not been used in a tournament to date. This enhanced interface has been maintained for future expansions of the platform to general SPHERES programming.

The most recent version transitioned the code base for the API from C to C++ and set up an object-oriented architecture for the user and internal code development. Though creative use of C++ is somewhat limited by code size and execution performance on the satellite hardware, the new API lifts restrictions from the user code and eases the process of creating new games.

3.4.2 Software Architecture

After several iterations, the Zero Robotics API has matured into a flexible library with which many interesting games can be developed. Though every additional season will hopefully improve the functionality, the core components are stable and generic enough to be reused in most games. There are two main parts to the software architecture:

1. A standard set of control commands, controller implementations, and state representations available in all games. To the users, this component is referred to as the *ZR API.*

2. A changing set of functions and rules implemented by the game designer for each tournament. To the users this is referred to as the *Game API.*

Both components of the architecture are split into a user-facing API, accessed through a C++ object, and an internal implementation to process the actions triggered by the users. An important challenge with this configuration is creating a game implementation where internal code remains hidden from access to the users while still available for internal processing. For example, consider a game like HelioSPHERES

where the users consume a limited virtual resource like "charge," represented by a private counting variable in the game object. The game also internally replenishes the resource under certain conditions, such as facing toward the sun. If the variable is private to the game object, the internal game rules can only update the resource through a public mutator method or by residing in the object itself. A public method will not work because the user could easily call the method to replenish the resource outside of the game rules, and implementing the game rules inside the object is not practical because at some point information must be passed into the object through a public interface.

To solve the problem, the Zero Robotics API uses a programming idiom called *Pimpl* for *Pointer to Implementation* [67]. The user-facing API is a class containing public methods and fields intended for the user along with a private pointer to an implementation class (*pimpl*). The implementation class contains the game logic along with public fields for all of the internal game variables. When the user calls an API member function, it can access the implementation fields through the private pointer, but the user cannot modify them directly. Inside the game implementation, all the fields are public and easily accessible for update.

In addition to the game-oriented features, the architecture has been designed with an eye toward future expansion of Zero Robotics into a platform for programming all features of the SPHERES Guest Scientist Program. Both the game and the user code are implemented in classes that extend from a generic interface called `GSPBase`. This class supplies empty implementations for all of the standard GSP callback functions, allowing future iterations of Zero Robotics to have either optional or required implementations of these functions.

Normally, allowing a set of optionally implemented functions is performed through the use of virtual functions in C++. One commonly used feature of virtual functions is *dynamic polymorphism*, where a specific class implementation is bound to a pointer to its base class. Calling methods on the base pointer will refer to the appropriate implementation in the derived class. Unfortunately, virtual functions also require significant code space overhead because a hidden internal function table must be

```
1  class ZeroRoboticsGame {
2  public:
3      //The user can access this method
4      void useCharge() {
5          pimpl->charge--;
6      }
7  private:
8      //but not the pointer to the implementation
9      ZeroRoboticsGameImpl *pimpl;
10 }
11
12 class ZeroRoboticsGameImpl {
13 public:
14     //This field, and the addCharge method are
15     //only available to the game implementation
16     unsigned int charge;
17     void addCharge() {
18         charge++;
19     }
20 }
```

Listing 3.1: In this example of the Pimpl idiom, the user-facing game API class ZeroRoboticsGame exposes the method `useCharge()`, which in turn accesses the hidden `charge` field. The user cannot modify the implementation field directly.

generated for the dynamic binding of functions. With the extreme space limitations of SPHERES, virtual functions are only used in Zero Robotics to dynamically assign which team's code is running on the satellite via a pointer to the base class `ZRUser`. For other use cases an alternate method is available.

As with `GSPBase`, virtual functions can also be used to enforce an interface contract, where all classes that derive from the base are either required to implement a method (also known as *abstract* methods), or the base provides an optional default implementation for the method. To achieve a standard interface with minimal space overhead, `GSPBase` avoids the use of a virtual function with a Curiously Recurring Template Pattern (CRTP)[13]. As shown in Listing 3.2, in CRTP, derived classes inherit from a C++ template base class while supplying themselves as a template argument, hence the *recurring* part of the name. In the template base class, a static cast binds the methods to a derived implementation at compile-time instead of through a virtual function table. The code usage savings comes at the cost of dynamic polymorphism. Classes deriving from the CRTP base cannot be referred to with a base pointer.

3.4.3 User-Facing API Design

3.4.3.1 User Code Template

While many other robotics programs give students free access to writing source code at the level of files, with the general requirement of maintaining a flight-like code configuration, it is important to impose several restrictions on the way users implement their programs. One of the most onerous requirements is maintaining a non-conflicting set of variable and function symbols across all the user programs that share the same program memory space on the satellite DSP. The most recent solution is to insert the user code into a C++ class body, as shown in Listing 3.3. With this configuration, the users are free to declare methods and fields with any name because they will be constrained to the scope of the class.

Before the user's class the template declares two variables:

```
1  template<typename T> class GSPBase {
2      //A method that must be defined in the derived class (abstract)
3      void gspInitTest(unsigned int test_number) {
4          static_cast<T*>(this)->initTest(test_number);
5      }
6
7      //A method with a default implementation
8      void gspTaskRun(unsigned int gsp_task_trigger, unsigned int extra_data) {
9          static_cast<T*>(this)->taskRun(gsp_task_trigger, extra_data);
10     }
11     void taskRun(unsigned int gsp_task_trigger, unsigned int extra_data) {
12         //...Default implementation...
13     }
14 };
15
16 class Derived : public GSPBase<Derived> {
17     //Implementation of required method
18     void initTest(unsigned int test_number) {
19         //...
20     }
21     //Override of default implementation
22     void taskRun(unsigned int gsp_task_trigger, unsigned int extra_data) {
23         //...
24     }
25 };
```

Listing 3.2: In the Curiously Repeating Template Pattern (CRTP), a derived class inherits from a base template with itself as an argument. Methods in the base class perform a static cast to the derived type to achieve static (compile-time) polymorphism. Methods without a default implementation like `gspInitTest()` must be implemented in the derived class or compilation will fail.

```
1  #include <math.h>
2  //... Additional Includes ...
3  #include "ZRUser.h"
4
5  //Global references to game and ZR API
6  ZeroRoboticsGame &game = ZeroRoboticsGame::instance();
7  ZeroRoboticsAPI &api = ZeroRoboticsAPI::instance();
8
9  class ZRUser01 : public ZRUser {
10 public:
11
12     ${codeBody}
13
14 };
15
16 ZRUser *zruser01 = new ZRUser01;
```

Listing 3.3: User code is inserted into a template similar to the one above. The template token `${codeBody}` is replaced with the user code. Placing code into a class body gives users access to object-oriented features while preventing method name collisions.

`game` An imported global instance of `ZeroRoboticsGame`, the user-facing game API class.

`api` A shortcut reference to the instance of `ZeroRoboticsAPI`, the user-facing ZR API class.

These objects serve as the access point to the user API functions.

3.4.3.2 Entry Point Functions

As with the GSP API, users begin their programs by implementing entry point callback functions. All user classes extend from the parent class `ZRUser`, which defines two abstract methods, `init()` and `loop()`, required in all implementations. The names of the functions were selected with a similar intent to the Arduino API [2], or the Processing Language [59] which contain the methods `setup()` and `loop()` to indicate the initialization and repeating phases. The API similarity gives students a starting point to transition to or from the other platforms.

The `init()` method is called at the beginning of user code execution with the intended purpose of resetting internal variables before entering into the game. Since the user class is only constructed when the satellite first turns on, running the `init()` method is essential to resetting internal variables on repeated test runs. Failure to initialize the variables is difficult to detect in software, but one approach is covered in Section 3.4.5.3.

The `loop()` method is the main control loop for the user's program. It is triggered once per second by the game, though where in relation to the rest of the game rules is specific to the game implementation. From `loop()` the user may access API functions or call other methods in the program.

3.4.3.3 State Representation

By querying the satellite state, users can build programs that dynamically react to the position of their satellite and make choices based on the observed behaviors of their opponents. The ZR API includes two different representations of the SPHERES state vector to accommodate different skill levels. The first advanced version is a standard SPHERES state vector, accessed through `api.getMySphState()`,

$$state\_vector = \begin{bmatrix} \mathbf{r} & \mathbf{v} & \mathbf{q} & \omega \end{bmatrix}^T. \tag{3.26}$$

where $\mathbf{r}$ is the position of the satellite, $\mathbf{v}$ the velocity, $\mathbf{q}$ the quaternion attitude, and ω the body-frame angular rates. The quaternion representation follows the SPHERES standard with the fourth component representing the scalar part. Since the student programmers are not all expected to have familiarity with quaternions, the Zero Robotics standard state vector has a simplified attitude representation, accessed through `api.getMyZRState()`,

$$ZRState = \begin{bmatrix} \mathbf{r} & \mathbf{v} & \hat{\mathbf{n}} & \omega \end{bmatrix}^T. \tag{3.27}$$

The $\mathbf{r}$, $\mathbf{v}$, and ω components are identical to state_vector, but $\hat{\mathbf{n}} = \begin{bmatrix} n_x & n_y & n_z \end{bmatrix}^T$ is a unit vector that points in the direction where the user would like the satellite to point. Unlike other 3 parameter attitude formats such as Rodrigues Parameters or Euler Angles, this representation is well suited for high school students because the user can simply indicate a pointing direction without complicated transformations. However, like all 3 parameter attitude representations, the system does not fully define the attitude of the satellite. The pointing direction is defined by aligning a reference vector, $\hat{\mathbf{n}}_{ref}$, in the body frame of the satellite[4] with the desired pointing direction, but the satellite has complete freedom to rotate about this direction while maintaining the same pointing vector. In some games, such as HelioSPHERES and AsteroSPHERES, game induced torques about the pointing direction were used to visually indicate an event in the game. When not used for these purposes, the internal SPHERES controller that maintains the attitude will naturally damp residual angular velocities around the pointing direction, though small perturbations may cause the satellite to gradually rotate to different positions during a match.

Converting from the quaternion representation to the pointing vector representation is straightforward.

$$\hat{\mathbf{n}} = \mathbf{R}(\mathbf{q})\hat{\mathbf{n}}_{ref} \tag{3.28}$$

As usual, $\mathbf{R}(\mathbf{q})$ is a rotation matrix from the body frame to the global frame based on the quaternion $\mathbf{q}$. The pointing vector is calculated by rotating the reference vector from the body frame to the global frame.

Converting from a pointing vector to a quaternion is more complicated because there is an unconstrained degree of freedom and therefore an infinite number of quaternions for any pointing direction. To remove the ambiguity, the conversion algorithm takes three arguments:

[4]The first game to use the simplified attitude representation was the 2010 HelioSPHERES tournament. For this game, the pointing face of the satellite was defined to be the -X face where the satellites have several patches of Velcro for docking. This face has been used ever since for the attitude representation. Future games might consider switching to the +X face for consistency with the SPHERES coordinate frame and any games that may use expansion items on the +X side of the satellite.

refVec The body-frame reference vector, $\hat{\mathbf{n}}_{ref}$

attVec The global frame attitude vector, $\hat{\mathbf{n}}$, to convert to a quaternion.

baseQuat A quaternion representing the initial orientation of the satellite when $\hat{\mathbf{n}}_{ref}$ is aligned with the global frame.

The last argument, baseQuat, removes the rotation ambiguity because it sets a defined initial orientation for the reference vector. The conversion algorithm first creates a quaternion corresponding to the rotation between reference vector and the target vector about an axis perpendicular to both of them, then multiplies the base quaternion by the result.

$$\begin{aligned} \hat{\mathbf{e}} &= \hat{\mathbf{n}}_{ref} \times \hat{\mathbf{n}} \\ d &= \hat{\mathbf{n}}_{ref} \cdot \hat{\mathbf{n}} \\ \theta &= \text{atan2}\left(\|\hat{\mathbf{e}}\|,\, d\right) \\ \mathbf{q}_{rot} &= \begin{bmatrix} \mathbf{e} \sin\frac{\theta}{2} \\ \cos\frac{\theta}{2} \end{bmatrix} \\ \mathbf{q} &= \mathbf{q}_{rot} \otimes \mathbf{q}_{base} \end{aligned} \tag{3.29}$$

The most important consideration when converting the pointing vector to a quaternion for attitude control is making sure small angles between successive pointing vectors are preserved as small relative rotations. The Zero Robotics API achieves this by using the current pointing vector as the reference vector, and the attitude quaternion as the base quaternion whenever the user commands a new pointing direction. Stepping through the conversion process, this results in applying a rotation quaternion corresponding to the minimal rotation between the current attitude and the target attitude.

In addition to retrieving the current satellite state, the user may also query the state of their opponent's satellite with `api.getOtherSphState()` and `api.getOtherZRState()`. A configuration option within the ZR API allows game implementation to modify or disable access to either of the state vectors if some obfuscation of the state is required

within the rules of the game.

3.4.3.4 Control Options

Commands to change the satellite position and orientation are the fundamental building blocks of all Zero Robotics programs. To match the broad range of skill levels using these commands, the control interface to SPHERES exposed through the ZR API is designed to cover several levels of complexity.

The most basic type of control command is a position or orientation command. Users can call `api.setPositionTarget()` or `api.setAttitudeTarget()` to direct the satellite to a specific location in the test volume or point the satellite in a provided direction. The attitude commands use the normal vector attitude representation from Equation 3.27, but advanced users may also use `api.setQuatTarget()` to command a quaternion attitude. All of the commands are routed to internal closed loop PD and PID controllers as described in Section 3.4.4.3.

The second level of closed loop control is for commanding velocity. The methods `api.setVelocityTarget()` and `api.setAttRateTarget()` are the velocity equivalents of the position commands. The velocity controllers can be layered together with position control for the purpose of trajectory tracking instead of basic point-to-point targets.

The finest grained layer available to Zero Robotics programmers applies open loop force and torque commands to the satellites. The commands `api.setForces()` and `api.setTorques()` will set global fame forces and body-frame torques respectively. In general competitors are discouraged from using open loop commands because thruster strength may vary significantly between the simulation and hardware. The commands are occasionally useful when adding feedforward actuation for trajectory tracking. In addition, advanced users may use the force and torque interface to implement their own closed loop control algorithms.

Starting with the ZRASCC tournament, several additional advanced features were added to make the Zero Robotics API suitable for algorithm development while preserving the basic control interfaces. The following utilities can be used to modify the

behavior of the internal control algorithms:

`setControlMode(posCtrl, attCtrl)` Defines which internal controller will be used for position and attitude control. Can be set to CTRL_PD or CTRL_PID to select PD or PID control.

`setPosGains(P, I, D)` Modifies the gains of the internal position PID controller. I gains are ignored if the controller is configured in PD mode. As with force/-torque commands, modifying the well-tested gains of the SPHERES control system is discouraged for ISS tests, but teams have full access to these parameters for advanced usage.

`setAttGains(P, I, D)` Modifies the gains of the internal attitude PID controller. I gains are ignored if the controller is configured in PD mode.

`setCtrlMeasurement(sphState)` Uses the specified SPHERES state vector as the incoming measurement when calculating error: $e = \mathbf{x}_{target} - \hat{\mathbf{x}}$. This command is useful when performing control relative to another moving object.

`spheresToZR(sphState, zrState)` Converts a SPHERES state vector (13 elements) to a ZR state vector (12 elements) with simplified attitude representation.

`attVec2Quat(refVec, attVec, baseQuat, quatOut)` Converts a 3 parameter attitude vector attVec to a unit quaternion. The refVec specifies a vector in the body frame of the satellite that should be used as the pointing direction (such as $\begin{bmatrix} -1 & 0 & 0 \end{bmatrix}^T$ for the -X axis). The baseQuat input defines an additional rotation to the satellite that should be applied to the global reference frame before calculating the output quaternion. This function is normally used internally within the API to translate a user target to a standard quaternion attitude.

`quat2AttVec(refVec, quat attVec)` Converts a unit quaternion to a Zero Robotics pointing vector, using refVec to define the pointing direction corresponding to no rotation.

With this library of functions, most control approaches that have been applied to SPHERES can be implemented. The only levels of control that remain hidden are the translation of forces and torques into thruster on/off firing times (usually known as a "mixer"), and the activation of the thruster firing times. Since this functionality involves more detailed knowledge of SPHERES Core, and game rules may perform additional modifications of the user commands, the final thruster firing commands are reserved for the game implementation.

3.4.3.5 Math Libraries

To perform mathematical computations, users have access to most of the functions in the standard ANSI C standard floating point math library. To minimize code space, users are encouraged to use the single precision floating point versions of the math libraries. Double precision libraries have a significant overhead because the SPHERES DSP does not natively support double precision operations and must add significant wrapper code to double precision calls to do so. All double precision calls have been carefully removed from the Zero Robotics API and SPHERES Core libraries to conserve space for this reason.

Users also have access to the SPHERES matrix math library for performing matrix and vector computations. Basic multiplication, addition, vector normalization, and vector product operations are available. The matrix math library also includes several quaternion operations, including quaternion multiplication and conversion to rotation matrices.

3.4.3.6 Debugging

Zero Robotics users have two forms of runtime debugging options that are only available in simulation. At any point in the program, the macro DEBUG() may be invoked to print text. The text is captured by the simulation engine and displayed in the 3D visualization for playback. The macro and usage are defined as follows:

```
#ifdef ZRSIMULATION
    #define DEBUG(arg) debugPrintf arg
```

```
#else
    #define DEBUG(arg)
#endif

//Typical usage
int d = 3;
DEBUG(("Hello World, with format: %d!", d));
```

Though it comes with a slightly awkward syntax compared to a typical `printf` statement, the macro serves two purposes. First, it can be easily removed from the code for hardware tests by defining the macro to empty text, as indicated by the second definition in the code above. Second, it re-routes arguments to a custom implementation of the `printf` function, `debugPrintf`, which captures the printed text for storage in the simulation telemetry. The text can be viewed later in the 3D simulation visualization tool covered in Section 3.7.

Users also have access to a second debugging option. By calling the function api.setDebug, the user may supply an array of 7 floating point numbers that will be appended to the satellite telemetry. After a simulation run, the values can be plotted in the web-based report tool.

3.4.4 Internal API Design

After many iterations of Zero Robotics game development, a standard sequence of maneuvers has been formalized to initialize, run, and terminate the game. An overview of the standard maneuvers is discussed with game design in Section A.1.3, but this section will elaborate on several important implementation details.

3.4.4.1 Game Base Implementation

Like the user-facing API, the internal API is split into standard ZR API components and game-specific code implementations. The API also includes the class `ZeroRoboticsGameBaseImpl`, the base class for all game-specific code. As shown in Listing 3.4, the class is a template using the CRTP pattern covered in Section 3.4.2. CRTP is used to require all base classes implement the game functions, `init()`,

`sendDebug()`, and `update()`, while supplying a default implementation for the required `GSPBase` functions `initTest()` and `control()`.

The two GSP functions are the bridge between the SPHERES Guest Scientist Program and the Zero Robotics API layer. As with all SPHERES tests, the `initTest()` portion clears internal variables for the test run and initializes the SPHERES standard estimator. It also triggers the `init()` method of the derived game implementation, which has the same purpose for the game-specifc code. The 1 second loop of `control()`, includes four distinct GSP maneuvers:

1. Estimator convergence and opponent selection
2. Initial positioning
3. Game execution
4. Termination

The first phase allows the ultrasound metrology system to converge and gives the crew a chance to select an opponent. The second phase moves the satellites from their deployment positions to their starting positions. In the online simulation environment opponent selection happens immediately, and the initial positioning period is skipped because the satellites are initialized at their starting locations. As the program transfers from maneuver 3 to maneuver 4, the user's own `init()` function is called. It is important to trigger the user's code at this point and not at test initialization because the code may make use of the current satellite state.

The third phase drives the game and user updates. The following actions happen in sequence:

1. The API takes a measurement of the current satellite clock time.
2. The `update()` performs a game-specific update in the derived class:
   (a) Calls any game rule updates prior to triggering user code.
   (b) Activates the user code by calling the `loop()` method.

(c) Determines force and torque values to apply based on user and game code

(d) Returns a boolean value indicating if the force and torque commands should be actuated as thruster commands. If no firing commands are to be applied during the current cycle, the method returns false.

3. A second measurement of the satellite clock time is taken and the measurement from the first step is subtracted to determine the total elapsed computation time for the game and user code.

4. Force and torque commands are mixed into thruster times by the SPHERES mixer and scheduled to activate. The period available for actuation is shortened by the elapsed computation time. This prevents the code from accidentally attempting to fire thrusters during an estimation period if the student code runs too long.

5. The game's `sendDebug()` method is triggered at the very end of the code to send telemetry updates. Telemetry is used to synchronize game information and animate the visualization.

During all maneuvers, state of health telemetry packets from the opponent satellite are checked for an early termination of the test due to an error or reset. The program ends immediately for either of these conditions to prevent wasted time in the event of a problem.

All of the functionality is inherited by extending form ZeroRoboticsGameBaseImpl, so the game designer can focus specifically on how to structure the game rules and interface functions.

3.4.4.2 Opponent Selection

As described above, each user's program extends from the base class `ZRUser`, and each implementation file exports a global `ZRUser` pointer to a local instance of the user class. The pointers are collected in a table in an order determined by the bracket

```
1  template <typename T>
2  class ZeroRoboticsGameBaseImpl :  public GSPBase<T>{
3  public:
4      // Instance of the ZR API implementation
5      ZeroRoboticsAPIImpl apiImpl;
6
7      // Total runtime of user and update code
8      unsigned int compute_time_;
9
10     // Game initialization function
11     void init(){
12         static_cast<T*>(this)->init();
13     }
14
15     // Rules update function to be called during control()
16     bool update(float forceTorqueOut[6]){
17         return static_cast<T*>(this)->update(forceTorqueOut);
18     }
19
20     // Called at the end of control() to send debug packets
21     void sendDebug(){
22         static_cast<T*>(this)->sendDebug();
23     }
24
25     void initTest() {
26         //... Standard ZR test initialization implementation...
27     }
28
29     void control() {
30         //... Standard ZR 1 Hz control loop implementation...
31     }
32 }
```

Listing 3.4: The `ZeroRoboticsGameBaseImpl` class is the base class for all Zero Robotics games. The CRTP pattern requires methods `init()`, `update()` and `sendDebug()` to be implemented by the game, while the `GSPBase` methods `initTest()` and `control()` implement a standard test initialization and control loop for all games.

structure of the competition. Accesor methods can retrieve any of the pointers based on a 0-based team identifier.

At the start of a match, the crew member operator selects the first player through the traditional test number interface. Inside the ZR API, the test number is translated into a team identifier and an internal `ZRUser` pointer is bound to the associated implementation on the primary satellite. In the first 10 seconds of the match, the crew is also instructed to press a number on the keyboard to identify the second player. The secondary satellite receives this command and binds the `ZRUser` pointer to the user implementation associated with the number. In this way, any two teams can be configured to compete on either of the two satellites involved in a test.

3.4.4.3 Position and Attitude Control

The internal Zero Robotics API contains the method `getForceTorque()` to automate the calculations of forces and torques from the controller commands issued by the user. Algorithm 3.4 compactly calculates all three levels of control (position, velocity, and force/torque) in a single function through several steps. For the case of controlling position, on Line1, the procedure initializes the target state to the state selected by the user for control, with the current global metrology state as default. At Line 4, any commanded user forces are copied into the force vector. Lines 5-9 check for position and velocity commands. For only position commands, the target velocity is cleared to implement traditional rate feedback servo control. For trajectory control, the user can layer both position and velocity commands together.

The final block starting at Line 12 first computes the error between the targets assigned above and the user-selected control state. Since the target state is initialized to the control state at the beginning of the algorithm, any fields that have not been modified will produce 0 error. This is the key to allowing all three levels of control. Lastly, either PD or PID control forces are calculated based on the user control selection and added to the force vector.

Following calculation of the user control forces, the game implementation may choose to modify the forces based on constraints in the game. Typical applications

Algorithm 3.4 Compact Force and Torque Calculation

1: $\mathbf{x}_{target} \equiv \begin{bmatrix} \mathbf{r} \\ \mathbf{v} \end{bmatrix}_{target} \leftarrow \mathbf{x}_{state}$
2: $\mathbf{f} \leftarrow \mathbf{0}$
3: **if** $userSetForces$ **then**
4: $\quad \mathbf{f} \leftarrow \mathbf{f}_{user}$
5: **if** $userSetPosTarget$ **then**
6: $\quad \mathbf{r}_{target} \leftarrow \mathbf{r}_{user}$
7: $\quad \mathbf{v}_{target} \leftarrow \mathbf{0}$
8: **if** $userSetVelTarget$ **then**
9: $\quad \mathbf{v}_{target} \leftarrow \mathbf{v}_{user}$
10: $\tilde{\mathbf{x}} \leftarrow \mathbf{x}_{target} - \mathbf{x}_{state}$
11: **if** $userSetPosTarget$ **or** $userSetVelTarget$ **then**
12: $\quad$ **if** $mode = PD$ **then**
13: $\quad\quad \mathbf{f} \leftarrow \mathbf{f} + K_P\tilde{\mathbf{r}} + K_D\tilde{\mathbf{v}}$
14: $\quad$ **else if** $mode = PID$ **then**
15: $\quad\quad \mathbf{f} \leftarrow \mathbf{f} + K_P\tilde{\mathbf{r}} + K_I \int \tilde{\mathbf{r}} dt + K_D\tilde{\mathbf{v}}$

include overriding the forces and torques to avoid collisions, preventing the users from moving outside the game boundaries, visually signaling a condition in the game, or disabling a satellite for a penalty.

3.4.5 Catching Common C/C++ Coding Errors

While C/C++ offers an efficient, expressive base for developing programs, several common yet potentially fatal coding errors can go unnoticed in the simulation. The Zero Robotics platform has several experimental safeguards to detect and warn against these problems. Where possible, the platform explicitly warns users that an issue has occurred, but frequently it is only possible to simply crash the simulation and issue a generic warning. While not desirable from a usability standpoint, the experience is similar to real-world debugging for embedded programming. The following sections will describe the current solutions for catching the errors, while the polished versions will remain for future work.

3.4.5.1 Compiler Flags and Banned Keywords

Though most of the challenging problems relate to runtime issues, several basic problems can be detected at compile-time with extra flags supplied to the compiler. Before running any simulation, user code is compiled by both the SPHERES Texas Instruments compiler and the gcc/g++ compiler. The following flags ensure that gcc throws the same errors as the TI compiler and check for simple errors.

-Werror=implicit-function-declaration Checks for functions that are called without a declaration. Some C compilers, including gcc, will provide a default implementation for a function even if it has not been declared. This can be very confusing if a function implementation was missed or slightly misspelled.

-Werror=uninitialized Checks for the use of a local variable before it has been initialized.

-Wall Sets the compiler to print all warnings. This is particularly useful for guiding the users to making the code more efficient by printing information about unused variables or function, as well as any other minor but helpful code warnings.

Prior to compilation the platform also checks for several keywords present in the user source code. The following keywords may not be used:

`static` Static variables are explicitly banned because they can be used to create non-resettable local variables in functions. The Zero Robotics API assumes the user code can be run multiple times by calling the user function `init()` to re-initialize all variables. Any persistent variables that carry over from one test without resetting could result in inconsistent behavior. Broadly banning the static keyword comes at the cost of eliminating static member variables or methods in the user code, but these are not widely used unless the user creates inner utility classes.

`new`, `malloc`, `calloc`, **and** `realloc` SPHERES does not support dynamic memory allocation from within the software interrupt where user code is activated. Re-

moving `new` is problematic because it is common to language that might be used in comments. Better regular expression parsing would help with this issue.

3.4.5.2 Invalid Floating Point Computations

In some situations, it is possible to perform floating point computations that result in Inf or NaN outputs. For example:

- Divide by 0
- Performing an inverse trigonometric operation with an out of range argument, such as $\arccos(-1.1)$
- Taking the square root of a negative number

In addition to causing undefined behavior in the user code, supplying invalid arguments to API functions can result in corrupted internal game variables. A simple strategy for detecting invalid operations is to perform comparison checks for valid arguments since all comparisons involving NaN return false. For example, when checking to make sure the users always supply forces that do not contain NaN, the following check can be computed to alert the user:

```
1 float mag = mathVecMagnitude(forces, 3);
2 if (mag >=0) {
3     //... normal behavior ...
4 } else {
5     DEBUG(("ERROR: invalid forces have been commanded.
          Check for invalid floating point operations"));
6     //... additional actions such as disabling control ...
7 }
```

Although it has not been implemented on the Zero Robotics platform, on Linux operating systems it is also possible to instrument code to throw exceptions for invalid floating point operations. A handler is registered to respond to the SIGFPE signal, which could in turn perform actions to alert the user.

3.4.5.3 Uninitialized C++ Member Variables

Although the uninitialized variable compiler flag will catch local variables that are used before definition, it is not possible to determine at compile time if global variables or class member variables are used before definition. While users are explicitly warned to initialize variables in the `init()` function, the advice is not always followed. In the 2012 High School tournament, the tournament with C++, a third of the teams competing in the ISS tournament did not initialize member variables, requiring modification of the programs before they were flown on ISS.

In simulation the problem is difficult to detect because the user classes are constructed by declaring them as uninitialized global variables. Most compilers will, by default, zero-initialize the data members of global classes, so users mistakenly assume all the class members are automatically set to zero at the start of a test. While it is not clear how to signal a warning message it is possible to emulate the behavior of randomly initializing member variables. Instead of initializing the user classes with global variables, the classes are constructed using the `new` operator. Invoking `new ZRUser01;` without the trailing parentheses constructs the C++ object without zero-initializing the data members, and the current values occupying the memory space where the object is constructed will set the initial values. The code stays flight-compatible because the dynamic memory allocation is performed at the startup of the SPHERES program, not during a software interrupt.

Simply constructing the variable is not enough to produce random initial values. Much of the heap memory available for dynamic allocation on a personal computer may already be zeroed, so even though the object has not reset the memory space, the variables will still be zero. The solution in this case is to override the `new` operator in the `ZRUser` class with a specific implementation to randomize the memory space. Listing shows an example of overriding `new` with memory allocation directed by `malloc`. To match the memory allocation type, the `delete` operator must use the `free()` function. The override section is wrapped in preprocessor definitions because it does not need to be executed in the flight code.

Combined with a warning in the user code template and reminders throughout the season, this approach should give enough warning by the time of the ISS competition to prevent unexpected behavior due to uninitialized member variables. It is also broadly applicable to any learning situation where it is desired to instruct students about the problems associated with variable initialization.

3.4.5.4 Array Access Overflows

One of the most frequent mistakes made by users is the incorrect indexing of arrays. For example, during the 2D phase of a tournament, users may mistakenly provide a length 2 array to the API function `setPositionTarget()` while it expects a length 3 array or they may provide the length 12 `ZRState` to the function `getMySphState()`, which expects an array of length 13. Reading or writing beyond the boundaries of arrays can cause undefined behavior that may not manifest itself until the code is changed slightly ,or in the worst case, until deployed to hardware. Neither C nor C++ have native array bounds checking, so the task of ensuring correct memory access quickly becomes intractable.

Several tools exist that incorporate array access checking. The general purpose debugging tool Valgrind [54] can detect invalid memory allocation or violations of array boundaries for dynamic memory allocated on the heap, but experiments with Zero Robotics show that it does not catch the more frequent error of incorrectly indexing arrays allocated locally on the stack. In addition, the execution time overhead of running a program with Valgrind make it impractical for use in every simulation.

On Linux versions of the gcc compiler, a special library called Mudflap can detect array allocation errors in programs that are explicitly compiled with flags to enable runtime array access errors. Mudflap adds guards to memory allocated on the stack and forcibly crashes the program if a violation is detected. This approach was used during the 2011 season to alert users that their code had memory access problems. There are two major limitations to Mudflap. First, it is not possible to use a pointer to memory that exists outside of the code compiled with the Mudflap flags. Special simulation-specific memory copy routines must be used to copy any inbound infor-

```
1  class ZRUser : public GSPBase< ZRUser >
2
3  //...ZRUser definition...
4
5  #ifdef ZRSIMULATION
6      //Set the memory for the user class to random non-zero
           values at allocation
7      void *operator new(size_t size) {
8          //Manually allocate memory of the requested size
9          void *p = malloc(size);
10         if (!p) {
11             throw std::bad_alloc();
12         }
13         //Iterate through the memory and assign a random
               non-zero byte at each location
14         unsigned char *memptr = (unsigned char *) p;
15         srand((unsigned int)time(NULL));
16         for (int i = 0; i < size; i++) {
17             *memptr++ = rand() % 255 + 1;
18         }
19         return p;
20     }
21
22     //To match malloc(), delete must use free()
23     void operator delete(void *ptr) throw() {
24         free(ptr);
25     }
26 #endif
27 };
```

Listing 3.5: By overriding the **new** operator, it is possible to randomly initialize the memory where the user's class will be constructed. This action will simulate inconsistent initialization of member variables on the SPHERES hardware and give advance warning that the user should explicitly initialize data members.

mation into guarded memory regions. Second, like Valgrind, there is a smaller but noticeable overhead associated with the memory checks, particularly in routines that run in iterative loops. During the 2012 ZRASCC tournament, Mudflap was removed because iterative algorithm implementations resulted in computations running over the simulation timeouts and forcing a crash of the simulation. After the simulation transitioned to a C++ version, Mudflap was abandoned due to a combination of having to re-implement the special memory transfer operations and the computational overhead.

The most promising solution to date is a special compiler called Safecode developed by researchers at the University of Illinois [19]. The compiler performs initial static analysis of the code at compile-time to check for errors and optimize runtime error checking, then monitors the code during execution. Like Mudflap, an access error will force a crash of the system. Initial tests show significantly better performance compared to Mudflap and Valgrind, and the code is compatible with other parts of the program that are not compiled with the tool, allowing for targeted error checking in just the user code for maximum efficiency.

With an array bounds checking system in place, the simulation will be able to provide additional confidence that user code will function as intended on the satellite hardware. Additional improvements include isolating which satellite caused a simulation crash when two implementations are running and feeding back stack trace information to the user to isolate the direct cause of the array access error.

3.5 Zero Robotics Simulation and Compilation Interfaces

Over the course of a Zero Robotics tournament, teams execute hundreds of thousands of simulations in a deployed version of the SPHERES simulation. There have been three distinct iterations of the simulation interface, each of which has had an important impact on the most recent design. This section will examine the common

features of the simulation as well as details from each of the design iterations.

3.5.1 General Architecture

All simulation interfaces have had several common components with varying implementations. The architecture can be divided into the following pieces, outlined in Figure 3.13:

Task Distributor This component is the main gateway between the simulation processes and incoming requests. The task distributor is responsible for assigning actions to an appropriate service. The distributor is also responsible for relaying information back to the requesting client.

Compilation Service The compilation service turns user source code into libraries that can be loaded by the simulation to command satellites.

Simulation Service Encapsulates the SPHERES simulation in a simplified interface for initializing parameters and user code, advancing the simulation time, and retrieving results.

3.5.2 2009 Pilot: Downloadable Standalone Simulation

3.5.2.1 Objectives

One of the main drivers for the first Zero Robotics interface was the short time scale for development. The project started as a rough concept in July 2009, and most of the development took place during August 2009 and early September 2009. The main objectives of the simulation interface were:

- For the dual objectives of constraining development scope and for the best possible fidelity, use the existing SPHERES simulation as much as possible.
- Only require users to obtain freely accessible software packages.

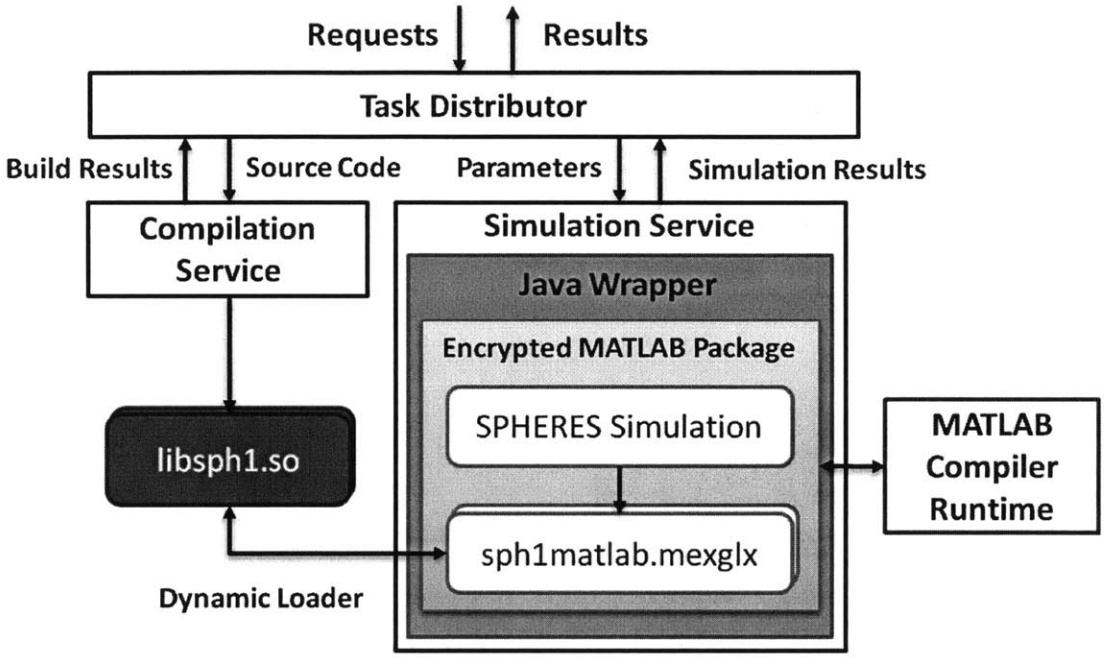


Figure 3.13: All Zero Robotics simulation architectures have featured three main components: a compilation service for creating shared libraries, a simulation service for loading the libraries and producing simulation results, and a task distributor for routing requests to and from the services. The example pictured above represents the web-based simulation architecture used in the 2010 pilot and the current simulation architecture.

- Create a minimal interface appropriate for high school students to run and analyze simulations without access the MATLAB command line.

Although later projects have increased development scope considerably, the first objective of giving users access to a high fidelity simulation has become a fundamental part of the Zero Robotics program. The remaining objectives were related to being able to distribute the simulation to students unfamiliar MATLAB environment. Looking toward the future and the desire to scale the program to tens and hundreds of teams without complicated licensing agreements, the tools needed to be freely accessible.

3.5.2.2 Architecture

The 2009 simulation was only loosely based on the general simulation interface architecture. In this case the user acted as the primary task distributor, and the compilation and simulation services were implemented as separate, standalone modules.

One of the first major design decisions was choosing a packaging method for the existing SPHERES simulation. The simulation, then in the v2009 configuration covered in Section 3.2.4, was mostly implemented in object-oriented MATLAB files, so achieving the objective of reusing the existing implementation meant finding a way to package and distribute MATLAB code. A toolbox called the MATLAB Compiler fit this purpose with the ability to generate either a standalone executable or a C++, C#, or Java wrapper around existing MATLAB code. The MATLAB compiler relies on a freely distributable library called the MATLAB Compiler Runtime, which contains a limited, headless version of the MATLAB engine and toolboxes. At the time of packaging, the constituent files of the application are placed in an encrypted archive and embedded in the executable or library. At runtime the files are extracted, and the user interacts with the program through the command line or through custom graphical interfaces.

The next challenge was giving students the ability to change behavior of the simulation with custom C code. As shown in Figure 3.5, the 2009 simulation design used compiled MEX functions *sph1matlab* or *sph2matlab* to communicate with the

simulation engine. However, the MATLAB Compiler requires that all files used by the simulation are present at packaging, and the encrypted packaging makes it impossible to replace the MEX functions with updated versions. The solution was to pre-compile the MEX functions but link them to shared libraries representing the user code. As long as the users libraries maintained the same set of exported symbols (e.g. the user gateway function `setTarget()`), and the libraries were present at the launch of the program, the implementation could be swapped out with different content.

Users developed code in the free IDE Visual C++ Express with a custom template developed by the Zero Robotics team for initializing the project files and build configuration. A typical development cycle included the following steps.

1. Initialize the project from the template.

2. Develop new code in the provided *setTarget.c* source file.

3. Compile the code for testing.

4. Move the resulting shared library to the executable location.

5. Start the executable and run a simulation.

The simulation included a limited graphical interface for performing initial positioning of the satellite and controlling the execution of tests. A 3D visualization ran during the simulation run, and additional plots could be created to view the results after completion of the tests. Figure 3.14 shows a screenshot from the simulation GUI.

The final component of the architecture was a pluggable zip archive containing scoring scripts for various games. Although most of the behaviors of the game were compiled into the simulation package, different scores could be calculated by interpreting the motion of the satellites. The game archive was extracted by the simulation executable at the start of a test run, then triggered at test initialization to reset variables, at each step of the simulation to log information, and at the end of the simulation to calculate the final score.

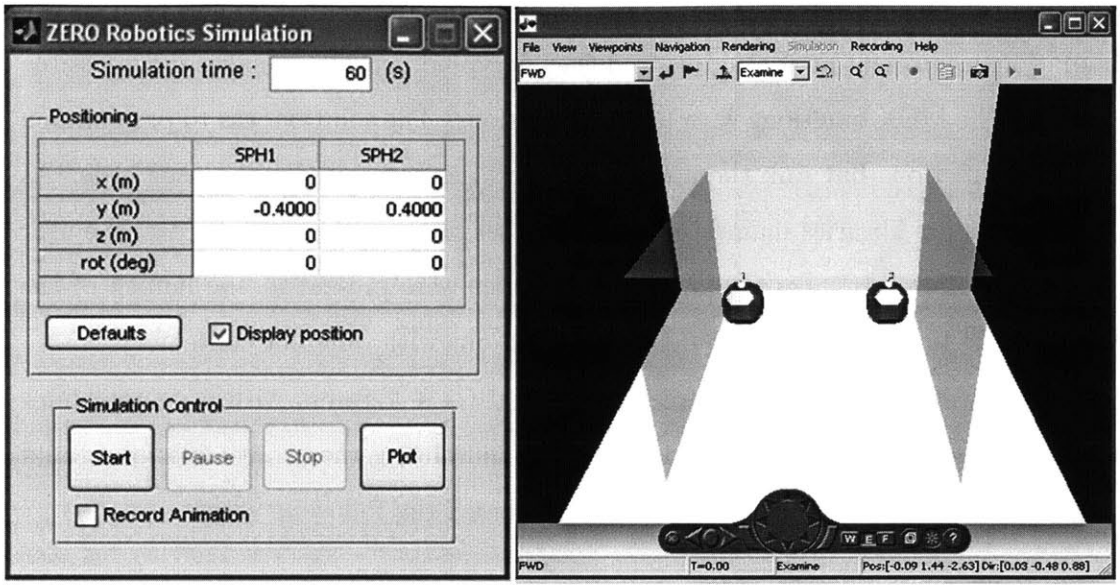


Figure 3.14: During the first Zero Robotics season, users ran simulations with a downloadable GUI wrapper around the SPHERES simulation. The interface allowed repositioning of the satellites in the test volume, simulation execution controls, and basic data analysis with plotting tools.

3.5.2.3 Limitations and Lessons Learned

As an initial stepping stone, the first simulation interface was very successful in giving high school students a gateway to program SPHERES, but it also illuminated many significant weaknesses in the approach. The following items were important lessons for designing the next simulation iteration.

Despite the convenience of a local simulation environment, many of the challenges related to maintaining the code deployment. The setup procedure required installation of at least three different programs and required significant internal development time to ensure a consistent, correct configuration. Furthermore, any updates to the game or software required users to download and install the latest version. Even with just two participating teams it was difficult to ensure all users had the latest versions of the software running. At the end of the season, one of the main requests by the pilot teams was to move the development environment online where minimal setup would be required to jump into programming.

The difficulties with updating the simulation were further compounded by the

packaging method. Any change to the internal satellite code required recompilation of the MEX functions and redistribution of the simulation. Users were also forced to quit the simulation executable before replacing the shared library. The long load times related to the startup of the MATLAB Compiler Runtime made the development cycle overly tedious. Making the game implementation part of the simulation was a clear mistake and ultimately motivated the development of the dynamic library loader now used in the simulation.

The remaining problems were less critical but still important. Users were locked into developing on the Windows operating system by the choice of Visual C++ as a development environment. This approach also meant that some of the coding limitations present in the SPHERES compiler were not enforced, and components of the submitted code had to be modified to compile. Lastly, the data analysis and playback tools were quite limited. With only start, pause, and stop commands, repeating a portion of a simulation run required a restart of the simulation or playback with a 2D recorded file.

3.5.3 2010 Pilot: First Web-Based Simulation Service

3.5.3.1 Objectives

The suggestion to move the Zero Robotics simulation to a web-based interface became an integral part of the concept proposed for DARPA InSPIRE version of Zero Robotics. Between the start of the DARPA InSPIRE program in 2011 and the end of the 2009 pilot, two additional pilot programs took place as demonstrations of the web-based architecture. The first program, a Zero Robotics middle school tournament held as part of the Summer of Innovation 2010, debuted the first web-based version of the simulation. The fall 2010 high school tournament, the first nationwide pilot, also used the prototype simulation interface. Participation jumped from 10 teams in the summer to 24 teams in the fall. To transition between a downloadable executable, and a web-based, multi-user service, several new objectives were added:

- Create a version of the simulation that can be distributed to a Linux-based

server environment.

- Separate the game implementation from the simulation and allow for multiple, easily deployable games.
- Add the capability to handle multiple simultaneous requests for simulations with minimal turnaround times.
- Robustly handle simulation crashes to maintain availability without manual monitoring.
- At every point possible in the program, the code developed on the platform should be compatible with the flight hardware and configured to run in the final tournament.

3.5.3.2 Architecture

There are several features of the 2010 simulation architecture that have remained key components of all future architectures. The first feature addressed the objectives of creating a modular, distributable version of the simulation. Starting in 2010, the SPHERES simulation incorporated the dynamic loader discussed in Section 3.3.6, making it possible to load SPHERES programs outside of the simulation package without any initial linking. To facilitate faster compilation times, the shared library was split into several components detailed in the description of the current architecture below. From this point forward it was possible to distribute games separately from the simulation.

The second part of the system, the compilation and simulation service architectures, addressed the objectives of supporting multiple users and robust simulation execution. Instead of using the MATLAB Compiler to generate a standalone executable, the simulation was generated into a Java wrapper library with exposed methods for initializing and stepping through the simulation. Crashes in user code loaded in the simulation had the unfortunate side effect of propagating all the way to the Java Virtual Machine running the library, so it became necessary to isolate

the process running the simulation from the web server. This naturally led to a distributed architecture where multiple participating processes were connected by an interprocess communication link.

The layout of the 2010 prototype website architecture is shown in Figure 3.15. Duplicate simulation processes, also called instances, were individually started and supervised by an operating system process monitor. If any instance crashed, it was restarted automatically by the process monitor. Java's Remote Method Invocation (RMI) provided the interprocess communication architecture between the simulation instances and the main web server. At startup, each process registered itself with a centralized RMI registry, and the web server, acting as a task distributor, maintained a pool of available simulation instances by periodically querying the registry. For each incoming request, the web server atomically marked the instance as unavailable, then dispatched a simulation job to the instance. If a failure occurred, the failed instance would be removed from the pool.

Separate from the simulation process, the web server contained a module for compiling user code, either for quick checks in the IDE or for full simulation runs. The compilation process first invoked an external GNU makefile in a separate process, then monitored the outputs of the compilation for errors. Successful completion of the compilation produced a shared library in a temporary folder on the filesystem to be used by an upcoming simulation request.

3.5.3.3 Limitations and Lessons Learned

The 2010 design successfully prototyped the key components of a distributed simulation architecture. The main limitations of the system were related to scaling issues with the prototype:

- The combination of RMI-based interprocess communication and the use of the filesystem for transferring files from the compilation service to the simulation service constrained the simulation system to operating on a single server.

- To preserve processing time for the web application server, at most 3 simulation

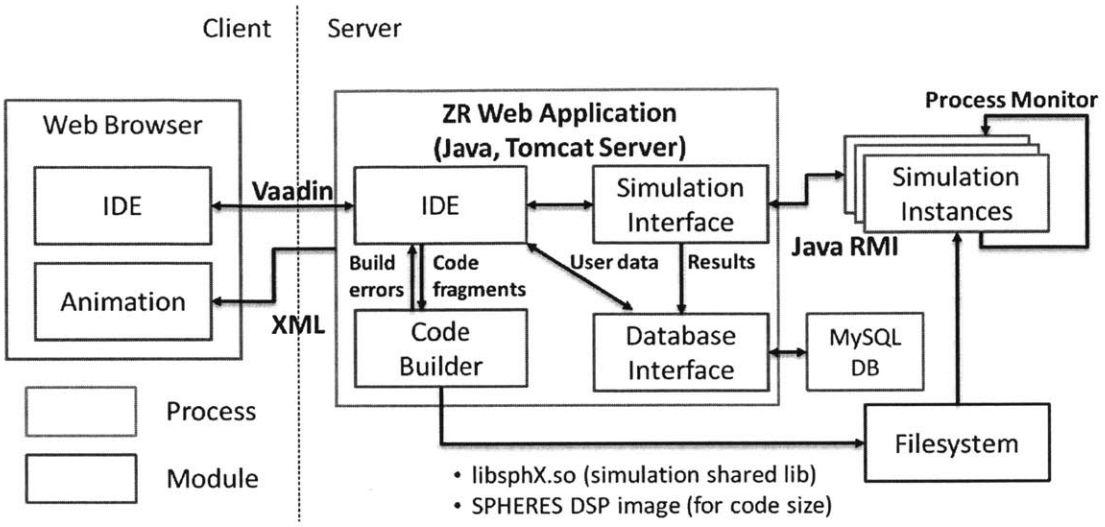


Figure 3.15: The Zero Robotics 2010 website operated on a single server with communication between several independent processes.

instances could be launched at once. This limited the simultaneous user capacity of the website, especially during peak usage times.

3.5.4 2011 Onward: Current Design

3.5.4.1 Objectives

In addition to incorporating objectives and lessons from the previous architectures, the primary objective of the present design is to provide a scalable simulation service for Zero Robotics. The architecture is intended to accommodate growth in team participation over many years.

3.5.4.2 TopCoder Simulation Farm

The current implementation is heavily integrated with a custom, distributed processing framework developed by a TopCoder member, known as the Farm. For responsive scaling, the Farm runs on virtual machine nodes instantiated on Amazon's Elastic Compute Cloud (EC2). New nodes can be added or removed from the system at will. This architecture is highly cost effective because the computing cluster can scale to meet demand during the season and drop down to a minimal configuration for

off-season loads without the purchase of hardware.

When a new node is created from a template Amazon Machine Image (AMI) , it can be configured to serve one of two roles:

Controller A single node dedicated to distributing requests. This node is equivalent to the task distributor in the general architecture.

Processor Multiple nodes for executing simulation or compilation requests. Processors can provide both the compilation and simulation services in the general architecture.

Once a controller has been activated, it can accept connections from clients over a custom TCP/IP communication protocol accessible through a Farm API. Each client identifies itself with a unique string, which is mapped to a database table on the controller. By adjusting the client configuration in the database, clients can be assigned different priority levels. For Zero Robotics, each server connected to the simulation farm uses a separate client for compilation and simulation, allowing for compilation and simulation requests to be scheduled with different priorities.

The controller node also typically runs another process called a *deployer* which is responsible for distributing game libraries and common tools to the processors when they start. A deployment starts by configuring a file and folder structure on the deployer node to be replicated on all processor node, then incrementing a configuration version. An update can be triggered by restarting the processor nodes, each of which checks its latest configuration version with the deployer upon startup and downloads the latest version if necessary.

3.5.4.3 Zero Robotics Simulation API

Just as with the GUI interface from the 2009 pilot, a simplified interface is necessary for bridging the SPHERES simulation with external applications. The Zero Robotics Simulation API is an additional layer around the MATLAB wrapper for the simulation that collapses the steps for running the simulation into two basic operations:

`init()` Perform initial startup and loading of the simulation from a specified configuration set. Includes loading of the user's shared libraries.

`step()` Advances the simulation forward by a configured step time. Steps are executed until the simulation completes as indicated by a flag in the simulation API.

Both functions operate on a set of simulation parameters that configure the specifics of the simulation execution. The configuration is designed to be composed of mostly optional arguments while giving fine-grained control over the behavior of the simulation. For example, it is possible to correctly initiate a simulation by simply supplying the location of the shared libraries containing SPHERES code and specify the starting position of the satellites. Alternatively, for detailed control, it is possible to directly modify any exported global variable in the memory of the loaded shared library, constrain the dynamics from 3D to 2D, or change the total simulation time.

When the simulation completes, the API extracts a JSON-encoded telemetry string from the simulation engine to be stored in the website database for review by users. It also records the standard SPHERES test result numbers for immediate feedback about the success of a test.

The Zero Robotics Simulation API and the MATLAB Java library are distributed as archive files to all nodes of the Farm. Updates to these libraries are only necessary when a component of the simulation or the API changes.

3.5.4.4 Game Libraries

Only user code is expected to change between simulation runs, so Zero Robotics games are pre-compiled and deployed to the simulation farm as static libraries. The libraries contain:

- SPHERES Core and SPHERES Core Wrapper
- Zero Robotics API
- Game-specific API

A separate library is compiled for each satellite and distributed to the farm processors along with header files for the game implementation. At link time the Farm processor combines the game static library to produce a shared library for loading by the simulation.

3.5.4.5 Code Size Estimation

The space allocated for user programs is tightly controlled to ensure the final implementations will fit in the available flash memory on the SPHERES satellites. A careful search for an accurate code size estimation tool took place during the 2011 season when code size constraints first became problematic. The first approached attempted to determine a scaling ratio between the size of the object code for the user's program from the simulation compiler and the size of the user code in the SPHERES images. No consistent relationship could be determined that led to predictable results, likely due to the very dissimilar processor architectures and compilers.

Ultimately, the only consistent way of creating a code size estimate was to follow the exact same compilation steps used to prepare a SPHERES image for download to the satellite. The implementation of this strategy, still in use on the current architecture, requires executing the TI DSP compiler, a Windows executable on the Linux-based processor nodes. The compiler runs under a Windows compatibility layer called Wine , driven by the same GNU Makefile that compiles the user code. The user's program is compiled and linked with a SPHERES Core and game static library (also deployed along with the simulation static libraries), then converted into a SPHERES flash image. The utility that prepares the flash image produces a total size for the image.

Estimating the size of the user code requires several more steps:

1. Before deployment, the game developer compiles the SPHERES image with an empty user program to calculate the base project size, $size_{base}$. The initial code allocation is

$$alloc_{base} = (size_{max} - size_{base})/n$$

where $size_{max}$ is the largest possible SPHERES program, currently 57344 words, and n is the total number of satellites in the final ISS image.

2. The game developer also compiles the project with a stub implementation containing calls to all API functions and commonly used functions. This is required because the compiler will not include the object code for these functions if they are never called in the program. This may lead teams to the mistaken assumption that calling specific API functions will result in more code usage than others. The resulting code size is the additional implementation "cost" of the API. Dividing by the total satellites gives an adjustment to the allocation

$$\begin{aligned} size_{api} &= (size_{stub} - size_{base})/n \\ alloc_{adjust} &= alloc_{base} - size_{api}. \end{aligned}$$

In this way, all users contribute to the overall cost of the API. This is the most conservative approach, but it risks significantly overestimating the usage and needlessly constraining the user allocation. For example, $alloc_{base}$ for the game RetroSPHERES was approximately 1550 words, while $size_{api}$, amounted to nearly 240 words per team. Nonetheless, students were allowed approximately 1600 words total due to careful monitoring of the code size of each team as the final submission approached. The adjustment step requires careful judgment from the game developer about which functions to include in the API estimate. In general, it is best to start with a conservative estimate, then periodically re-evaluate the code size based on tests with actual user code.

3. The static library distributed to the Farm processors should contain the stub implementation. Once the user code is compiled and linked to the static library, the code size estimate is determined using

$$size_{user} = size_{image} - size_{stub}.$$

The final code size is usually presented to the user as a percentage of the total

allocation

$$\%_{alloc} = size_{user}/alloc_{adjust}.$$

In addition to accurately determining the code size of the user program, the image creation approach has the major benefit of checking for compatibility with the SPHERES hardware during every code size estimate. This ensures that the final code submitted by competitors for the ISS finals will be ready to begin testing on hardware.

3.5.5 Step-by-Step Simulation Outline

To summarize the complete process for running a simulation on the distributed farm interface, this section follows the actions from initial request to final results. Several additional details have been added to clarify the actions of the web server.

1. A user makes a request to run a simulation from the web interface. On the web server, the user's project is inserted into a code template. The selected opponent's code is also assembled into a template.

2. The web server invokes a combined compilation and simulation request using the Farm API. The templated code is passed to the Farm controller along with specific parameters for the simulation run. After posting the request, the server immediately returns a response to the user.

3. The Farm controller checks for available processor nodes to handle the request. If no nodes are available, the request is queued, otherwise it is sent to the processor node.

4. The processor node starts by compiling the user code into a shared library:

   (a) Both code implementations are written to a temporary folder on the virtual machine filesystem.

(b) The processor launches a shell process to execute a GNU Makefile. The Makefile is supplied the temporary folder containing user-specific code and the base folder for the game implementation.

(c) The user code is compiled into an object file, then linked with the game object files in the game implementation to create a shared library.

(d) The user code is also compiled with the TI DSP compiler to ensure it is valid for SPHERES. Code size limitations are not enforced at this stage.

5. Next, the processor invokes the simulation

   (a) The processor launches a separate Java executable containing the SPHERES simulation library. This process separation serves the same purpose as the prototype RMI implementation for separating the simulation from the calling process. If the simulation hangs, the processor will kill the child process, and if it crashes, the processor will report an error.

   (b) Simulation parameters, including the temporary location of the satellite shared libraries are passed to the simulation.

   (c) The processor remotely invokes `init()` then `step()` until the simulation reports that it is done. At each step iteration, the processor node communicates the simulation time back to the controller, and in turn back to the web server. An asynchronous handler on the server updates the status of the simulation in the website database. The handler is also notified in the event that the simulation crashes.

   (d) The processor extracts the final telemetry results and remotely signals the web server that the simulation has completed with specific test result numbers. The web server updates the database with the full telemetry log.

6. The processor signals its availability and awaits the next request.

An important aspect of this process is the asynchronous nature of the requests passing from the web server, to the controller and processors, and back again. By returning

a response when the user first posts a request for simulation, the web server releases a thread it is consuming while the response is being handled. If it instead blocked while waiting for the entire simulation to complete, the server could rapidly deplete the pool of threads available for handling requests, leading to load-related crashes.

3.6 Zero Robotics IDE

A unique feature of the Zero Robotics platform is its built-in Integrated Development Environment (IDE). Unlike most robotics programs, Zero Robotics code development takes place entirely online, and the IDE has been gradually enhanced over time to take advantage of its integration with the rest of the website. The editor can function in two modes: a text-based IDE, primarily for high school students, and a graphical block diagram editor, primarily for middle school students.

3.6.1 Graphical IDE

3.6.1.1 Overview

Two iterations of the graphical IDE have been designed and implemented by James Francis of Aurora Flight Sciences, a Zero Robotics partner. The editing environment contains a simplified block diagram programming language used to linearly construct a sequence of actions for the satellite to follow during each iteration of the `loop()` function. There are several basic goals of the programming language, aimed at helping to introduce students to programming:

- Always produce compilable code.
- Prevent common coding mistakes like array overflows and infinite loops by construction.
- Provide a way to preview the C representation of a block diagram program.

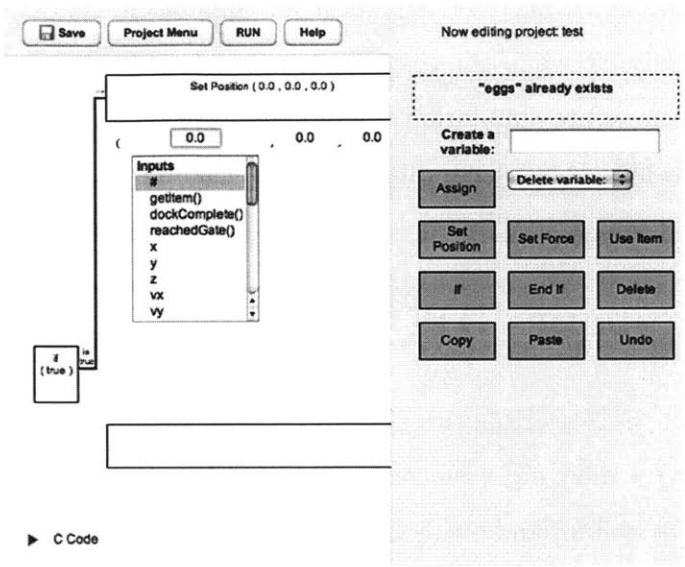


Figure 3.16: The 2010 Summer of Innovation Graphical Editor was the first prototype of the Zero Robotics graphical IDE. Users selected operations from a set of blocks containing API commands and logical operations to construct a horizontal diagram representing a single iteration of the `loop()` function.

3.6.1.2 2010 Summer of Innovation Prototype

The 2010 Summer of Innovation graphical editor was the first IDE launched on the Zero Robotics website. The IDE ran as a standalone web application embedded in a separate content management system (Joomla). A screenshot of the editing environment is show in Figure 3.16. Users started with a simple project management page (also accessed through the Project Menu button) where they could restore a previous project or create a new one. Every time the user hit Save or executed a simulation via RUN, the IDE would save a new revision of the project. Users could access the full history of incremental saves.

To create a satellite program, users selected an empty box in the block diagram, then clicked an item from the palette of available commands. The editor added a line connecting the preceding control block to indicate the direction of program flow. Conditional statements like `if` blocks created a branching structure in the diagram. Arguments to API functions were selected from drop-down menus, and basic expressions could be written by selecting operator symbols. Basic editing functions

like copy, paste, and undo could be used to manipulate the blocks in the diagram. The graphical editor also dynamically generated C code from the block diagram in real time. Users could view the C code representation of the program at any time by expanding a drop-down panel at the bottom of the screen.

Several limitations of the initial prototype prompted further development of the graphical editor:

- Each update to the diagram involved a complete redraw of the underlying HTML table, resulting in slow performance for large programs.
- The horizontal format made long programs with many branching statements run well off the screen requiring scrolling to place new blocks.
- The initial API limited implementations to very simple programs.

Despite these limitations, the prototype was very successful in its initial deployment. Students as young as 5th grade were able to participate in developing code with the interface.

3.6.1.3 Current Design: Waterbear Implementation

As part of the development for the 2011-2012 DARPA InSPIRE program, the graphical editor interface has been upgraded to address prior limitations. The new version is built on top of Waterbear[5], a block diagram programming language originally created for writing JavaScript. The graphical elements and user interaction elements of Waterbear have been modified to preserve the original code safety and code generation features of the prototype graphical language.

In this version of the editor, the user drags puzzle piece shaped blocks onto a blank canvas. Programs are constructed vertically with a similar layout to the underlying C++ code. Special blocks are available to wrap logical statements and iterative operations, and the overall toolset has been significantly expanded to include most of the Zero Robotics API. As part of the game configuration, game developers can list

[5]http://waterbearlang.com

Figure 3.17: The current graphical editor is based on a JavaScript editing tool called Waterbear. Programs are constructed by dragging puzzle-shaped blocks to form a vertical program. Functions, arguments, and conditional statements are all represented as different types of blocks.

all game-specific API functions, and they will also appear in the toolbox. Arguments are represented as blocks that fit into vacant slots in the functions. The editor enforces type agreement when arguments are dropped into a space, and more complex statements can be created by layering operators and additional arguments together. An example user program is shown in Figure 3.17.

For improved modularity of the program, users may create multiple pages, each containing its own diagram. Each page is a separate procedure that can be configured with arguments and return types. Advanced users can even mix C++ and graphical programs by adding pages containing text. When a user creates a new graphical page, an additional block becomes available in the toolbox, and the user may supply arguments to and receive values from the custom function, just like the rest of the blocks.

Users may also declare global variables accessible on all pages of the diagram. Variables can be the target of an assignment, or they can be dropped into slots as arguments.

3.6.2 Text-Based IDE

3.6.2.1 2010 Prototype

The first Zero Robotics text IDE was deployed for the 2010 nationwide pilot. The objectives of the editor were:

- Add support for editing, compiling, and simulating C code from an online editor.
- Allow for multiple, user-defined functions.
- Maintain interoperability with the graphical editor.

The last two objectives were intended for giving users a natural progression between editing in a graphical environment to editing C code directly, and in many ways, dictated the initial design of the text editor.

Shortly after the end of the 2010 Summer of Innovation program, the prototype graphical editor was enhanced with the concept of *procedures*, functions with a carefully controlled prototype declaration. Procedures were represented as separate tabs of the IDE with their own graphical editing canvas. By default, each program initially contained `ZRInit()` and `ZRUser()`, the original API entry points. Additional procedures could be added to the program by filling out a dialog to create the function prototype. Prototypes were validated with the graphical editor, then registered as new blocks in the program.

To add text editing capability, the IDE extended the idea of procedures by replacing the graphical editing canvas with a syntax-highlighting text editor. The process of creating a new procedure remained the same, except users were provided the option of selecting between graphical and text editing modes. In this way, text procedures also appeared as blocks in the graphical editing toolbox.

This approach was not without drawbacks. While the user was free to specify an arbitrary number of arguments, their types, and the return type of the procedure were constrained to maintain compatibility with the simplified inputs and outputs of the graphical editor. For example, during the 2010 tournament it was initially impossible to pass an array as an argument to a procedure. Adding support for more complicated types often broke compatibility with the graphical editor.

In retrospect, these objectives may have overly constrained the design. Most high school participants have either elected not to use the graphical editor, or due to other development priorities, it has not been possible to have both editing systems working at the same time. In the transition to C++, compatibility with the graphical editor was partially dropped in favor of releasing constraints on the text editing environment.

3.6.2.2 C++ Text Editor

The current text editing environment significantly simplifies the interface for constructing programs in the IDE. Instead of dividing the project into procedures, the user is now given full control over function and variable declarations. Code is entered onto a series of *pages*, user-defined logical divisions of the program. Before compilation, the pages are sorted alphabetically and simply concatenated together. As noted in Section 3.4.3, the user code is embedded in a C++ class body, so naming conflicts cannot occur between user-defined functions. With this interface, users may access many features of the C++ language, including declaration of custom classes[6].

Partial compatibility with the graphical editor is maintained with the philosophy that users will tend to progress from graphical projects to text projects. When creating a project, a user may choose to start the project with the graphical editor or the text editor. Text projects start immediately with a text editor containing empty `loop()` and `init()` functions, while graphical projects start with the entry point functions as graphical procedures. In both modes the user may append new pages to the project with the option to make them text pages or graphical procedures. New procedures still utilize the dialog-based function declaration system and are registered with the graphical editor to appear in the block diagram toolbox. Text pages are simply appended to the project and their contents cannot be referred to from graphical programs. However, if a graphical procedure is converted into a text page, the function definition stays in the graphical toolbox, allowing for a mix of graphical and text code.

3.6.2.3 Project Revision Control and Collaborative Editing

The latest editing environment implements a set of features to enable multiple users to simultaneously edit a project, merge changes, and address conflicts. Several of the initial concepts for collaborative editing were developed by Thai in [70].

[6]Since any declared class resides within the body of the surrounding code template class, it is considered a C++ *inner class* and must be declared at the top of the program (first page alphabetically).

Most web-based collaborative editing environments, such as Google Drive™, focus on synchronizing the state of the collaborators' screens with as little delay as possible. In contrast, code editing poses a unique challenge because immediately synchronizing text between editing screens will almost certainly break the ability to compile the program. Even if compilation errors are resolved, if collaborating authors simultaneously change different regions of the same program, they may break each other's assumptions about how the program is functioning, leading to complicated debugging scenarios. In [28], Goldman partially addressed the compilation issue by only merging changes from programs that compiled. The approach depends on continuous compilation of the project by the server back end and must still address the program behavior conflicts. In situations where the loop from making changes in the code to seeing outcomes is on the order of seconds, this approach might be feasible. For Zero Robotics, processing a single simulation takes approximately 10-30 seconds, and a cursory review of the results can take a minute or more, so the loop hinders real-time code merging.

The current approach relies on a mix between traditional repository-based version control systems and real-time communication enabled by a web presence. First, the editor implements a version control system similar in style to systems like Subversion or CVS. When a user creates a new project, the server establishes a special version of the project to be considered as the *trunk* or repository copy. Any user opening the project for the first time triggers the creation of a separate *working copy*, based on the latest revision of the project.

As the user works, the IDE periodically saves the project, preserving changes in the event the user is disconnected from the website. The saves are persisted in the user's working copy.

When the user is ready to share changes with other collaborators they issue a *commit* command and select a list of pages they wish to post the server. The server copies the specified pages into the project trunk and increments the revision number of the project. Before a commit, the user is required to perform an *update* operation to retrieve the latest code for the selected pages. Upon update, the IDE will attempt

to automatically merge any changes from the server into the local copy for the selected pages. A simplified difference and merging tool is provided for completing this task, including resolving conflicts between the procedure versions. At any time the user may also choose to *revert* changes to a page to match the corresponding parent in the trunk. Due to the ability to partially commit the project, different pages may have parents at different revisions in the trunk. An indicator on the upper right side of the page shows the user the revision number of their page and the latest revision of that page.

For real-time collaboration, the editor takes advantage of the simultaneous presence of multiple users working online. Special indicators, based on the status lights created by Thai, help to alert the user that changes are taking place. There are three types of status lights, intended to convey consistent messages:

No Light No changes have been made.

Green (All Clear) Local changes exist, but the user may commit without conflict.

Yellow (Warning) Another user is currently editing the page.

Red (Potential Conflict) There have been simultaneous edits of the same page. It may be necessary to address conflicts when merging the pages.

Since merging conflicting changes can be quite complicated, the general philosophy of this approach is to provide full freedom to simultaneously edit the projects, while providing guidance to the users that they may be entering into potential conflicts.

Editing status is communicated between clients via an asynchronous connection to a websocket server. At IDE startup the client initializes the connection and subscribes to a channel specifically designated for the current project, similar to a chat room. Whenever a user edits a page, the client posts an update to the socket server containing information about which pages are being actively edited. All other clients asynchronously receive the update notifications and update their status lights based on the information.

The clients also transmit a special patch, computed as the difference between the current page and the parent revision in the trunk of the project. Other clients can apply the patch to a page to construct a real-time preview of the text another user is typing. The patching behavior can be problematic if users never commit their projects and the patch ends up containing most of the text in the program.

3.7 Data Analysis Tools

3.7.1 3D Visualization

The Zero Robotics visualization is the primary tool available for reviewing simulations[7]. The visualization is an essential tool for *Efficient Inquiry* because it allows teams to review simulations without re-running the test.

After a simulation completes, the web server persists telemetry data produced by the satellite, including:

State Estimates Telemetry from the SPHERES state estimator. Used to show the user where the satellite thinks it is located.

Debug Data All Zero Robotics games transmit game state in 32 byte telemetry packets known as *debug* packets. All information needed to synchronize the game between satellites or update the visualization with game information is transmitted here.

User Data The user may optionally send up to 7 floating point values for custom analysis.

Text Data The simulation captures text printed by DEBUG statements and stores it in the telemetry.

Separate time stamps are associated with the state, debug, and text data. To create an animation, the visualization constructs a timeline starting with the first time

[7]The first online visualization was first created for the 2010 Summer of Innovation program by Aurora Flight Sciences and later updated by the author.

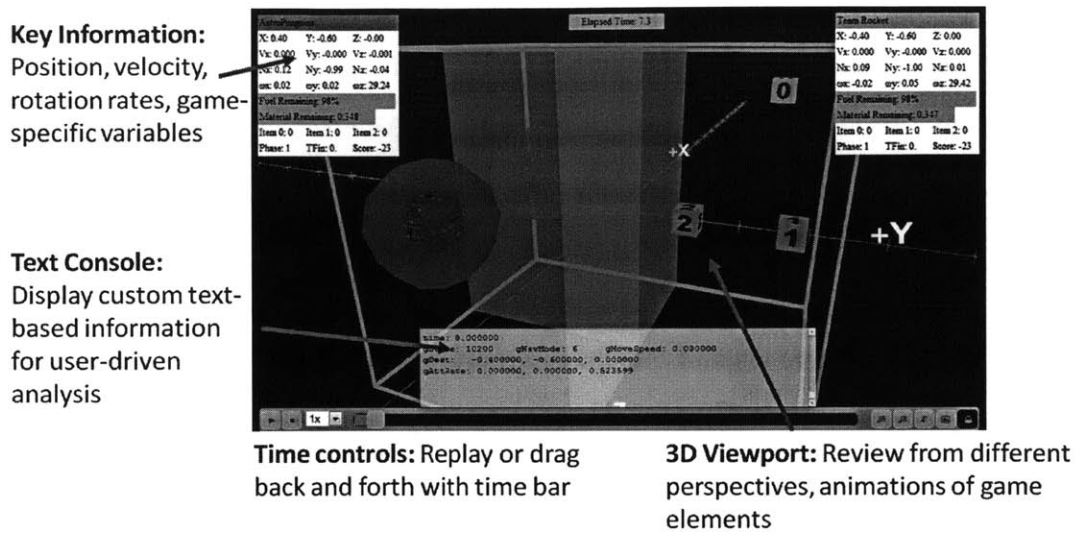


Figure 3.18: The Zero Robotics 3D Visualization

stamp and ending with latest time stamp. The sequence of states is pre-processed to establish a set of keyframe points for each time the satellite transmitted a state telemetry packet. During playback, a built in *tweening* engine automatically interpolates the states between the keyframe points to smooth the animation at each rendering update. Debug telemetry is also inserted into the timeline as a series of events. As the playback sweeps past the time when telemetry packet was sent, the visualization triggers a callback in the game-specific animation code to perform an action. Using the telemetry, any component of the display can be updated.

An example view of the visualization is shown in Figure 3.18, highlighting the main user interaction features. The results can be viewed from multiple angles, and the animation can be played at accelerated speeds. A slider bar at the bottom of the viewport allows the user to drag time forward and back to repeat a specific slice of time. User debug text is printed to a console at the time steps associated with the text.

3.7.1.1 Report Tool

A second tool to aid *Efficient Inquiry* is the Report Tool. The tool complements the 3D visualization with a more quantitative view of the data in the form of line

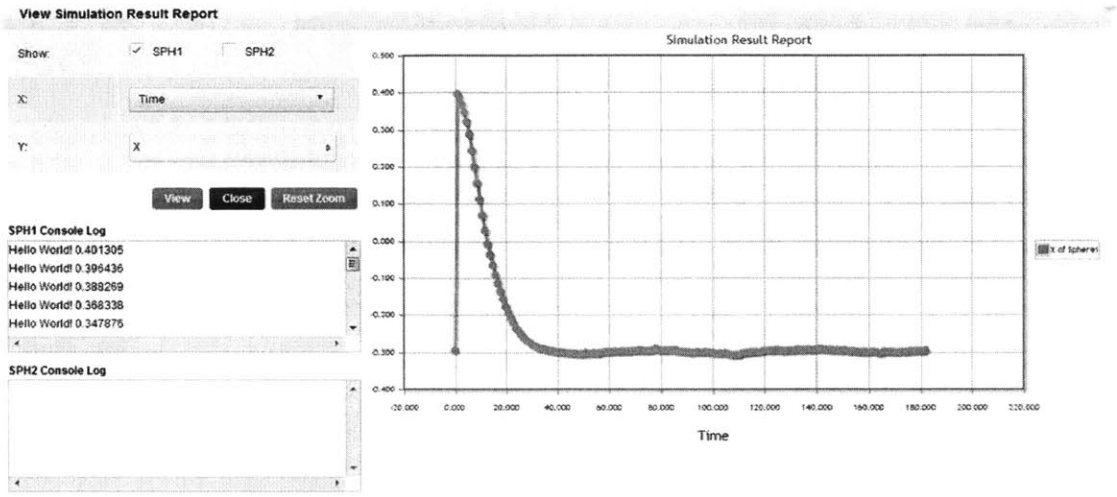


Figure 3.19: The Zero Robotics Report Tool

plots of telemetry data. The plots can include up to 7 different user-specified data values as well as the satellite state. Important parameters for each game such as the score can also be displayed. An example plot showing the X position versus time in a simulation is shown in Figure 3.19. A user can select data for both X and Y axes to analyze the behavior of a variable with respect to time or with respect to another variable.

3.8 Zero Robotics Website

Since much of the activity for Zero Robotics takes place online, there is a unique opportunity to centralize both the resources for the program and the online community. The main site contains resources for developing code, running simulations, and analyzing results, and it is the dedicated launching point for the Zero Robotics tournaments. The site includes:

Public Information Page Displays information about currently active tournaments, basic information about the Zero Robotics program, and clear instructions about how to get started. It is important to make this component of the website easily readable and visually attractive to draw guests into the site.

Tutorials A collection of organized, sequential lessons to climb the learning curve of programming for Zero Robotics. The tutorials also introduce relevant concepts in math and physics, then tie them to programming examples.

Forums Dialog between users can greatly boost the ability of the site to respond to user inquiries and build a sense of community around participating in the activity. In a 2010 survey of *FIRST* robotics teams, 74 percent of participants actively used the popular but independent forum Chief Delphi to exchange information[65]. By focusing on building an effective forum tool from the outset, Zero Robotics keeps discussion under the same organizational banner to take advantage of site integration features like profile linking and tournament performance statistics.

Support While user to user support facilitated by forums takes care of minor learning curve issues, bugs and website performance problems need to be directed to the website managers. A ticket-based support system allows support staff to track and respond to requests initiated by users of the site.

Styling and implementation of the website were accomplished using TopCoder crowdsourcing competitions. These competitions created the cloud-based server architecture for hosting the website along with the code implementations of the website components. More information about the website components and their development can be found in the Nag's study of crowdsourcing applied to the Zero Robotics platform [53].

3.9 Summary

Through the design of the Zero Robotics platform this chapter has established the core components necessary for a virtual robotics competitions based on a real hardware. The components can be generalized as:

- A detailed model of the robot (the SPHERES simulation)

- A programming interface for controlling the robot (the Zero Robotics API)

- A user-friendly framework for connecting the model and programming interface (the Zero Robotics website and compilation infrastructure)

While the components have been deployed to an online web-based environment for the broadest possible access, it is important to note that a similar framework could be developed for offline use as in the first Zero Robotics season, though it would lack the benefits conferred by an online presence.

Based on accumulated lessons from many iterations of SPHERES simulations, the current simulation contributes several essential characteristics for effective design of simulations in the context of a robotics competition platform:

- *Software-in-the-Loop Capability.* To increase the chance of success on the hardware platform, the code running in simulation must be easily transferable from the simulation environment to the robot.

- *Modularity and Dynamic Loading.* Separating the simulation into a generic core executable containing only the dynamics and minimal software interfaces and a separate module to load user-specific code saves an immense amount of compilation time for each simulation run. The modular implementation also allows new games to be deployed without changing the simulation.

- *Operating System Level Software Model.* Timing and execution should be modeled with sufficient detail to ensure consistent execution on the hardware. Based on the thread synchronization method in section 3.3.5.3, the SPHERES simulation closes the gap as much as possible between software running in simulation and software running on the robot while preserving execution speed.

- *Compromise Between Complexity and Ease of Future Expansion.* The current simulation makes use of the modeling framework Simulink for easier customization of the satellite dynamics model. Where Simulink is overly restrictive, mainly for the SPHERES Core model, workarounds are implemented in C++ to

provide better fidelity. This compromise helps to keep the simulation accessible to the SPHERES research team and aids fast turnaround changes to the model.

- *High Portability.* By using code generation and cross-platform C++ libraries, the simulation can be transferred between operating systems. This is critical for simulations that run on cloud computing infrastructure primarily based in Linux, or when making the simulation freely available for download by a broad spectrum of PC users.

The Zero Robotics API contributes a simplified programming interface for controlling 6DOF satellites. The API includes a layered set of controllers that promote *Incremental Difficulty* with options for selectively commanding position, velocity, or low level forces and torques. Users have access to a simplified attitude representation or can choose advanced control with quaternions. With a layer to emulate the double integrator dynamics of the satellite the API should be general enough to apply to other holonomic systems like quadrotors or mecanum-drive robots provided some of the degrees of freedom are constrained.

Lastly, the combination of online programming and simulation environments is still quite unique among modern web applications. In the years spanning the implementation of the Zero Robotics platform, a large variety of online IDEs have emerged in a host of programming languages. The overall architecture for these programs likely follows a similar pattern to the Zero Robotics IDE and back end processing farm, but few if any connect the compiled code to a dynamic simulation. This additional integration enables the website to be a centralized, zero-configuration resource for learning and experimentation.

Chapter 4

Zero Robotics Scoring Systems

4.1 Introduction

Chapter 2 established a set of guidelines for designing and scoring individual Zero Robotics games and matches, but the issue of evaluating performance over the course of a competition is the subject of this discussion. Each of the four years of Zero Robotics tournaments has involved a different method of competition scoring, driven by the growth of the program and shortcomings identified in previous seasons. One of the main contributions of this thesis is the development of a continuous ranking system for the platform called the Zero Robotics Leaderboard. The Leaderboard improves upon previous Zero Robotics ranking systems by:

- Providing teams multiple opportunities to submit code, compete against opponents, and evaluate performance throughout the competition.
- Spreading computational load for ranking teams throughout the competition season.
- Computing nearly instantaneous estimates of team standings.
- Supplying teams with tools to analyze performance at each submission.

This chapter first reviews the history of Zero Robotics competition scoring systems, then presents the most recent Leaderboard scoring algorithm adapted from

Whole History Rating, a Bayesian skill rating system developed by Coulom in [15]. Implementation-specific details to improve the stability and responsiveness of the algorithm are discussed. Performance of the algorithm is reviewed from the perspectives of accuracy, user experience and programmatic outcomes for the 2012 high school tournament. The discussion concludes with recommendations for future implementations based on the 2012 results.

4.2 Other Zero Robotics Ranking Systems

4.2.1 ZRHS 2010 and ZRHS 2011: Round-Robin Competitions

In both 2010 and 2011 seasons, all-to-all round-robin competitions were used to rank teams for ISS down-selection. Participants in the 2010 high school tournament first competed in a live 2D competition from the MIT flat floor facility in a double elimination bracket, then competed in a simulated round-robin tournament in 3D. A mix of scores based on rank in the 2D competition and the total wins in the round-robin phase selected the finalists to proceed to ISS.

The 2011 game AsteroSPHERES included a cooperative component where teams could work together to optimize scores in individual matches. For this to be a benefit, the competition-wide scoring system had to reward high-performing teams based on results from multiple matches. The solution was a single round-robin tournament performed at the end of each competition period. Teams were ranked by their cumulative score over all matches.

After running two full tournament based on the round-robin system, the following drawbacks were identified:

- The final batch simulation took many hours to run and resulted in thousands of simulations to store on the website. For the number of teams in the 2011 competition, this did not present a problem, but it presented a concern for future growth of the program.

- Teams only had one chance to make a strong submission. A small error or unanticipated situation could result in poor performance. The likelihood of these events was increased because the teams only encountered other team submissions during the competition with no chance to revise code.

- Balancing issues and bugs in the game were not fully realized until after official results were released to the teams. The first chance to observe real teams competing in the competition came at the end of the competition period, and any corrections to the game or scoring algorithm required a retraction of official results.

While the system served its purpose for scoring HelioSPHERES and AsteroSPHERES, the downsides warranted additional exploration of alternative scoring systems.

4.2.2 ZROC 2012: Relative Scoring Leaderboard

The main deficiencies of the round-robin competition format stemmed from the emphasis on a one-shot batch based on a single submission from the teams. To distribute scoring over many matches other gaming environments implement real-time centralized scoring systems to rank participants as they compete. The list of rankings, or *leaderboard*, gives competitors continuous feedback about their current performance and motivates constant improvement while providing an opportunity to recover from mistakes.

The 2012 Zero Robotics Autonomous Space Capture Challenge (ZRASCC) featured the first leaderboard-style scoring system used in a Zero Robotics tournament. The Zero Robotics Leaderboard differs most with gaming leaderboards by the fact that the competitors are autonomous programs. To compete on the Leaderboard teams make a *submission* consisting of a program–alternatively called a *player*–to be scored. The submission is paired with several other teams and simulated matches are conducted to produce a sample of match outcomes. A ranking algorithm processes the match outcomes to establish the scores. Once the submission has been entered on the Leaderboard, it stays active to "defend" its position indefinitely. If the team

is matched with another submission, their player competes with the new submission, and the ranks of both teams are updated.

ZRASCC was an algorithmic challenge to optimize fuel consumption while docking with a tumbling target, but individual matches at each submission were run head-to-head against an opponent. One of the teams in the match selected a set of parameters for the tumbling target, then both players worked independently to complete the docking maneuver. The final score used for ranking was the difference in the fuel consumption between the two players. Teams were ranked on the Leaderboard by their average point difference over all matches. At the end of each of the four weeks in the tournament, the Leaderboard was reset and the challenge was modified to add additional problems to solve.

The relative scoring format was intended to encourage teams to out-perform their opponents by: 1) choosing difficult tumbling target parameters and 2) developing a robust docking implementation that could match a wide variety of docking scenarios. Promoting this behavior proved to be a struggle. Without absolute benchmarks, the relative scoring system only awarded higher performance with respect to other competitors. Many novice teams could not complete the challenge or could only complete a basic scenario with low efficiency. When competing against these teams, it was possible to advance in rank simply by performing well in the basic scenario, eliminating the incentive to try more difficult strategies. In the second week of competition, nearly every team chose to compete in the same scenario and additional rules were necessary to encourage more exploration.

The relative scoring system also highlighted the difficulty of ranking players based on game-specific scoring metrics. As the tournament progressed the scoring system had to be updated several times to account for small loopholes in the way points were awarded. While it was usually clear from the match data which satellite performed better overall in the challenge, choosing a scoring system that fairly weighted performance in all scenarios took several trial and error attempts.

4.3 ZRHS 2012: Whole History Rating Leaderboard

As with ZRASCC, the 2012 high school tournament introduced a leaderboard scoring system with the motivation of engaging students throughout the tournament season. Based on the challenges with effectively ranking teams in ZRASCC, the scoring system switched from a average point system to one based on win-loss outcomes. This form of ranking is less susceptible to manipulation and has stronger theoretical underpinnings. It also has broader applicability to Zero Robotics because the game rules are only responsible for producing a winner and a loser in each match. For the most part, the game scoring system can be developed independently of the tournament without concern that the scoring system will be coupled into the competition dynamics.

The Leaderboard algorithm is based on the Whole History Rating (WHR) method developed by Coulom in [15]. WHR is a Bayesian rating system that accounts for time-varying changes in the skill of the participating players. The probabilistic formulation of the algorithm is particularly useful for rating games where the outcomes are stochastic.

4.3.1 Overview of the WHR Algorithm

Individual matches in a competition are represented with a Bradley-Terry paired comparison model [6], where the outcome of a match is predicted by the relative ranks r_i and r_j of the two players. Let P be the probability that player i beats player j, then the Bradley-Terry model predicts

$$P(i\,wins|r_i, r_j) = \frac{e^{r_i}}{e^{r_i} + e^{r_j}}. \tag{4.1}$$

In the event that two players have the same ranks $P = \frac{1}{2}$, so the ranking number can be interpreted for a given player as a dividing line between teams that will perform (on average) better than the player and those that will perform worse.

Ranks are computed by observing the outcomes of games and performing an estimation algorithm based on Bayesian inference. Ultimately, we are interested in

computing $p(r|\mathbf{G})$, the posterior distribution of probabilities of a player's rank given the game outcomes $\mathbf{G}$. The rank can be inferred from the the observations with Bayes' Rule

$$p(r|\mathbf{G}) = \frac{P(\mathbf{G}|r)\,p(r)}{P(\mathbf{G})} \tag{4.2}$$

where $p(r)$ represents a prior distribution of the rank, and $P(\mathbf{G})$ can be viewed as a normalizing constant. Instead of explicitly calculating the full posterior distribution, the WHR algorithm computes the value of r that maximizes $p(r|\mathbf{G})$, or the maximum *a posteriori* (MAP) estimate of r.

One of the key components of WHR is its representation of the prior distribution $p(r)$. Instead of assuming ranks remain static throughout the competition, player skill is assumed to vary with time following the random walk pattern of of a Wiener process

$$r(t_2) - r(t_1) \sim \mathcal{N}\left(0, |t_2 - t_1|\, w^2\right). \tag{4.3}$$

The times t_1 and t_2 are individual *Epochs* at which a rank is calculated and might contain one or more matches. The parameter w is the main tuning parameter of the model, which controls the growth in the rank variance with time. A high value of w assumes that ranks will change quickly and heavily weights new match observations, while the limiting case of $w = 0$ assumes that the ranks are static. A well-tuned value of w will produce a compromise between previous match performance and new outcomes. Qualitatively, this behavior is important for preventing wide swings of ranks and helps to ensure a single set of bad matches or poor performance in the early phase of the competition does not doom a player's prospects. The Wiener process assumption is important because it is a Markovian process,

$$p(r_k|r_{k-1}, r_{k-2}, \ldots r_0) = p(r_k|r_{k-1}) \sim \mathcal{N}\left(r_k - r_{k-1}, |t_k - t_{k-1}|\, w^2\right).$$

With the models for individual match outcomes and the evolution of rank over time, the next step is to calculate the ranks. As the name indicates, the WHR algorithm performs the MAP optimization over the "Whole History" of ranking epochs.

Each update produces not only the current rank, but a revised history of the ranks at all previous epochs. This is in contrast to other rating systems such as TrueSkill [31], or the standard Elo rating system for chess [20], which compute recursive updates to a rank based on new match outcomes[1]. The estimate is expected to be more accurate than recursive estimates because it "corrects" errors in previous rank estimates with the most recent information.

To compute the rank estimates, Equation 4.2 can be expanded as a Markov chain at and between each epoch. Neglecting the normalizing factor, and using the shorthand $r_k \equiv r(t_k)$,

$$P(\mathbf{G}_k|r_k) = \prod_i P\left(g_k^{(i)}|r_k, r_{other}\right) \tag{4.4}$$

$$\begin{aligned} p(\mathbf{r}|\mathbf{G}) &= P(\mathbf{G}_n|r_n)\, p(r_n|r_{n-1})\, P(\mathbf{G}_{n-1}|r_{n-1})\, p(r_{n-1}|r_{n-2}) \ldots P(\mathbf{G}_1|r_1)\, p(r_1|r_0)\, p(r_0) \\ &= \prod_{k=1}^{n} P(\mathbf{g}_k|r_k)\, p(r_k|r_{k-1}) \end{aligned} \tag{4.5}$$

Equation 4.4 is the probability over all matches $M(k)$ with outcomes $g_k^{(i)} = \{won, lost\}$, $i \in M(k)$ at an epoch k, and Equation 4.5 is the probability of the entire sequence, including the variance evolution due to the Wiener prior. The final term $p(r_0)$ is the initial rank prior. For this application, the prior was initialized to a win and a loss to a player of rank 0, which sets the player's initial rank to 0. The estimation algorithm optimizes over log probability of the sequence, which conveniently converts the products into sums and avoids numerical problems with repeated multiplications of small numbers.

The optimization for the MAP estimate of the rank vector $\mathbf{r}$ is performed running several iterations of Newton's method

$$\mathbf{r} \leftarrow \mathbf{r} - \left(\frac{\partial^2 \log p}{\partial \mathbf{r}^2}\right)^{-1} \frac{\partial \log p}{\partial \mathbf{r}} \tag{4.6}$$

until convergence of the gradient $\mathbf{g} = \frac{\partial \log p}{\partial \mathbf{r}}$ to $\mathbf{0}$, the change in $\mathbf{r}$ is within a spec-

[1]There is a variant of TrueSkill called TrueSkill Through Time [17] that does incorporate the full rating history.

ified tolerance, or a fixed number of iterations are completed. Typically, a second order solver like Newton's method would be computationally prohibitive over large sequences of matches and hundreds of teams, because each gradient computation involves the costly $O(n^3)$ inversion of a Hessian matrix. However, the WHR formulation makes two important simplifying assumptions:

1. During a ranking update, all other ranks are fixed.

2. As noted above, the prior is Markovian.

With these two assumptions, at each epoch, the rank is related to at most the next rank and the previous rank, leading to a Hessian with a tri-diagonal structure. Coulom exploits this special structure with an $O(n)$ LU decomposition to perform the combined matrix inversion and solution of the linear system in 4.6.

To update all ranks, Newton iterations are performed one at a time for each rank, keeping all other ranks constant. After several iterations through the entire set of ranks, information from new match data propagates through the entire rating system through the rank histories. Even if two players do not directly compete, relative ranking information can propagate between the players through a third party proxy. This is another significant advantage of updating the entire rank history.

An important feature of the formulation is the natural weight of match outcomes by the relative rank of the players. If a match outcome is unexpected, either a win to a higher ranked player or a loss to a lower ranked player, the optimization algorithm will adjust the rank up or down to make the outcome more likely. The higher the disparity, the greater the change in rank. This behavior addresses the issue of balancing "easy" wins with "hard" wins experienced in the ZRASCC rating system.

4.3.2 Improvements and Implementation Considerations

4.3.2.1 Stability Improvements with Armijo Iteration

The Newton step $\mathbf{H}^{-1}\mathbf{g}$ in Equation 4.6 is optimal in regions of the function that are exactly or nearly quadratic. The update can be too aggressive when attempting to

run the algorithm in situations where the initial guess is far from the optimum such as when a new epoch is added to the rank history or when attempting to reconstruct rank history from previously unranked data. Coulom appears to have recognized this in the formulation of the hessian, but the proposed solution to subtract a constant (0.001) from the diagonal of the hessian for numerical stability (see B.1 in [15]) does not work consistently.

The stability of the algorithm can be greatly improved by implementing a *sufficient increase*[2] criterion for the step size when performing the Newton updates. The Armijo Rule criterion ensures that each step taken will result in an increase of the objective function by at least some fraction, $\sigma \ll 1$, of the increase predicted by a step of the same size with normal gradient ascent. The procedure is described in Algorithm 4.1. In each iteration through the inner loop, the condition on line 8 checks to see if the current step size will result in a sufficient increase and terminates if the condition is satisfied. If the condition is not satisfied, a step size reduction factor $\beta < 1$ is applied to the step, and the iteration repeats. The sequence of trial steps is therefore $\beta^0\alpha$, $\beta^1\alpha$, $\beta^2\alpha, \ldots, \beta^n\alpha$. If the algorithm reaches an iteration limit, an error can be triggered for safe handling and cleanup in the calling function.

To choose β and σ, it is usually best to make β as close as possible to 1 to take large step sizes and σ very small to make sure even marginal increases are accepted. The values used for the Zero Robotics ranking system, $\beta = 0.9$ and $\sigma = 10^{-6}$, have shown good performance with real ranking data. When reconstructing the match ranking information for the 2D and 3D simulation competition analyses in the remaining sections over approximately 70,000 matches, the algorithm never exceeded the iteration limit. For faster convergence, it may be possible to raise β to a value closer to 1 with some experimentation.

[2]The rule is more frequently stated as a sufficient *decrease* criterion in the context of function minimization.

Algorithm 4.1 WHR Algorithm with Armijo Iteration

```
1: while NOT converged do
2:   g ← gradient(r, Δt, G)
3:   H ← hessian(r, Δt, G)
4:   d ← H^{-1}g
5:   α ← 1
6:   while iter < max_armijo do
7:     r_tmp ← r − αd
8:     if loglikelihood(r_tmp) − loglikelihood(r) ≥ σαg^T d then
9:       r ← r_tmp
10:      break;
11:    α ← βα
```

4.3.2.2 Penalty for Variance

The TrueSkill ranking algorithm presented by Herbrich, Minka, and Graepel in [31] has the compelling feature of tracking both the mean μ and variance σ^2 of a player. A player's score for ranking is computed as $\mu - 3\sigma$, representing a 99% certainty that the player's true rank is higher than the score value. This is a useful addition to a scoring system because it awards both higher performance and consistency. A player that wins consistently against moderately high-ranked opponents (low variance) can be ranked higher than a player that wins against high ranked players but loses to low ranked players (high variance). To append a similar consistency metric to WHR, the Zero Robotics Leaderboard calculates both the epoch's rank mean and variance, both of which are provided by the WHR algorithm, then applies the TrueSkill weighting for ranking on the leaderboard.

4.3.2.3 Matchmaking

One of the motivations for implementing the WHR algorithm is to reduce the number matches needed to accurately score a team at each submission. If the number of matches is smaller than the number of teams, we wish to choose matches in a way that maximizes the amount of information content in the match outcomes. This problem has been studied from the perspective of adaptive tournament design for the Bradley-Terry model in [27], but has not been implemented on the ZR platform

due to the complexity of the algorithm. However, one of the observations from [27] is that the best matches tend to have means that are close together. Therefore, a simple heuristic matchmaking method is to choose the n teams with the closest rank mean score. A downside of this method is that it can prevent teams from competing against high ranked teams to get the added benefit of prevailing in an unlikely match outcome. This can be addressed by making n reasonably large, so each submission spans a large breadth of team ranks, and the selection can be biased toward higher ranked players.

4.3.2.4 Penalizing Ties

The WHR formulation only admits win/loss match outcomes in the probabilistic model. To introduce ties in the model, an explicit probability must be assigned to the possibility of a tie. In [38], Hunter suggests two options for incorporating ties with a modification of the Bradley Terry model but both require changes to the WHR algorithm. Typically these models assume tied outcomes represent an equal matching of skill.

In the 2012 RetroSPHERES competition, tied outcomes were only possible if neither competitor completed the challenge, so ties were not desirable from the perspective of the game intent and did not necessarily represent an even matching of skills. In this case tied outcomes were allowed, but a penalty was introduced to discourage intentional tying. In the WHR framework it is not possible to represent a tie as a double loss (team A loses to team B and vice versa), because the probabilistic model assumes one team wins and the other team loses. Using a double loss will result in the each pass of the algorithm incrementally adjusting team A down in rank to account for the loss to team B, then adjusting team B down to account for the loss to team A, and the scores will ultimately diverge.

Since the objective of a penalty is to cause a reduction in the score of the offending teams, an approach compatible with the WHR system is to introduce a fictional low rank that will be below the skill level of all teams on the leaderboard. Penalties are then scored as a win and a loss against a player with the fictional rank. Both win

and loss are necessary because if only a loss is used, the ranking algorithm will keep adjusting the player's score down to account for the fact that the player never wins against the fictional player. Because the fictional rank is fixed, the win and loss will pull the tying teams toward the low value but not below it.

This method succeeded in lowering the scores, but the case study in Section 4.3.4 suggests ties are best avoided without an explicit representation in the algorithm.

4.3.2.5 Grouping Identical Submissions

When scoring autonomous players created by humans, the assumption that skill varies with time is only valid if the program has been updated. If the program is updated, it is reasonable to assume the humans programmers have had a change of skill based on observing matches and making modifications to the program. If the program has not been changed but instead repeatedly submitted, the additional submissions should be grouped with the initial program submission since there has been no change with time. Judging what constitutes a significant change to the program is nearly impossible without detailed analysis of the simulation results because even a small parameter change can have a large effect on the satellite behavior. Instead, for the Zero Robotics platform, a submission is considered new if a new version of the program has been saved regardless of any change.

Ideally there should be no need for users to make identical submissions to the scoring system, but for large changes in the player performance it may take many matches for the system to respond. Tuning the minimum epoch period will help to improve the responsiveness, and additional enhancements have been suggested in the future work section.

4.3.2.6 Minimum Time Period

As teams iteratively improve their programs, they may make many closely spaced submissions as new problems are discovered or they encounter different opponents at different rank levels. Many repeated submissions will decrease the uncertainty in the rank with more samples, but it can also lead to the system becoming progressively

less sensitive to new match observations. This problem is similar to the issues faced by a standard least squares parameter estimator attempting to track slowly time varying parameters. Eventually the covariance of the parameters collapses to 0, and future measurements are ignored. A common solution is to inject a small amount of additional parameter certainty at each covariance update to keep the system sensitive to new measurements. The Wiener process in the WHR algorithm provides a similar effect over longer periods of time, but in several rapid submissions, the system may respond slowly to sudden changes in rank.

A good solution is to put a floor on the variance by setting a virtual minimum time t_{min} between ranking epochs. This ensures that the variance growth from the process will be at least $t_{min}w^2$, and the system will stay responsive to new measurements. This is more desirable than simply increasing w because longer time periods do not become overly sensitive. Teams may still be allowed to submit at intervals shorter than t_{min}, but the algorithm will internally adjust the time between submissions.

Figure 4.1 shows the effects of the minimum time period for a fictional submission scenario. The team switches from winning 50% of the time to winning 75% of the time against a player anchored at rank 0 and attempts to increase rank by submitting at 30 minute intervals. Without the minimum time setting, the rank takes over 30 additional submissions to rise to the new level. With the minimum time period in place, the rise time is reduced to 10 submissions (200 matches). Setting the parameters t_{min} and w involves balancing the response time with the level of noise. The final line in the figure shows the ranking history when match outcomes are determined by comparing two variables drawn from random exponential distributions with mean $e^{\bar{r}}$ and 1, which has the same probability distribution as the Bradley-Terry model [27]. Note that the model tends to track some of the short term noisy results, which is part of the tradeoff for higher sensitivity. Based on feedback to date users appear to favor the additional responsiveness over protecting against these shifts. For final deployment it is best to use actual match data either from the warmup phase or from previous competitions to set the values. The current standard on the Zero Robotics Leaderboard of $w = 0.15$ and $t_{min} = 1.0$ provided a good balance during the Alliance

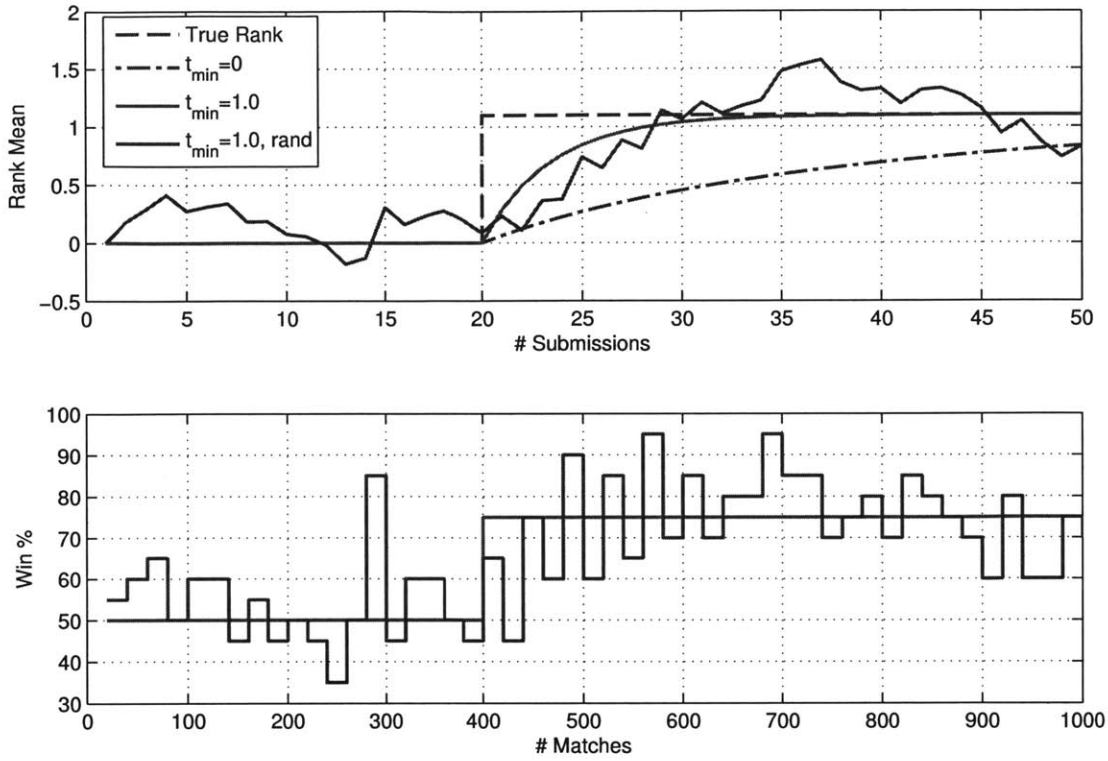


Figure 4.1: A team makes repeated submissions (20 matches each) at 30 minute intervals with a win performance of 50% against a rank 0 player. At the 10th epoch, the team switches to winning 75% of the time. With a virtual limit on the time between submissions, the rank responds much faster. The green profile shows the ideal step response, and the red profile shows the response when the match outcomes are drawn randomly with the win probability.

phase and should be the starting point for tuning future competitions. Another option for reducing sensitivity to noise is running more matches at each submission.

Figure 4.2 demonstrates the effect of the WHR algorithm improvements on the ranking history of a team in the 2012 3D competition. Setting a minimum time for the submission interval has the largest effect on the ranks, causing a much faster rise time during the period of increasing score at approximately 45 submissions. At approximately 100 submissions, grouping the identical submissions helps to prevent swings in the rank for repeated submissions of the same code.

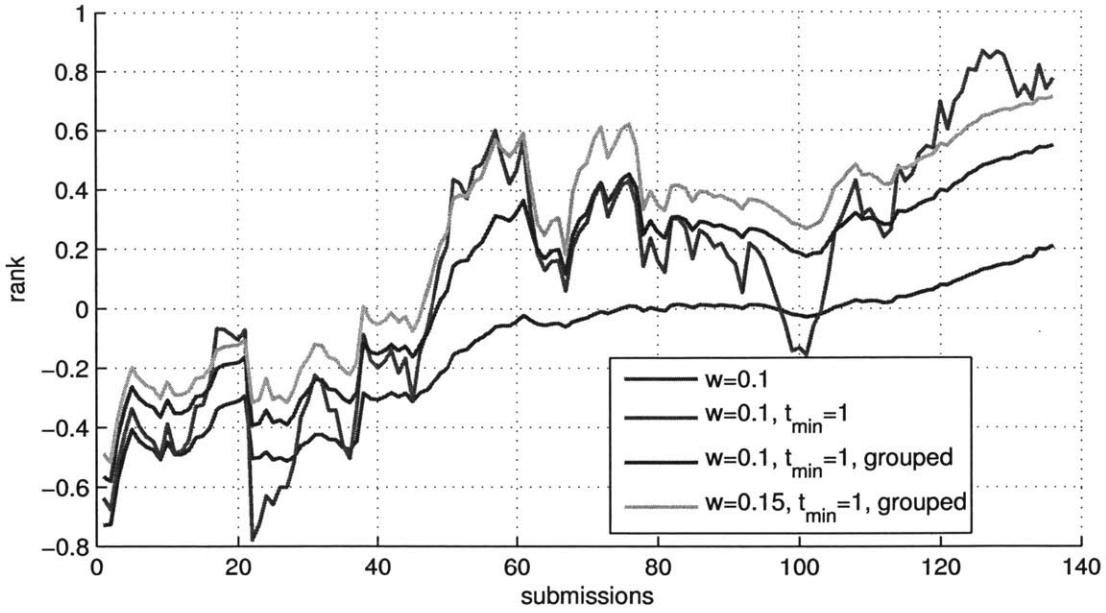


Figure 4.2: Improvements to the WHR algorithm applied incrementally to ranking data from the 2012 3D competition. Adding a minimum time interval allows the ranking system to respond much faster to new submissions, while grouping identical submissions prevents large swings for the same code revision.

4.3.2.7 Batch Optimizations

The backwards filtering effects of WHR are only beneficial if they have a chance to propagate to other teams in the ranking system. In the current implementation, each time a user makes a submission to the Leaderboard, the system runs a single algorithm update only for the 20 teams that were involved in the new matches. As other teams make submissions, the updated history will be used for any overlapping matches, and the information will slowly propagate to all teams on the Leaderboard. To speed up the propagation, it is important to periodically run batch optimizations through all submissions. One or two passes over the all teams can be triggered after a fixed number of matches, currently 100[3].

Batches with more iterations should be executed on a daily basis and at the end of the competition. An automated process can run nightly updates of 50 or more passes

[3]As an implementation caution, on days with significant activity, the individual passes may be triggered frequently enough to cause an excessive computational burden and should be disabled or raised to a largerr value.

through all teams to propagate information from the matches during the day. At the end of the competition, the batch typically involves many passes over the data set, as many as 500 to 2000. These passes are currently triggered and monitored manually. The stopping criteria for ending the optimization are:

1. The change in the overall likelihood computed as sum of log likelihoods from each of the rank histories.

2. The maximum change in any team's score.

When both of the items converge to small values, the optimization is stopped and the resulting scores are used as the final results of the competition.

4.3.3 Presentation to Users

In comparison to simple ad-hoc scoring systems, probabilistic ranking methods like WHR have a strong theoretical base, but the additional complexity introduces challenges in the way ranking results are conveyed to users. Without adequate information users can either project their own interpretations of how the system should work onto the results, or find the outcomes to be illogical, both of which lead to eventual frustration.

One approach is to hide most of the ranking information from the participants and display the teams in an ordered list. This is the strategy of some online computer games where complex ranking and matchmaking algorithms are constantly evaluating user performance with the objective of keeping the game interesting. Hiding the matchmaking calculations can also prevent users from trying to manipulate the system for better ranks. This is not in the spirit of an educational platform like Zero Robotics, where the objective is to provide participants with many tools to learn and improve performance. Therefore, it is very important to create an effective system for computing ranking results.

The approach for the Zero Robotics Leaderboard is to display three different views of the ranking results:

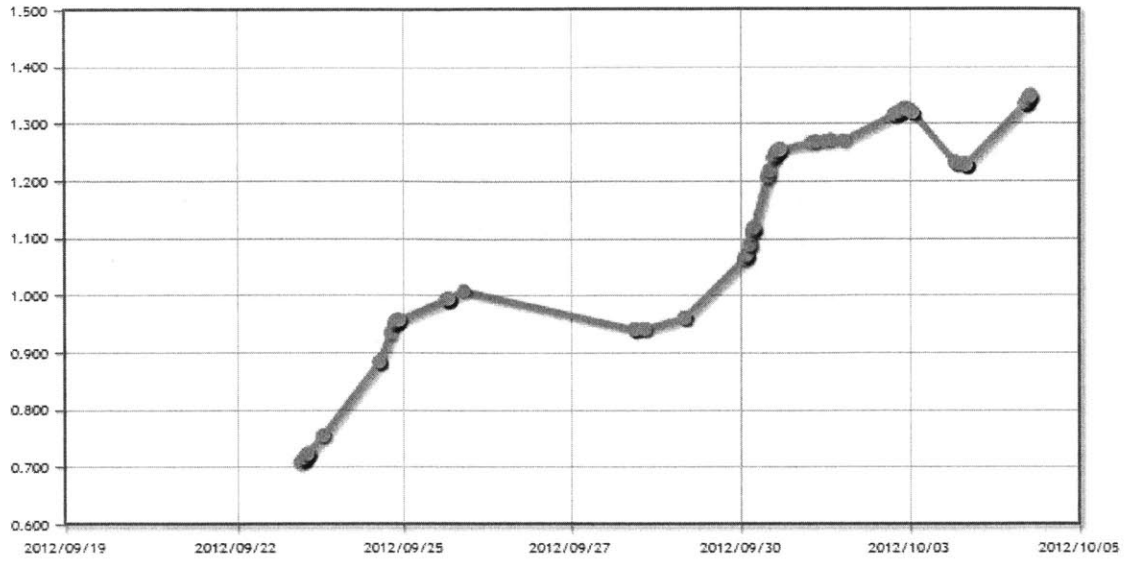


Figure 4.3: The Leaderboard's Match History view displays a line chart showing the *filtered* rank history of the player. Each point on the chart represents an epoch that can be clicked to see all match results for the submission.

1. The current order of all teams on the Leaderboard and their associated ranking score.

2. An interactive display showing the rank history for a specific team, including the matches played at each epoch.

3. A view of a selected epoch that graphically describes the quantities involved in computing the rank.

The first view is simply a list generated by ordering the teams by their scores. It is meant primarily as a brief summary of the current standings and as an index to access the remaining views. The second view, shown in Figure 4.3, allows users to browse the full history of ranking epochs and select specific epochs to see the associated matches. Every single match on the Leaderboard can be viewed in the 3D visualization for detailed review.

The final view, shown in Figure 4.4, conveys several of the components responsible for the the score at the currently selected epoch. The histogram bar chart shows binned wins and losses on a horizontal scale of rank to show how the team's current

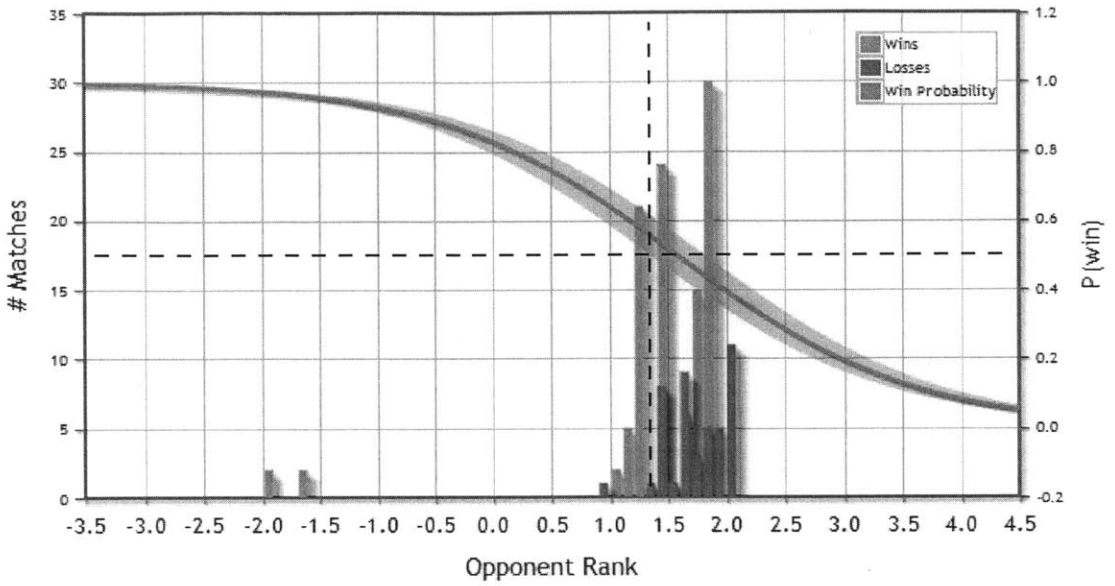


Figure 4.4: The Leaderboard's Histogram view shows the quantities involved in calculating a player's rank. The vertical bars are bins containing totaling wins and losses for the players, and the plotted line shows the win probability as a function of opponent rank. The shaded area is the 3σ uncertainty interval, and the cross-hairs shows the adjusted rank at the current epoch.

ranking is reflected by wins and losses weighted by relative rank. The plotted line is a trace of the team's win probability as a function of opponent rank. The shaded region shows the 3σ uncertainty bound on the rank, and the cross-hairs show the team's score as calculated by the variance penalty. In the example, we see that the 50% probability point shows roughly a dividing line between winning most matches to losing most matches.

4.3.4 Case Study: ZRHS 2012

4.3.4.1 Overview

The 2012 high school tournament was the first opportunity to test the WHR-based Leaderboard ranking system with an official tournament. The overall effectiveness of the Leaderboard system can be judged from several perspectives:

Engagement How does the scoring system affect involvement of the teams?

Computational Cost How does the scoring system compare to a single round robin tournament in the number of simulations required?

Prediction Accuracy Does the scoring system accurately predict the outcomes of matches?

User Experience and Programmatic Outcomes The scoring system has a major effect on how users interact with the tournament. How did the users respond to using the system? How does using the system affect the Zero Robotics program?

The Leaderboard was deployed for all phases of the competition. Each competition started with a 1-week warmup to give teams a chance to compete without affecting their final ranks, followed by the official competition. Teams were allowed unlimited submissions up to the final day of competition after which only 10 submissions were allowed. There were several differences between the phases to adjust the algorithm as the competition proceeded:

- For the 2D and 3D phases, submissions were scored independently (not grouped by code revision) with $w = 0.1$
- Between the 2D and 3D phase, the system switched from running against 10 opponents in both satellites to running against 20 opponents with even matches in both satellites. This helped teams encounter a broader range of opponent skills.
- At the Alliance phase, after analysis of results from the 2D and 3D phases, the additional modifications of grouping identical submissions, setting a minimum variance, and penalizing ties were appended. The prior standard deviation was also changed to $w = 0.15$.
- The 3D and Alliance phases conducted the warmup round in parallel with the official competition while the 2D phase did not launch officially until after the warmup ended.

4.3.4.2 Engagement

One of the motivations for building a leaderboard system is to enable teams to get an early start on solving the competition challenge and become more thoroughly engaged with the program as a whole. To measure engagement we can examine aggregate user simulation and submission activity. The first set of measurements are related to submission activity on the Leaderboard and are intended as benchmarks for future competitions. They cannot be directly compared to previous scoring systems because the other tournaments did not involve continuous submissions. The second set compares user simulation activity between seasons.

Figure 4.5 displays the percentage of teams that made at least n submissions during each of the 2012 competitions. In all three competitions a majority of the teams made had fewer than 12 submissions. This outcome is somewhat troublesome because even in the event that teams submitted on the last day, they were still allowed 10 submissions. Figure 4.6 gives additional detail about when teams first submitted projects to the system. In the best case, during the 3D competition, 50% of teams provided a submission at least 5 days ahead of the deadline. In the 2D competition, only 35% made a submission by the same time. These results are slightly more encouraging because they indicate that some teams may submit early and leave their player to be scored while privately improving the performance.

For comparison to the 2011 tournament, Figure 4.7 displays the number of private user simulations per day as a percentage of the total private simulations during the competition period for the 2011 and 2012 3D competitions. The plot indicates the relative distribution of simulation runs throughout the competition. Both years have very similar profiles with an extended period of lower activity followed by a week of higher intensity and a final spike at the submission deadline. Whether working with a Leaderboard or preparing for a single submission, teams appear to organize their activity in a similar manner. If the Leaderboard simulations are included in the comparison, the distribution is even more heavily weighted toward the final days. Figure 4.8 show a different perspective with the simulation counts normalized by

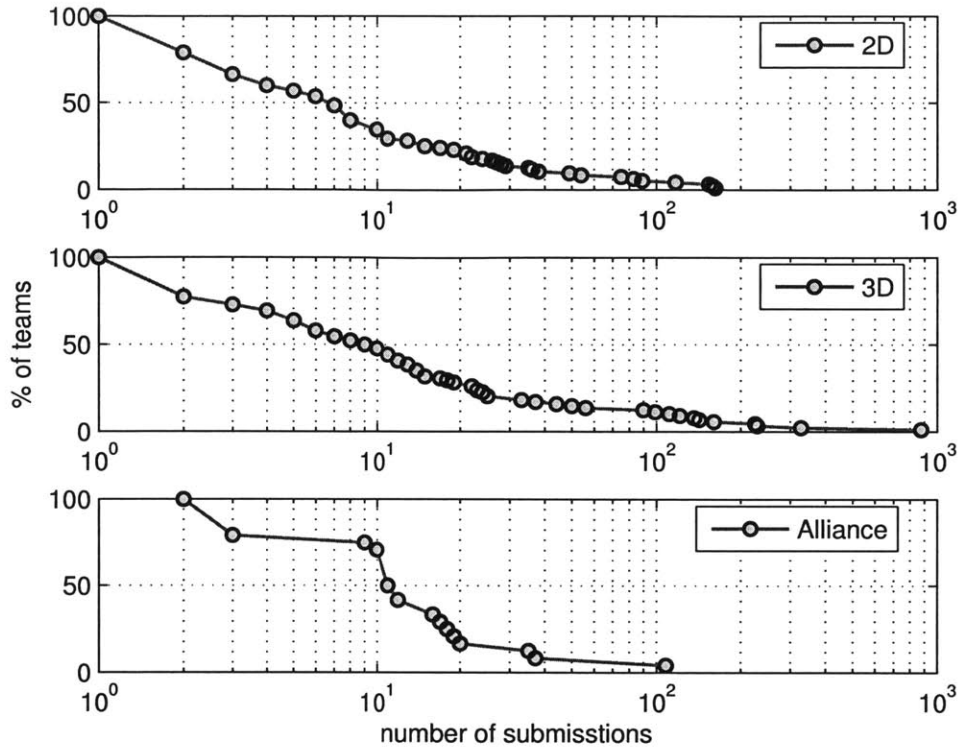


Figure 4.5: The cumulative percentage of teams having at least n submissions during the 2012 tournament. Markers indicate the number of submissions for at least one team. Approximately half of the teams made fewer than 10 submissions, while nearly all of the remaining half made fewer than 100.

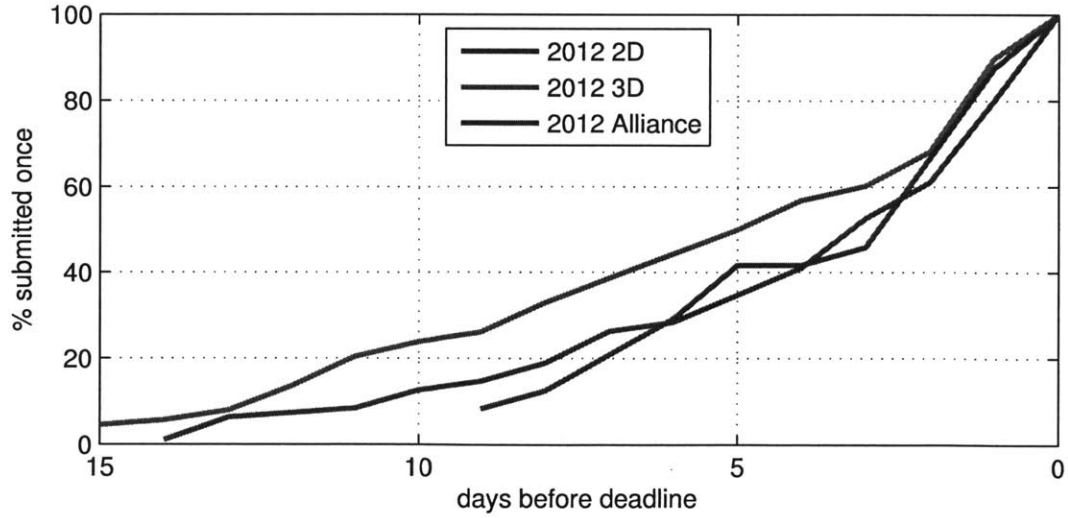


Figure 4.6: Percentage of teams submitting at least once vs. days before the submission deadline. At best, 50% of the teams made their first submission at least 5 days or more in advance of the final deadline.

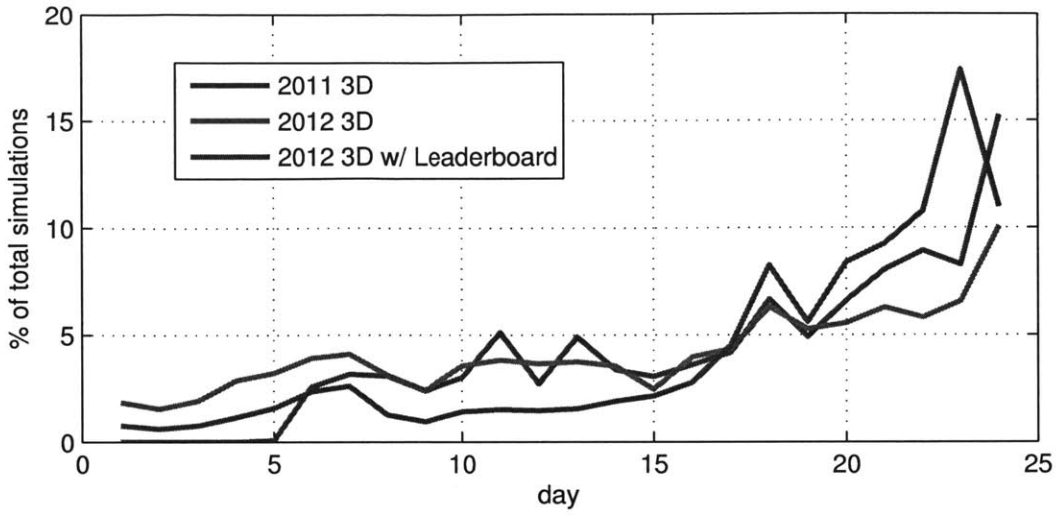


Figure 4.7: Percentage of total user simulations per day during the 3D competition in 2011 and 2012. Considering only simulations executed by the users, the trends for 2011 and 2012 are very similar with a heightened period of activity starting a week before the deadline, with a large spike near the end. Including Leaderboard simulations weights the distribution toward the final deadline due to the late submissions.

the number of teams with a final submission in the competition (88 for 2012, 91 for 2011). Here we see that the number of simulations per team was significantly higher throughout the competition without including the Leaderboard simulations.

Combined, the engagement results show some evidence of enhanced user participation, but additional seasons will likely be required before it is conclusive. The clearest indication is that teams tend to wait toward the end of the competition to make submissions. This may point to larger problems with engagement in the tournament and warrants significant attention. Given the similarities between the 2011 and 2012 3D competitions, the cause may be a natural team dynamic that can be adjusted with changes to the competition structure. A small point bonus for submitting early or even a requirement to make at least one submission before the final submission day may go a long way in encouraging teams to prepare earlier.

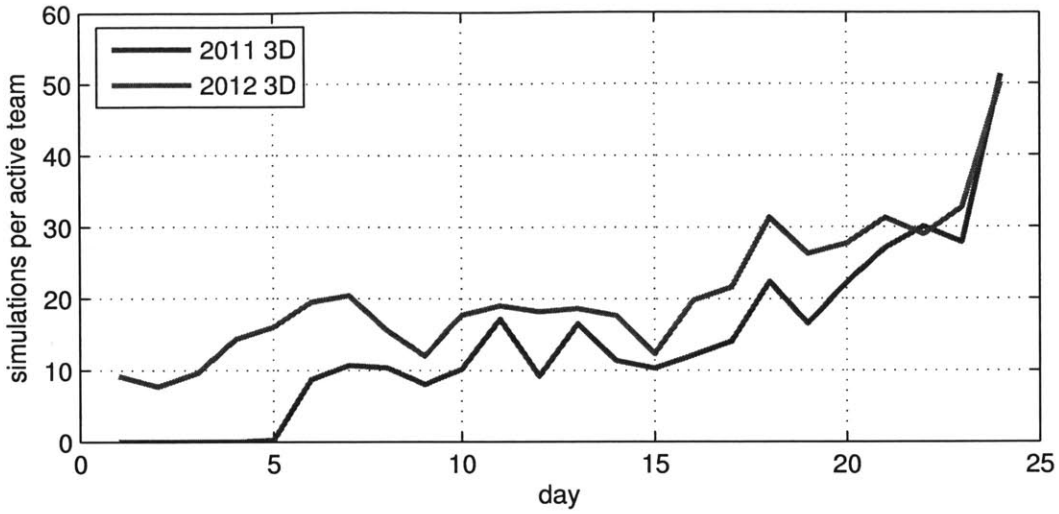


Figure 4.8: Number of simulations per active team. Normalizing the simulations by the total number of participating teams shows a more encouraging view of the user involvement. The simulation rate is consistently higher than in the previous year except for the deadline spike.

4.3.4.3 Computational Performance

For the purposes of evaluating computational performance, we will use the metric of number of simulations executed [4]. When executed in a single batch, the total number of simulations dictates the processing requirements Table 4.3.1 includes the total matches executed by the Leaderboard for all of the competitions. In comparison to a single round robin competition, which requires $\frac{n(n-1)}{2}$ matches, or 3828 for 88 teams, the Leaderboard system clearly involves many more simulations. Even running daily round robins over the course of the entire 24 day span of the 3D competition would have required only 14655 simulations assuming the same submission profile as Figure 4.6. This suggests it might be reasonable to add daily round robin batch simulations and apply the match results to the WHR algorithm with a relatively small overhead.

One qualifying consideration is the number of submissions made by individual teams. As shown in Figure 4.5, there were many teams with upwards of 100 submissions over the course of the tournament. In the 3D tournament, one team, made over

[4]Note that the WHR algorithm does require at least daily batch updates that iterate over the entire set of matches executed in a tournament. This update can take on the order of an hour but the time is still small relative to running thousands of matches.

800 submissions, accounting for 17,483 simulations–more than 25% of the total. Modest limits on the number of submissions per day would help reduce the total number of simulations required. The separate issue of addressing the apparent need to submit a large number of simulations led to several of the WHR algorithm modifications.

Ultimately, the main advantage of the WHR algorithm is the ability to provide nearly continuous feedback to the teams about their current performance. For the computational cost of running a single round robin simulation with 88 participants (3,828 simulations), all teams could make two unique submissions over the course of a day with 20 matches each and see results immediately. At 200 teams, one round robin competition of 19,900 matches would be the equivalent of 5 submissions for all teams. After the first submission, players stay on the Leaderboard, so additional matches become available as others submit. Even at a computational loss to full round-robins, the flexibility of being able to make updates to code and survey new strategies is important for *Efficient Inquiry.*

4.3.4.4 Prediction Accuracy

The Leaderboard rankings ultimately determine the winners of each competition phase, so it is important to have confidence that the ordering of teams produced by the system accurately reflects their performance. Coulom uses match prediction accuracy as a metric for comparing the WHR algorithm to other scoring methods, and the same method is applied here. To arrive at the prediction values in Table 4.3.1, the complete scoring history of the competitions was reproduced by processing submissions one at a time through the WHR algorithm, then running two full updates of all ranks in parallel. This roughly approximates the history of ranks during the competition, but it is slightly more accurate because a full batch update is performed at each iteration. Prior to processing the matches associated with the submission, the outcomes were predicted by the current rank of the player and the current rank of the opponent in the match. For a win probability $P > 50\%$, the match was predicted to be a win a loss otherwise. The total number of correct predictions is tallied in the table. Ties and failed simulations were not counted. The first prediction column is

Tournament	w	# Valid Matches	Ties	Correct by $\bar{r}$ (%)	Prediction by $\bar{r} - 3\sigma$ (%)
2D	0.1	26990	2770	15941 (59.1%)	15769 (58.4%)
3D	0.1	42235	15867	28523 (67.5%)	28215 (66.8%)
Alliance	0.15	4367	632	3298 (75.5%)	3230 (74.0%)

Table 4.3.1: Match Outcome Prediction Performance

Configuration	Correct by $\bar{r}$ (%)
$w = 0.1$	28523 (67.5%)
$w = 0.1$, grouped submissions	28511 (67.5%)
$w = 0.1$, grouped submissions, $t_{min} = 1$ day	29036 (68.7%)
$w = 0.15$, grouped submissions, $t_{min} = 1$ day	29203 (69.1%)
$w = 0.3$, grouped submissions, $t_{min} = 1$ day	29389 (69.6%)

Table 4.3.2: Effect of WHR Improvements on 3D Competition (42235 Matches)

the match prediction percentage based on the team's rank mean, $\bar{r}$, while the second column uses the adjusted score based on the rank mean and 3σ confidence interval.

Prediction performance increased as the tournament progressed. The prediction accuracy is coupled with the matchmaking system, and in the current matchmaking system matches are intentionally picked where the means are close together and the match outcome is uncertain. Between the 2D and 3D competition, switching to running against 20 opponents instead of 10 likely led to more encounters between teams with larger differences in rank and therefore higher certainty in the predictions. The equivalent increase in prediction accuracy between 3D and Alliance phases is less clear, especially with the significantly smaller sample size. Some of the increases may be attributed to the WHR algorithm improvements. Table 4.3.2 shows the incremental effects of applying grouped submissions, a minimum interval between epochs, and an increased variance on the prediction accuracy by re-processing the 3D tournament with each successive change. While the modifications account for a small change in prediction accuracy, they are not as large as the overall gains between competitions. Quite to the contrary it is significant to note that the changes have a relatively limited impact on prediction accuracy while having a more significant effect on the user experience.

Predictions based on the 3σ adjusted ranks were very close to predictions based

on the means, but they performed slightly worse in all cases. This outcome on its own does not impact the decision to use variance in the scoring system, but it should be weighed along with other factors in the user experience.

Lastly, it should be noted that the prediction accuracies were significantly higher than those reported in [15], which were at best 55.8% when applied to 2,331,757 games of Go. The disparity may be related to the system being used to rank (mostly) human players and the significantly larger number of matches.

4.3.4.5 Effect of Tie Penalty

In the final Alliance phase of the competition, the tie penalty system was put in place to discourage teams from intentionally tying matches to boost win percentage. The effort reduced ties to some extent, but they remained a higher percentage (14.5%) of the overall matches than in the 2D competition (10.3%). Not all ties were intentional; sometimes teams failed to detect conflicts picking up items and ran out of fuel repeatedly triggering collision avoidance maneuvers. Since most of the 24 competing alliances experienced ties, or competed against other players that had many ties, the tie penalty mainly drove all scores down toward the low rank of -4 set for the penalty score. The highest mean rank on the Leaderboard was -3.10 in comparison to 2.88 in the 3D competition. Due to the relative nature of the scoring system, the absolute scale is not as important, but it is clear the absolute score applied to ties re-centers the scale around the lower value. These results suggest that while the penalty may have been effective in lowering scores of teams that tied matches without causing the ranking algorithm to destabilize, it may be better apply penalties outside of the WHR system and true ties should be avoided until they can be explicitly incorporated into the probabilistic framework.

4.3.4.6 User Experience

After the 2D and 3D phases of the tournament were completed, it became clear from user feedback and analysis of ranking results that there were two main problems with the basic WHR implementation:

1. Repeated submissions were sometimes required to see a significant change in rank.

2. The scoring system was complex and difficult to understand.

Repeated Submissions During the 2D and to a much greater extent during the 3D phase, many teams attempted to optimize their scores by increasing their overall win percentages. The primary approach to doing so was to win against as many opponents as possible, then force a tie in the remaining matches. Since ties were not counted in the the 2D and 3D phases, the win percentages were higher, but only based on a small number of matches. Scores were observed to move slowly, leading to teams repeatedly submitting projects. To better understand how the number of matches and win percentage affect the ranking algorithm, we will consider a simplified example case. Suppose the user makes a new submission following heavy submission period where the previous epoch had a large number of example matches. We will assume for simplicity:

- the initial rank is $r_0 = 0$;

- due to the high number of previous submissions, the initial rank changes very little, $\Delta r_0 \approx 0$; and

- all matches in the new submission are against players of the same rank r_i.

For the epoch at $t = t_1$, over a range of n matches in the set M, using $\gamma_1 = e^{r_1}$ as the rank being optimized, $\gamma_i = e^{r_i}$ as the opponent's (static) rank, and $|W|$ the number of wins in M, the gradient for the match outcomes, including the prior form t_0 to t_1 is

$$\frac{\partial \ln P}{\partial r_1} = |W| - \gamma_1 \sum_{i \in M} \frac{1}{\gamma_1 + \gamma_i} - \frac{r_1 - r_0}{\sigma^2}. \tag{4.7}$$

The term $\sigma^2 = |t_1 - t_0| \, w^2$ is shorthand for the variance of the prior. Applying the simplifying assumptions $r_0 = 0$ and constant opponent rank, and noting the optimum

rank is found when the gradient $\frac{\partial \ln P}{\partial r_1}$ is 0, the new rank can be calculated by solving

$$|W| - n\frac{\gamma_1}{\gamma_1 + \gamma_i} - \frac{r_1}{\sigma^2} = 0. \tag{4.8}$$

Dividing by n, gives an interpretation based on the win percentage

$$win\% - \frac{\gamma_1}{\gamma_1 + \gamma_i} - \frac{r_1}{n\sigma^2} = 0. \tag{4.9}$$

To further simplify, we can assume that the opponent has a rank of $r_i = 0$, or $\gamma_i = 1$, then perform a Taylor series expansion about the initial guess (from the previous epoch) of $r_1 = 0$, then solve for Δr_1 to find the new rank

$$\begin{aligned} win\% - \frac{\gamma_1}{\gamma_1 + 1} - \frac{r_1}{n\sigma^2} &= 0 & (4.10)\\ win\% - \left(\frac{1}{2} + \frac{1}{4}\Delta r_1\right) - \frac{1}{n\sigma^2}\Delta r_1 &= 0 & (4.11)\\ \Delta r_1 &\approx \frac{1}{\frac{1}{n\sigma^2} + \frac{1}{4}}\left(win\% - \frac{1}{2}\right). & (4.12) \end{aligned}$$

From the linearized view in Equation 4.12, we see there are several contributing factors to changing the rank. First is the difference between the win percentage and the expected win probability. As intended, the rank increases when the team outperforms expectations. The second factor is more significant. Figure shows a plot of the leading term as a function of the number of matches for a fixed time period of 1 day and $w = 0.15$. For a fixed win percentage, we see the influence of winning increases nearly linearly from 1 to 20 scored matches. From this information, we can conclude for the simplified case that forcing a tie to increase win percentage causes the rank to change more slowly. Though the situation is more complex in practice with varied opponents and the full ranking history, the results agree observations from the tournament. In many ways the effect is desirable as it conveniently supplies one of the benefits of the variance measure in the TrueSkill ranking system. Teams can improve their scores either by playing more often (making more submissions) or by improving consistency (higher win percentage). With this in mind, the main conclusion is that

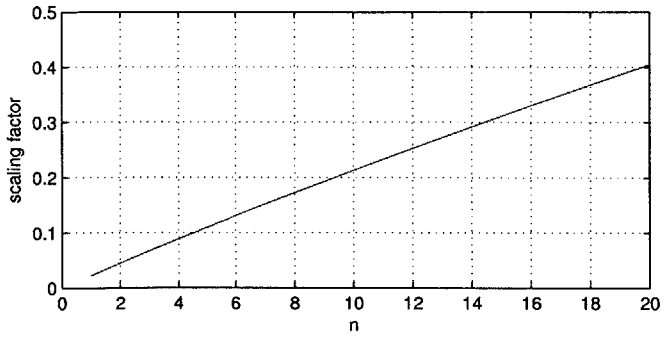


Figure 4.9: A plot of the leading term from Equation 4.12 for $n = 1$ to 20. The factor scales the effectiveness of the win percentage based on the number of matches.

the system should avoid discarding matches to remain optimally responsive.

Another factor that affects the speed of change is the relative rank of the players. If teams mostly won against lower ranked players, the scores only increased slightly because the match outcomes were expected. For some teams, seeing an incremental increase in score at each submission prompted them to repeatedly submit the same project until reaching a steady state, while others saw the slowly changing scores as a problem with the scoring system.

The final issue was due to the evolution of uncertainty between matches. The 2D and 3D phases did not set a virtual minimum time between submissions, so repeated submissions were considered to be in nearly the same epoch. Even if the code was modified during the process, the system rapidly became insensitive to new match outcomes. This can also be seen from the simplified model in Equation 4.12. If the time step between submissions is dropped form 1 day to 5 minutes, the leading term becomes 250 times smaller. This clearly motivates the minimum time step modification.

These responsiveness issues were partially addressed through updates to the scoring system and better presentation tools added between the 3D and Alliance competitions. The grouped submission and minimum time enhancements from 4.3.2 were developed to improve responsiveness when making closely spaced submissions. The time deviation was also increased slightly from $w = 0.1$ to $w = 0.15$ to allow more variation in the scores. The histogram tool was added to give more detailed informa-

tion about performance at each submission instead of the raw win/loss/tie statistics that teams were heavily focused on. There were many fewer complaints about the scoring system in the Alliance phase, but the number of participating teams was much smaller. The enhancements will be best verified during the first competition of the next tournament.

Scoring Complexity The second issue relating to complexity of the scoring system is mostly anecdotal based on feedback. Several teams noted that it was difficult to discern a relationship between match outcomes and changes in score, especially how match outcomes affected uncertainty. Beyond displaying the 3σ confidence interval on the histogram plot, a concise representation of the factors affecting uncertainty remains a challenge that should be addressed for future tournaments.

Another source of confusion is the difference between the filtered rank history shown on the Leaderboard and the ranks displayed as the team makes submissions. The full rank history is always displayed on the website, but it can change with each submission as the full history is updated. This means that the teams observe one sequence of ranks as they make submissions and see a different history shown on the chart. A better data display might incorporate both views. This would require explicit storage of the "experienced" rank history because the ranks are re-computed at each update. One potential problem with this approach would be a large jump in the rankings at the end of each day and the end of the tournament when the system runs batch updates.

Finally, some teams conflated the significance of win percentage with performance on the Leaderboard. Since the WHR algorithm takes into account both the win-loss outcome of a match and the likelihood of that outcome, a higher win percentage does not always translate to a higher score, especially if the win percentage is inflated by unscored ties as in the 3D competition. The issue was partly exacerbated by the way ranking epochs reports were initially summarized with a display of win/loss/tie percentage. The histogram tool created at the end of the 3D competition was designed to emphasize the importance of an opponent's rank on the score.

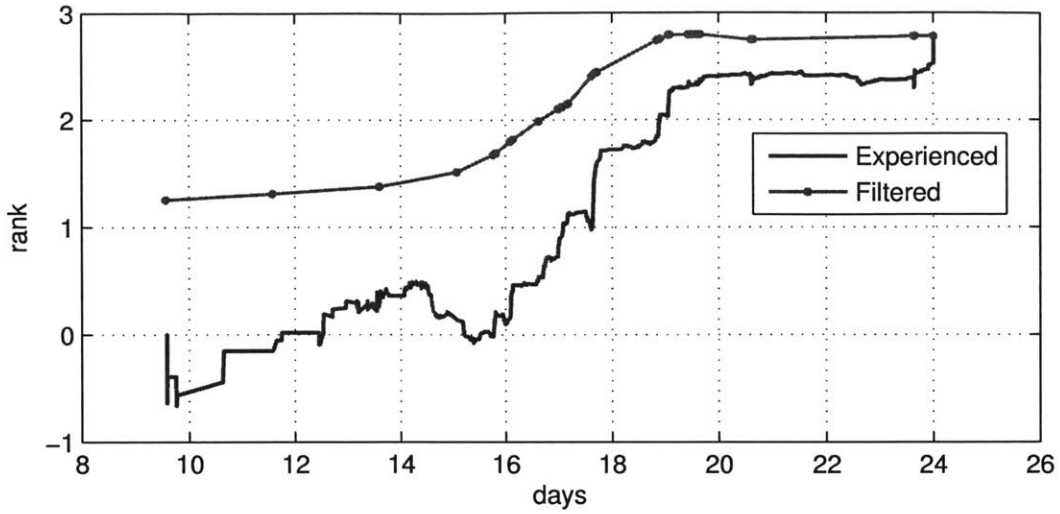


Figure 4.10: The match history presented to users as shown in Figure 4.3 is the a filtered version of the ranking history based on all match outcomes to date. This plot shows a comparison between what a user would see if they plotted the observed mean ($\bar{r}$) scores throughout the competition and the filtered rank history. The large jump at the end of the competition is due to adjustments in the final batch optimization.

4.3.4.7 Programmatic Outcomes

From the perspective of running a competition, the Leaderboard was a significant improvement over formats in previous seasons. With many teams actively engaged in creating submissions at an early point in the season, important bugs in the initial game deployments were discovered and corrected well before the submission deadline. The Leaderboard alone is not a replacement for thorough testing, but it is a useful guard in the event that problems to do arise. More importantly, teams have a chance to react to any changes that are released to fix the bugs. This prevents difficult situations such as those experienced in the 2011 round-robin tournaments where bugs were discovered when running the final batch competitions.

A live scoring system also provides continuous snapshots of user participation throughout the competition period. Team performance can be easily monitored by watching the public matches, and the number of active submitters can give the tournament organizers a sense for how actively teams are participating. With this information additional modifications to the game can be proposed for future rounds or

e-mails can be distributed to the teams with advice or encouragement.

A specific benefit of the WHR algorithm is the reconstruction of the full ranking history at every algorithm update. If there is a bug in the ranking code or the developers wish to make adjustments to the parameters, it is possible to trigger a re-computation of the ranks and let the algorithm converge to the new ranking results. The match outcomes do not have to be re-played through the algorithm because the system already incorporates all match events in each update.

4.3.5 Recommendations for Future WHR Competitions

Based on the results of the 2012 competition, the following changes are recommended for future seasons:

- Add a daily batch simulation during off hours consisting of a fixed number of matches per team. The batch does not have to be a complete round-robin tournament, but during slack periods additional matches can help the WHR algorithm converge to the players true rank. If teams know that the system will automatically run additional matches, they can focus on improving submissions instead of making many submissions.

- Set a minimum time between submissions to the Leaderboard. In addition to setting the virtual epoch spacing t_{min}, a real limit should be placed on how many submissions a team can make within a specified time interval. Setting a spacing limit will encourage teams to maximize performance for each submission instead of repeatedly submitting the same projects. Spacing out the submissions will also reduce load during high volume periods at the end of the competition. An alternative limit might be to give teams a fixed number of submissions per day. When implementing this change it will be important that the teams have high confidence that the scoring system is responsive enough to significant changes in performance that reflect code improvements. The additional batch simulations will help to improve this aspect.

- Avoid using the tie penalty system. Until ties are formally modeled by the ranking system, the method of penalizing ties remains an ad-hoc modification of the algorithm that may be prone to problems in large competitions. Games should be exclusively win-loss with no possibility for ties. The game should be carefully tested to ensure that tie breakers do not result in significant shifts of priority. See the lessons from HelioSPHERES in Section 2.4.3 for an example.

- Remove the variance penalty from the scoring system unless there is a clearer way to calculate and convey it. As shown by the prediction performance numbers in Table 4.3.1, the adjusted score with a penalty for variance does not improve ranking predictions, and in most cases performs slightly worse. The additional confusion it added to the 2012 tournament suggests that the connection with consistency is either not being conveyed well or is not being applied properly. Part of the problem is the relationship between time and the rank uncertainty. If a team waits for several days to make a submission, then only submits a single time, the variance will jump up briefly, resulting in a decrease in rank. Until both issues can be studied more thoroughly, using the rank means should simplify the interpretation of the ranking results.

- Improve visualizations and documentation to better explain ranking algorithm. The confusion between win percentage and ranking performance indicates a need for the users to better understand how the ranking system works. A very powerful update would be a tool to allow teams to perform "what if" scenarios by changing the match outcomes and observing the change in rank.

4.4 Summary

Several iterations of competition scoring systems have lead to the design and implementation of a scoring framework that will be robust to several seasons of growth. Experience from the 2010 and 2011 seasons with all-to-all round-robin competitions has shown that relying on a single one-shot tournament at the end of a competition

risks last minute bug surprises, and more importantly gives competitors only a single change to prove their skill. A second class of scoring systems with continuous leaderboard scoring has enabled teams to enter submissions throughout the competition at the cost of a higher computational cost spread throughout the season.

The most recent algorithm for the Leaderboard is an adaptation of the Whole History Rating system modified with improvements to integrate with the Zero Robotics platform. The stability improvements added by an Armijo iteration in the Newton solver have allowed the system to function with few problems over the course of nearly 100,000 matches. Additional modifications adding grouping of submissions and a virtual minimum time interval have helped to increase the responsiveness of the system to new match submissions.

It remains unclear if the objective of increasing active participation in the competition has been improved by the addition of a real time scoring system. Up to half of users made submissions 5 or more days in advance of competition deadlines, while others still waited until the final day to make submissions. 77% of teams made more than one submission in both 2D and 3D competitions, showing that teams are at benefiting from the ability to make more than one submission. Total simulations per active team were significantly higher than the 2011 tournament, though the overall distribution of simulation runs throughout the tournament remained the same. Future competitions with the same system will help to support these initial hopeful signs.

Chapter 5

Close-Proximity Collision Avoidance for Satellite Game Players

5.1 Introduction

This chapter covers the details of a collision avoidance algorithm originally implemented for close proximity formation flight. It has been used throughout the Zero Robotics program to prevent collisions between satellites in the virtual simulation environment and on the ISS.

Collision avoidance for satellites has been covered from several perspectives with a strong emphasis on planning and mathematical programming methods. Pre-planning a trajectory for collision avoidance is desirable because it saves propellant, but it often comes at the price of steep computational requirements. Richards, Breger, and How have demonstrated several variants of Mixed-Integer Linear Program formulations for solving trajectory problems that include collision avoidance constraints[8, 61]. Mathematical programming carries a high computational burden, incurring a large penalty for re-planning maneuvers or even requiring trajectories to be planned in advance.

For online planning with obstacle avoidance, incremental methods such as Lavalle's Rapidly-exploring Random Trees (RRT)[42] have been extended for moving obstacles by Hsu, Latombe, and others[34]. Randomized planning can quickly discover feasible,

collision-free trajectories at a significantly lower computational cost than mathematical programming methods but still presents a challenge to implement on a tightly constrained satellite system.

Furthermore, pre-planning trajectories may not be possible depending on the satellite mission. A close-proximity inspector satellite might be directed by a human operator but still require autonomous avoidance capabilities. For these applications, reactive controllers that adjust trajectories in real-time to avoid collisions are desirable. Highly computationally efficient algorithms can be formulated in the form of "steering behaviors" which command an acceleration to change the direction of the velocity vector. In [60], the authors predict the closest point of approach (CPA) of video game characters and steer the trajectories away from the potential collision. More recently, in [25] a similar approach has been generalized and applied to assist pilots in steering aircraft away from potential collisions.

The following sections present a formulation of the CPA steering behavior applied to close proximity satellite operations. It is both compact and computationally efficient and has been demonstrated successfully on SPHERES. Section 5.2 presents the dynamic model and introduces the collision avoidance controller. Section 5.3 covers important implementation considerations when applying the controller, and Section 5.4 presents experimental data from tests aboard ISS.

5.2 Steering Law

5.2.1 Relative Kinematics

For the entirety of the discussion we will assume the relative satellite motion is over small enough distances, time scales, and relative velocities that the effects of relative orbital dynamics can be neglected. The satellite dynamics are therefore of the form

$$\ddot{\mathbf{x}} = \mathbf{f}/m$$

where the external forces, $\mathbf{f}$, are provided by a thruster propulsion system. This has been a reasonable assumption for SPHERES operations aboard the ISS and double integrator dynamics have been shown to be a reasonable approximation in other close-proximity satellite studies[10].

The steering law operates on the closest point of approach (CPA), defined as the point in space and time in a relative trajectory when two objects are closest. For the time being we will consider the relative motion of two satellites. To predict the CPA, starting at time $t = t_0$, the motion of satellites 1 and 2 is assumed to continue along the current velocity direction.

$$\mathbf{x}_1(t) = \mathbf{x}_1(t_0) + \dot{\mathbf{x}}_1(t_0)t \tag{5.1}$$

$$\mathbf{x}_2(t) = \mathbf{x}_2(t_0) + \dot{\mathbf{x}}_2(t_0)t \tag{5.2}$$

$$\mathbf{r}_{12}(t) = \mathbf{x}_2(t) - \mathbf{x}_1(t) \tag{5.3}$$

$$\mathbf{u}_{12}(t) = \dot{\mathbf{x}}_2(t) - \dot{\mathbf{x}}_1(t) \tag{5.4}$$

Defining the relative position from satellite 1 to satellite 2 as $\mathbf{r}_{12}$ and relative velocity as $\mathbf{u}_{12}$, the time evolution of the relative position is

$$\mathbf{r}_{12}(t) = \mathbf{r}_{12}(t_0) + \mathbf{u}_{12}(t_0)t. \tag{5.5}$$

For clarity, the time index and subscripts will be omitted from this point forward. All values can be assumed to be from the perspective of satellite 1 at $t = t_0$ unless otherwise specified.

Taking the squared magnitude of the relative position and minimizing with respect to time gives the time at closest point of approach t_{CPA}.

$$d^2 = \mathbf{r}(t_{CPA})^T\mathbf{r}(t_{CPA}) \tag{5.6}$$

$$\frac{d}{dt}d^2 = 2(\mathbf{r}^T\mathbf{u}) + 2t_{CPA}(\mathbf{u}^T\mathbf{u}) = 0 \tag{5.7}$$

$$t_{CPA} = -\frac{\mathbf{r}^T\mathbf{u}}{\mathbf{u}^T\mathbf{u}} \tag{5.8}$$

The expression for t_{CPA} reveals several important characteristics about the point of closest approach. First, at $t_{CPA} = 0$, the relative velocity is perpendicular to relative position. This is intuitive because if there is no velocity in the direction of the relative position, then the two objects cannot get any closer. Second, it is possible for t_{CPA} to be negative if $\mathbf{r}_1(t_0)^T\mathbf{u}_1(t_0) > 0$, meaning the CPA has already occurred and the paths are diverging. When implementing the controller, it is important not to trigger the avoidance maneuver for potential collisions in the past.

The distance at closest point of approach, d_{CPA}, can be calculated by evaluating Equation5.6 at t_{CPA} and taking the square root.

$$\begin{aligned} d_{CPA} &= \sqrt{\mathbf{r}(t_{CPA})^T\mathbf{r}(t_{CPA})} \\ &= \sqrt{\mathbf{r}^T\mathbf{r} + (\mathbf{r}^T\mathbf{u})t_{CPA}} \end{aligned} \tag{5.9}$$

Potential collisions can be identified by examining the pair (d_{CPA}, t_{CPA}). If $t_{CPA} > 0$ and $d_{CPA} < d_a$, where d_a is a critical distance threshold, then the avoidance controller should be activated to avoid the collision. Note that for trajectories where the relative position and the relative velocity are exactly aligned

$$\begin{aligned} (\mathbf{r}^T\mathbf{u})t_{CPA} &= -\mathbf{r}^T\mathbf{r} \\ d_{CPA} &= 0. \end{aligned} \tag{5.10}$$

5.2.2 Avoidance Controller

The collision avoidance controller steers a pair of satellites away from a potential collision by commanding a change in velocity that increases the magnitude of d_{CPA}. To minimize the required velocity correction, the thrust is directed along the gradient

of $d_{CPA}$with respect to the satellite's current velocity, $\dot{\mathbf{x}}_1$.

$$\frac{\partial t_{CPA}}{\partial \mathbf{u}} = \frac{1}{(\mathbf{u}^T\mathbf{u})^2}\left(2(\mathbf{r}^T\mathbf{u})\mathbf{u}^T - (\mathbf{u}^T\mathbf{u})\mathbf{r}^T\right) \tag{5.11}$$

$$\begin{aligned}\frac{\partial d_{CPA}}{\partial \dot{\mathbf{x}}_1} &= \frac{\partial}{\partial \mathbf{u}}\sqrt{\mathbf{r}^T\mathbf{r} + (\mathbf{r}^T\mathbf{u})t_{CPA}}\frac{\partial \mathbf{u}}{\partial \dot{\mathbf{x}}_1} \\ &= -\frac{1}{2d_{CPA}}\left(t_{CPA}\mathbf{r}^T + \mathbf{r}^T\mathbf{u}\frac{\partial t_{CPA}}{\partial \mathbf{u}}\right)\end{aligned} \tag{5.12}$$

The gradient of d_{CPA} can also be used in a linear approximation to select the thrust magnitude. Assuming that the satellite provides an impulsive change in velocity along $\mathbf{g}^T = \frac{\partial d_{CPA}}{\partial \dot{\mathbf{x}}_1}$ with some magnitude, k, the approximation for d_{CPA} is given by Equation 5.13. After specifying the desired d_{CPA} target, d_t, the resulting thrust magnitude is calculated from Equation 5.14.

$$d_{CPA} = d_{CPA,0} + \mathbf{g}^T\frac{\mathbf{g}}{\|\mathbf{g}\|}k \tag{5.13}$$

$$k = \frac{d_{CPA} - d_t}{\|\mathbf{g}\|} \tag{5.14}$$

The avoidance controller in this form has the attractive property that the satellites need only to synchronize their relative state information to perform consistent avoidance. Using the definitions of relative positions and velocities in Equation 5.3 and Equation 5.4 ensures that the gradients and therefore the thrust directions will have opposite signs for both satellites.

There is a degenerate case for the condition in Equation 5.10 when the relative velocity and relative position are exactly aligned. In this situation any thrust perpendicular to the relative velocity will increase d_{CPA}, but there is no guarantee of synchronizing the direction of thrust. In practice, there is always at least an infinitesimal separation between the points of closest approach, and the controller will widen it.

5.3 Implementation Considerations

In proximity operations with multiple satellites, it is expected that a nominal controller receiving trajectory targets will provide most of the maneuvering commands. The avoidance controller can be used to override this controller when an imminent collision is detected. In this way, the upper levels of the control system do not need to plan explicitly for collision avoidance maneuvers, though adding a notion of avoidance will improve fuel usage. The following considerations will help improve the performance of the basic controller from Section 5.2.

5.3.1 Distance Threshold and Time Horizon

The avoidance controller is activated by checking for the condition $d_{CPA} < d_a$ and $0 \leq t_{CPA} < t_a$, where d_a and t_a are the distance and time horizon thresholds respectively. d_a should be selected to be the minimum allowable miss distance with a buffer for estimation estimation error. t_a depends on the ratio of relative velocity to available control authority as well as d_a. If the satellite is moving slowly and can easily change velocity, then the time horizon can be shortened. Likewise if safety concerns dictate large safety zones for avoidance, the time horizon should be lengthened to provide sufficient time to reach the target d_{CPA}. Also, if the maneuvers involve frequently changing directions, the time horizon should be shortened to prevent avoidance of unlikely collisions.

5.3.2 Distance Target

When considering cooperative and uncooperative avoidance scenarios, there is a distinction to be made between the avoidance threshold d_a and the controller distance target d_t from Equation5.14. In a cooperative avoidance situation, the satellites will share the effort of increasing d_{CPA}, so it possible to reduce the distance target. A rough approximation is half the avoidance trigger distance with a scaling parameter

k_m for margin.

$$d_{t,coop} = k_m \frac{d_a}{2} \tag{5.15}$$

$$k_m \geq 1 \tag{5.16}$$

For uncooperative avoidance, the distance target should be set to at least the avoidance distance threshold.

$$d_{t,uncoop} = k_m d_a \tag{5.17}$$

In both cases, the trigger distance, d_a, is unchanged because it indicates when a potential collision is detected, not how to react to the information.

5.3.3 Nominal Controller Override

A nominal controller that is paired with the collision avoidance controller may have no knowledge of an avoidance event and attempt to drive the satellite back toward toward a collision course. This can be wasteful for fuel usage if the two controllers fight back and forth until the threat is over, and in the worst case, the nominal controller can force a collision to occur. One strategy is to disable the nominal controller for a brief time period, letting the avoidance controller complete its change in direction then coast to maintain the straight line trajectory assumptions. Taking the idea further, if the nominal controller is disabled until $t_{CPA} = 0$, the avoidance controller will first correct the path then coast until the collision threat is over.

Depending on the application it might not be acceptable to override the nominal controller for long periods of time to avoid collisions. In practice, a good approach is to set another critical time horizon, $t_c < t_a$, such that if $t_{CPA} < t_c$, the nominal controller is disabled and the satellite coasts maintaining the straight line assumption.

5.3.4 Multiple Satellites

The avoidance controller, as formulated in Section 5.2, only considers two satellites. In [60], multiple agents avoid collisions using pairwise checks, prioritized by a calculation

of the most imminent threat. The approach works well for tens of 2D game characters in confined spaces as well as simulations of large flocks of birds. Similarly, the satellite avoidance controller can use pairwise checks between each of the satellites in the group. Avoidance maneuvers are executed one at a time, prioritized by the smallest value of t_{CPA}. As shown in the ISS tests below, this approach works well in practice for a three-satellite formation, and simulations with up to 10 satellites with closely spaced avoidance events have been performed successfully. Further study is required to provide a theoretical basis for the pairwise interactions or explicitly consider multiple satellites.

5.4 Initial Development ISS Test Session Results

The collision avoidance controller was implemented and tested in several microgravity sessions on the SPHERES testbed aboard the International Space Station. SPHERES was developed to test advanced autonomy and formation flight algorithms for satellites in a representative microgravity environment. The facility consists of three nanosatellites flying inside the station with on-off CO_2 thruster propulsion, a 6DOF pseudo-gps estimation system based on ultrasound, and onboard processing and telemetry systems[62, 56]. In all tests the nominal controller was a PD setpoint controller operating at 1 Hz paired with the low level collision avoidance controller.

The first test in September, 2008 verified the the basic controller presented in Section 5.2 applied to a 3-satellite collision avoidance problem. In this test, each satellite started on the vertices of an equilateral triangle, then simultaneously attempted to cross through the center of the triangle as shown in Figure 5.1. Figure 5.2 shows d_{CPA} and t_{CPA} as a function of time during a representative avoidance event. At $t = 150$, the satellite activates the collision avoidance maneuver as shown by a vertical line in the d_{CPA} plot. This was the first time the CPA distance was below the avoidance trigger distance of 26 cm at the beginning of a control cycle. After each avoidance event, t_{CPA} times show a significant decrease because one way the controller attempts to maximize d_{CPA} is to move t_{CPA} to the current time. Likewise, after each event

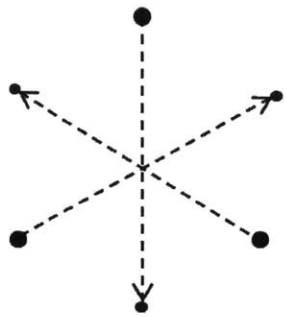
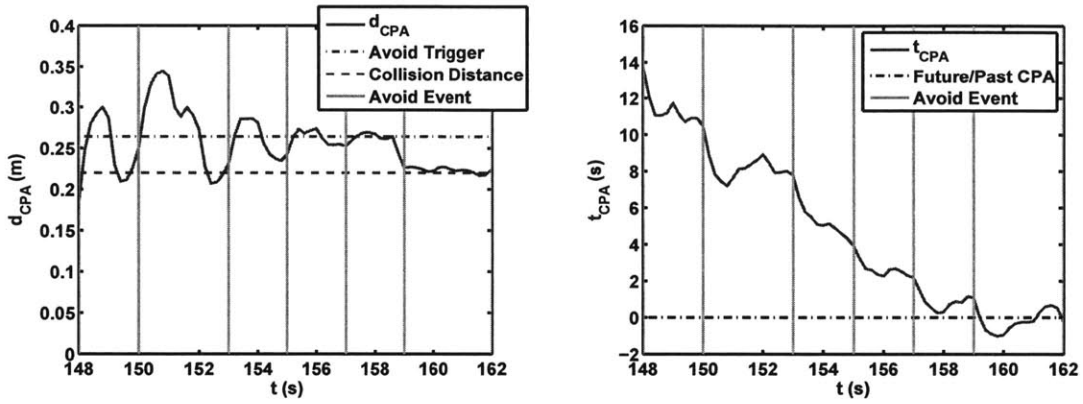


Figure 5.1: In the 3-satellite collision avoidance test, the satellites start at the vertices of an equilateral triangle and attempt to cross through the center simultaneously.



(a) (b)

Figure 5.2: The distance at closest approach (a) and time of closest approach (b) during a collision avoidance maneuver. Each time the collision avoidance controller is activated, d_{CPA} is moved above the avoidance trigger threshold, while the nominal controller tends to move the trajectory back toward collision. Each avoidance event also drives t_{CPA} closer to the present time.

d_{CPA} distances increase. It is also interesting to note that near the end of this event, the avoidance maneuver was activated every other control cycle. When t_{CPA} finally reaches zero, there is a slight grazing collision. Here we see that the PD controller kept driving the satellite back toward a collision on every other cycle. Results from this test showed that the nominal controller could significantly interfere with the low level collision avoidance maneuvers, to the point of causing a collision. For all future tests, the nominal controller override described in Section 5.3 was added near $t_{CPA} = 0$.

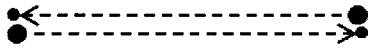


Figure 5.3: The 2-satellite head-on avoidance test examined the behavior near the degenerate case of exactly aligned relative velocity and relative position vectors.

Two more tests were performed in August, 2009. For these tests

$$\begin{aligned} t_a &= 10s \\ t_c &= 3s \\ d_a &= 0.3m \\ d_t &= 0.33m. \end{aligned}$$

The first of these tests examined the avoidance problem near the degenerate case when the relative position and velocity vectors are exactly aligned. Figure 5.3 shows the trajectory of both satellites in the X-Y plane. The inward facing arrows indicate the direction of thrust commands issued by the nominal controller and the outward facing arrows show the avoidance controller commands. The estimated CPA is also plotted for the time steps that the avoidance algorithm was active.

The test results verify two important properties of the avoidance method. First, the algorithm takes advantage of small separations in the projected CPA positions of the satellites and increases them to achieve a safe trajectory. In the first avoidance step there is a large thrust maneuver that significantly increases the CPA distance, and the remaining CPA positions remain clustered at a point the trajectory eventually passes through. Second, the controller can operate successfully even in the presence of a nominal controller driving it in the wrong direction. Following the first maneuver the back and forth exchange between the nominal controller and the avoidance controller can be seen until the satellite reaches the CPA, where the nominal controller is disabled for several seconds. The satellites coast safely past the CPA, then resume a trajectory to the final target.

A repeat of the 3-satellite tests from the previous session was also performed to see if the new parameters would help prevent grazing collisions. The first crossing is

pictured in Figure 5.5, where the arrows indicate the direction of the avoidance thrust commands, and the nominal controller is omitted for clarity. This test included a second crossing, where one of the satellites moved to the center of the test volume to act as an obstacle, and another moved slightly out of plane to include a 3D component in the initial trajectories. As shown in Figure 5.6, the two satellites attempting to cross through the center successfully avoid the uncooperative obstacle.

In contrast to the 2-satellite Head On Avoidance test, the 3-satellite avoidance maneuver performs pairwise calculations with each other satellite. Since the avoidance algorithm only chooses the most imminent threat (smallest t_{CPA}), these avoidance events occasionally conflict. For instance, during the first crossing, SPH1 and SPH2 perform an avoidance maneuver indicated by the arrows in opposite coplanar directions. At approximately the same time, SPH3 performed a maneuver in the positive Z direction. During the next time step, SPH2, now on a path clear of SPH1, performed a maneuver to avoid SPH3. SPH3 had to correct its initial move by thrusting in the negative direction. Similar behavior can be seen in the second crossing in Figure 5.6 when SPH2 initially attempts to go over the obstacle then changes direction and flies below it. With three satellites there are sufficient degrees of freedom that the pairwise conflicts do not pose a problem, and no collisions occurred. However, certain carefully constructed cases could be imagined that force the satellites into a collision trajectory. Though these test results indicate excellent avoidance behavior they encourage an investigation into a multi-satellite version of the collision avoidance algorithm that considers more than pairwise interactions.

5.5 Conclusions

This architecture is particularly well suited for close proximity science demonstration missions where the nominal controller may not have the main objective of avoiding collisions but still requires some form of avoidance. This is ideal for Zero Robotics because the student users can focus on developing general control strategies without additional code space consumed for avoiding their opponents. With only a few small

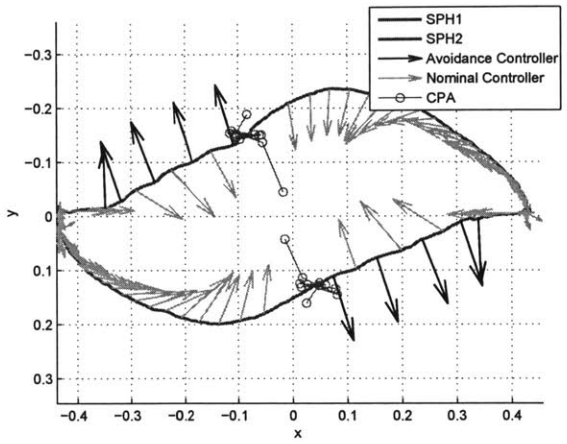


Figure 5.4: A planar view of the head-on collision avoidance test. Shortly after the satellites move toward the center the avoidance controller starts pushing the satellites away from each other. The estimated CPA moves from the center of the volume to a safe distance where it stays. Near the CPA, the nominal controller is disabled for several seconds until the collision threat is over.

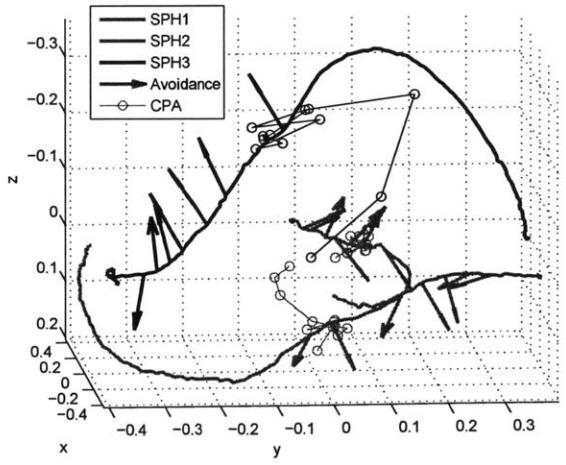


Figure 5.5: In the 3-satellite test, all three satellites attempt to cross through the center simultaneously. The pairwise avoidance events are clear from the coplanar thrust commands.

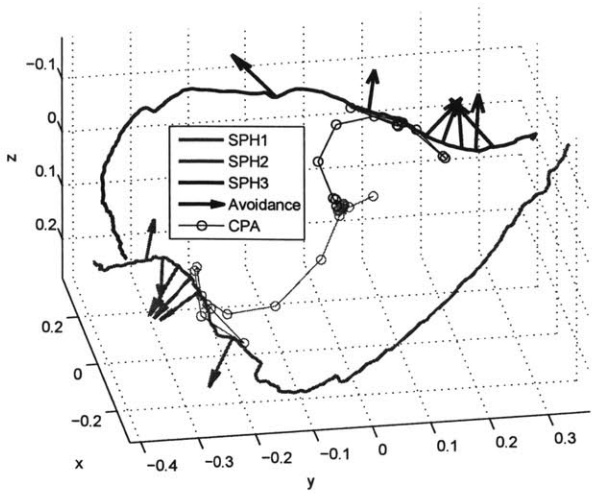


Figure 5.6: When the satellites cross the test volume a second time, SPH1 becomes an uncooperative obstacle, and SPH2 moves out of the initial crossing plane. The collision avoidance controller effectively directs both satellites around the stationary obstacle on opposite sides.

calculations, the algorithm is easy to implement in a computationally constrained system and requires only a small overhead in code size. In several cases, the controller trades performance for simplicity, particularly in terms of fuel consumption. Lastly, it is important to note that for the purposes of Zero Robotics it has been sufficient to assume that the limited horizon controller override will successfully prevent collisions in most scenarios. Rare exceptions are permissible and will likely only result in grazing collisions. More formal analysis of the switching behavior to exclude the possibility of instabilities or limit cycle behavior and provide a formal guarantee of collision avoidance remains as future work.

Chapter 6

Conclusions and Future Work

6.1 Thesis Summary

To conclude, we will return to the design principles introduced in Chapter 1 and examine them from the context of the complete implementation.

6.1.1 Engage and Educate

To create a platform for both drawing students in and building an educational experience, this principle must play a role in the design of all components. At the most basic level, Zero Robotics tournaments are built on the fundamental attraction of a chance to run an experiment in space along with the inherent excitement of competition. Once students begin competing, the game design and tournament structure covered in Chapter 2 serve to keep them involved and learning. With a link to a realistic space-related theme, incorporating math and physics challenges, solving the game requires a cross-disciplinary application of STEM subjects. Careful balancing helps to keep the game interesting.

As an educational tool, the Zero Robotics platform presented in Chapter 3 is not restricted to just the annual scoped challenges of the tournament. It can serve as an open-ended tool for learning about programming and a wide range of physics principles. On the programming side, students have access to most features of the C

programming language along with some support of C++. The programming concepts can then be connected to exploring the concepts of force and torque, and how they relate to the motion of a rigid body.

Lastly, with the Leaderboard scoring system in Chapter 4, efforts have been made to supply several tools for the analysis of ranking information instead of hiding the underlying implementation.

6.1.2 Accessibility

A virtual competition like Zero Robotics has the distinct advantage of being able to utilize mature web-based technologies and the wide availability of internet access in schools for opening access to a large audience of participants. The Zero Robotics platform is the main embodiment of this principle with centralized resources for writing, compiling, and simulating programs for SPHERES. By operating completely in a web browser, the system avoids the need to license expensive software, and users can begin programming from any computer without obtaining installation privileges or configuring software.

Accessibility is also achieved through the design of the tournament. By starting out with a simplified problem in two dimensions and progressing to more difficult challenges, the format provides an onramp for novice teams to get started and be successful.

6.1.3 Incremental Difficulty

Incremental difficulty operates at two scales in Zero Robotics. The first is the progressive increase in difficulty built into the tournament season and game design intended to promote accessibility, keep teams engaged throughout the season, and keep flexing their skills. The second is a spectrum of options to accommodate different skill levels. The game balancing process includes considerations to ensure multiple difficulty levels, and the Zero Robotics API provides a range of options for controlling the satellites.

6.1.4 Efficient Inquiry

A concerted effort has been made throughout the design of the Zero Robotics platform to minimize the round trip time from writing code to analyzing simulations. The SPHERES simulation is designed with separate modular components so that each simulation run only involves the compilation of user code, and the simulation has been carefully optimized for runtime performance. Simulations are recorded to a database for review at any time, and the 3D visualization tool can replay a match from any perspective.

The Leaderboard system takes the same concepts and applies them to providing scoring information throughout a tournament. Teams can submit a project and see an updated ranking within a few minutes of submission, along with the full details of each match performed.

6.1.5 Authenticity

Through a high fidelity simulation model, Zero Robotics gives students a realistic picture of the operation of a dynamic robot in space. Over many years, the SPHERES simulation model has been refined to provide a close representation of the hardware platform with a complete model of the sensor, actuator, and software subsystems. The online compilation infrastructure enforces compatibility with the hardware and challenges students with accurate though tightly constrained code size and computation time restrictions.

To achieve the most reliable results on the ISS and accurate simulation results, restrictions are added to the virtual game in the form of a collision avoidance law presented in Chapter 5 and boundary limits enforced by the game rules.

6.2 Contributions

Collectively, the design, implementation, and analysis of the Zero Robotics program have resulted in the following contributions:

Created a framework for building virtual platforms as a surrogate for hardware in student robotics competitions

The combined platform design principles and the example case of Zero Robotics represent a model for implementing virtualized hardware competitions with a simulation in place of a robot. The same approach has applicability to other hardware platforms and research laboratories with limited available testing time.

Created a versatile, easy to use programming API for controlling 6DOF satellites

The Zero Robotics API provides generic control of double integrator dynamics at the level of forces, velocities, or position targets or in any combination with a simplified interface for access by students. In application, the API has allowed students to successfully implement and demonstrate programs on SPHERES. Students have accomplished tasks such as obstacle avoidance, attitude scanning maneuvers, docking, and formation flight, all using the interface.

Designed a web-based architecture for writing, compiling, and simulating code for a dynamic robot

The architecture in this work links a highly accessible programming environment to a dynamic simulation, all hosted online. Combined, these tools extend the elements of the virtual platform framework to give anyone with a web browser the ability to write programs for the simulated robot platform with very little startup time.

Developed a pattern for fast, highly detailed, web-based simulations

This work details the components required to make a sufficient simulation representation of a robot, including software-in-the-loop capabilities; modular, pre-compiled components; dynamic loading; and operating system level modeling of execution. The latest SPHERES simulation is the fastest (5x-10x real time) and most accurate version of the SPHERES simulation to date. It is in use by not only thousands of stu-

dents on the Zero Robotics platform, but also by all SPHERES researchers developing programs for ISS tests.

Designed a responsive, game-agnostic scoring system

The Zero Robotics Leaderboard scoring system is an improved implementation of the Whole History Rating rating algorithm designed to give competitors continuous feedback about their simulation performance. The algorithm has been updated with an Armijo Rule guard to improve stability along with implementation modifications to improve responsiveness.

Created a lightweight control law for close-proximity satellite avoidance

An always-on supervisory control law originally developed for close proximity satellite formation flight with SPHERES has been used successfully in all Zero Robotics tournaments as a basic guard against collisions. The implementation is compact in size and requires little computational effort.

Co-founded and led the Zero Robotics STEM outreach program

Lastly, the Zero Robotics program itself has contributed to STEM education and the world of competition robotics. It provides the unique features of model-based online simulation, programming, 6DOF dynamics, a multi-week team collaboration phase, and the chance to test satellites aboard the ISS. It is now an international program with 6 iterations of unique challenges, over 2000 students to date, and has led over 200 students to "touch space" with the direct experience of running an experiment on the space station.

6.3 Future Work

As an immense collection of interconnected components, generating hundreds of thousands of simulations each year, the Zero Robotics program has many opportunities

for future exploration. This section will highlight several of the paths that could be possible in the near term along with a number of pressing challenges.

6.3.1 Research Directions

Game Design

Though this work provides a set of game design guidelines there are many more ways to improve the game design process for Zero Robotics. Interesting ideas include developing automated tools for tuning game parameters or automated generation of AI players to play against competitors.

Recent expansions to the SPHERES hardware, including a new single board computer and cameras added for the recently launched VERTIGO program, open up many interesting directions for future game designs. Incorporating these components must be approached with careful attention to the original platform design principles. For instance, ensuring a simulation of the new hardware components is available will be essential before adding them to an official tournament. It is highly recommended not to add a major hardware change to an official tournament without running a limited scale pilot competition first. Neglecting to do so risks frustrating and alienating teams if the available tools are not at par with the rest of the infrastructure.

Innovative Interfaces for Web-Based Programming

The current tools for collaboration allow for project sharing and some real-time interaction. A frequently requested improvement is the addition of real-time editing shared between browser similar to the functionality of Google Drive. Careful research will be required to find a solution for preserving compilability of the code while collaborating.

Data Mining

During the course of a Zero Robotics season, hundreds of thousands of simulations are run by the teams. Aggregate analysis of these simulations could provide a real-time snapshot into how well teams are performing in the current challenge or lend insight

into the performance of control laws running underneath the student code. The large data set could even be used to build gradually improving AI players from competitor examples.

Visualization Tools for Simulation

The current 3D visualization is based on a flash library that is no longer supported by the developer. The visualization should be improved with modern web technologies such as WebGL and include new tools for more accessible data analysis.

6.3.2 Monte Carlo Tools for the Zero Robotics Platform

Appendix C presents a concept for a Monte Carlo simulation tool for exploring robustness to parameter variations in programs. Users could use the tool to explore custom variations in a program and evaluate user-defined constraints to determine if the simulations were successful. This tool would enable better understanding of the effects of parameter changes over an ensemble of tests instead of from one simulation to the next.

6.3.3 Formal Evaluation Studies

With the basic technical infrastructure completed, Zero Robotics urgently needs to begin more formal studies of its impact on students. As part of the 2012 season, a formal pre- and post-survey was designed to evaluate the impact of Zero Robotics on measures of self-efficacy as well as career expectations. The initial results has a very low response rate, and the overall results were inconclusive. Future studies will need to find effective ways of increasing participation in the studies in accordance with regulations on the use of humans as experimental subjects.

6.3.4 Scaling Challenges

ISS Dilution

As Zero Robotics grows and a smaller percentage of teams gets to proceed to the championships, there will be a natural dilution of the effect of the final ISS tournament on encouraging participation. Plans for growing the program need to account for this in the way the tournament is structured. There are a number of paths to keeping the program interesting:

- Add in-person real-time programming competitions to the final event. Users that don't reach the ISS could still travel to the championship to compete in head-to-head live programming challenges. This model has been used successfully by TopCoder in the TopCoder Open programming event.

- Add a hardware component. Several ideas have been considered for enabling a hardware phase of the competition. Users could program a ground-based robot using the Zero Robotics API, then compete in local regionals using the hardware. Winners from the regionals would proceed to the ISS phase.

Leaderboard Scoring and Cloud Infrastructure

As participation grows, the data storage and processing requirements will grow significantly. If 1000 teams run 20 matches per submission for 20 submissions, the 400,000 simulation runs over a single competition will equal the number all simulation runs to date in Zero Robotics. With this many teams, it may be necessary to transfer updates of the Leaderboard algorithm to a dedicated computer responsible for running the algorithm updates. If at some point the competition becomes too large to efficiently process all match results in the main algorithm updates, it may be necessary to change to a different scoring algorithm or divide teams into regional tournaments of a fixed size. The algorithm also admits a parallel implementation that could be spread across a large cluster of computers.

6.3.5 A Development Roadmap For Zero Robotics

Based on user feedback and future planning, the following is a rough sequence of priorities for new software development on the platform

1. Performance improvements for the ZR IDE. Users report sluggish behavior and occasionally lose projects. This is critical for maintaining the accessibility of the platform.

2. Separate ZR simulation deployment from MATLAB. To simplify updates of the online simulation and pave the way for making the simulation freely available. The main execution loop should be removed from MATLAB. This is possible using the code generated version of the simulation.

3. Add a visualization plugin for the ISS GUI. To enhance the final tournament it would be possible to implement a 3D visualization running as a plugin within the SPHERES GUI, then use a camera to relay the view to the ground.

4. Implement an open API to the online project editing and simulation tools. Many users request the ability to edit code offline or perform simulations independent of the ZR infrastructure. A clearly defined web API such as a RESTful interface would enable this capability.

5. Provide an in-browser version of the SPHERES simulation that allows line-by-line debugging. It is possible to compile the simulation into a module loadable by Google's Native Client, a tool for running C++ in a web browser. This would allow students to run simulations in their browsers and maintain the accessibility of the platform.

6. Create a downloadable version of the SPHERES simulation and game libraries. The next step after a browser-based simulation is a downloadable version of the simulation. This would allow teams interested in working separately from the online environment to do so.

7. Open up programming interface for full SPHERES programming. The Zero Robotics API can in principle support all functionality of the GSP interface. This step would create a special "advanced" game that would allow any researcher to use the Zero Robotics game for developing flight code.

Appendix A

Zero Robotics-Specific Implementation Details

The following items are additional requirements for game and implementation that are specific to the Zero Robotics Tournament.

A.1 Game Implementation

A.1.1 Scoring Systems

Scores will eventually need to be restricted to the range of $[0, 22]$ due to the test result value restrictions discussed in Section A.1.3.5, but it is possible to re-scale an arbitrary scoring scheme to this range if needed. For the online visualization environment, any range of scores can be displayed as long as it is possible to transmit them in telemetry.

A.1.2 Code Size Limits

Code allocation for a high school tournament game should not drop below about 6.4KB per user.

A.1.3 Standard Game Phases

The basic components of a Zero Robotics match have stabilized to the point that a common outline can be used for all implementations. This section describes the format of a typical Zero Robotics game. Except where noted, most of the actions described here are automatically performed when a game is built using the Zero Robotics API. Each of the phases are denoted *maneuvers*, a standard component of the SPHERES Guest Scientist Program API that usually represent states in a state machine controlling the behavior of the satellite.

A.1.3.1 Maneuver 1: Estimator Convergence and Opponent Selection

Prior to the start of an ISS match, the crew positions the satellites in the approximate location where the program expects them to start. At the start of the test, the satellites drift freely for 10 seconds while the SPHERES estimator converges close to the true position of the satellites. During this period, the crew is instructed to press a key on the keyboard to assign an opponent for the match. If the crew does not select an opponent, the satellites automatically terminate the test run.

When the game runs in the online simulation environment, the initial estimation period still takes place, but the logged telemetry automatically truncates the 10 second period of the simulation. where the satellites are in free drift. The opponent selection is also ignored because the simulation software automatically loads the correct programs for each satellite.

A.1.3.2 Maneuver 2: Initial Positioning

Correct initial positioning is a critical assumption for achieving game symmetry and ensuring a fair match on the ISS. The initial positioning phase moves the satellites to the pre- programmed starting condition for the match with 30 seconds of time to reach the target. The match starts immediately after the time limit expires. The simulation places the satellites in their intended positions and skips the initial positioning maneuver. ISS tests will have some initial error in the positioning, so perfect

initial placement should not be critical to the game outcome. An analysis of initial positioning errors is available in Appendix C.

A.1.3.3 Maneuver 3: Game Update Loop

The third maneuver contains the main updates of the game and actions by the user. It typically executes in a 1 Hz loop inside the method `update()` (see Section 3.4.4.1). The implementation is up to the game designer, but it usually includes the following components:

1. Game rules update pre-user
2. Run user code
3. Game rules update post-user
4. Command thrusters
5. Check termination conditions
6. Send communication packets

The first step performs updates to the game code that must be propagated from the previous control cycle. Any flag or setting that should be active for the current control cycle should be set here. This can include adjusting the game state based on the satellite's current position such as revealing hidden objects according to the satellite orientation, or indicating the completion of an item pick-up from conditions on position, velocity, and pointing direction.

The next step is to trigger a user code update by calling the `loop()` method in the user's custom-designed class. It is up to the game designer to decide where in the game update the user code is called. Triggering the user's program will result in calls into the Zero Robotics and game APIs, which are processed in the third step. The third and fourth steps perform final cleanup operations based on the user actions, prepare for the next time step, and command the satellite thrusters based on the desired forces and torques.

The fifth step checks for all conditions that might end the game. All Zero Robotics games include a maximum duration timeout, but the games may end early for other reasons like both satellites expending their allocated fuel or one or both of the satellites completing all tasks in the challenge. An endgame finale can even be implemented in such as way as to immediately terminate the match if it is achieved. The satellites may also terminate if an error condition occurs. If one of the satellites resets during a match the other will immediately terminate the test with an error indication so the match can be restarted.

The last part of the game loop is to broadcast synchronizing game state information to the other satellite in the match. The updates usually contain key shared information about the final state of the game after the most recent update. Section 2.2.4 will discuss strategies for sharing this information.

A.1.3.4 Maneuver 240+: Termination Phase

Except when an error occurs, the satellites do not immediately terminate when game end conditions are met. Instead, they enter a holding maneuver, usually numbered 240 or higher, and wait for the partner to do the same. The holding maneuver can include additional visual motions to signify the completion of the game such as adding a slight spin about a known axis. Entering a termination maneuver keeps the satellite program active but stops the user code from running. This is especially critical for keeping the global metrology system running because one of the satellites is responsible for triggering the infrared updates that initiate each metrology cycle.

After both satellites enter a termination maneuver, they wait for several additional cycles to ensure the final game state is in sync. This is very important when game state values like the finish time in a race are set at the very end of a match and may take an additional cycle to be processed on the partner satellite. Once synchronization completes, the satellites terminate with a test result number.

Based on experience from several Zero Robotics test sessions, it is a good idea to keep the game volume boundary limits active in the termination maneuver to prevent the satellites from drifting out of the volume. Even though the satellite's user code

is disabled, it will continue sending state telemetry updates to the partner satellite, allowing for collision avoidance events to take place if the other user is still active.

A.1.3.5 Test Result Value

The only feedback currently available from a hardware Zero Robotics match is a standard 1 byte SPHERES test return value. Numbers 0-10 and 255 are reserved for standard result codes. The current standard practice for Zero Robotics is to split the result code into a rounded score value and a numeric identifier corresponding to an assigned team number 1-9. The score part of the number indicates the winner of the match and their performance, while the ID ensures the correct competitors were selected by the crew. Though the identifier consumes a large portion of the number space, it has proved to be essential in on-orbit testing and should not be omitted.

The test result value is computed with the following formula:

$$result = \text{round}(score \cdot 10) + 10 + ID. \tag{A.1}$$

Here *score* is the floating point score in the range of $[0, 23]$. To prevent the possibility of ties, *score*, is first compared to the opponent's floating point score and awarded an additional bonus point if it is larger. In this case, the score range is restricted to $[0, 22]$. The final result is a number in the range $[11, 249]$, within the 1 byte limit and outside the range of the reserved test result numbers.

To give the highest fidelity of scoring performance, the $[0, 22]$ range is often mapped to the most likely subset of the possible scoring values. If a user happens to score outside of the expected range, the score should simply be saturated at the maximum value of 22, and the same bonus point calculation can be applied to indicate the winner of the match.

Simulated matches on the Zero Robotics website have more facilities available to process the results of a match. At the completion of a simulation run, the website receives all telemetry data sent from the satellite. As part of the website game configuration, the game designer specifies a script to process the telemetry into a

numeric score. Any component or composite calculation from the telemetry may be used to judge the performance, but it is best use something that can eventually be calculated during the ISS finals.

A.1.4 Communications

A.1.4.1 Typical Uses for Standard SPHERES Packets

Standard practice for Zero Robotics to date has been to transmit only 3 packets per second per satellite, corresponding to the 3 standard data types that can be overlaid on the payload. The types and their common uses are explained below:

float Array of 8 single precision 4-byte floating point numbers. This packet often contains quantities that require floating point precision instead of a rounded integer representation. The score is usually stored and transmitted as a floating point number to ensure ties are very unlikely.

unsigned short Array of 16 2-byte unsigned integers. This packet usually contains simple counting variables or event completion times that do not require a sign.

short Array of 16 2-byte signed integers. The short packet is helpful for sending approximations of continuous values that will not fit in the float packet. The numbers are usually scaled by a multiplier to avoid truncation.

For visualization purposes, all packets sent by the game must use the first element of the array as a time stamp, leaving 7 elements for the float packet and 15 elements for the two integer packets. The time stamp should be in seconds for the floating point packet and tenths of seconds for the integer packets. When processing the telemetry after a run, the simulation software will replace packets with duplicate time stamps with the most recently received packet.

Both the Zero Robotics visualization and the simulation report tool assume the packet format is fixed for each game, so changes to the packet format should be avoided during the season. Any updates must be propagated to the visualization and report tool.

A.1.4.2 Improving Ephemeral Data Transmissions

For more consistent ephemeral transmissions, two additional approaches might be implemented for future games. First, the packets could be sent multiple times during the 1 Hz control cycle by adding an additional minor update period half or a quarter way through the 1 second period. Second, the item in use could have a finite power-up time, during which an incrementing sequence count with a known limit is transmitted to the opponent. If the opponent receives the sequence at any point during the power-up period, it can complete the sequence without receiving any additional packets. Again, a delay is introduced, and the satellite must commit to using the item as soon as the sequence starts.

A.1.5 Game Manual

A Zero Robotics game manual should contain at least the following items:

Challenge Statement It has been traditional in Zero Robotics games to craft a fictional story based on the theme that summarizes the tasks in the game. This statement is usually included at the beginning of the manual.

Gameplay A detailed description of each component of the game. The description should start with a broad overview of the game, including diagrams to summarize the layout of key features. The section covers the both the capabilities of the satellite, including any restrictions, like time, fuel, charge, and code size, and the steps necessary to complete the game. All steps should include scoring information where relevant.

Scoring Summarizes the scoring information presented in the game game play overview for quick reference.

Tournament Detailed description of the phases of the tournament and the overall tournament scoring system.

Rules Summary of all rules presented in the preceding sections. Also outlines rules for updating the game and a code of conduct.

Version History It is expected that the manual will undergo changes during the season, and this section tracks all updates. A version number should be clearly visible at the beginning of the document, and a dated change log should be present at the end.

Throughout the manual it is useful to refer to elements of the game API with brief summaries. Links should be available to documentation generated from the source code of the API for more detailed usage information.

Prior to the game release, a careful examination of the entire manual is necessary to ensure the rules are in sync with the programmed game implementation. The best way to achieve this is to develop the manual in parallel with the game, using the manual as a rough software specification. As updates are made to either manual or implementation, the changes are kept in sync.

A.1.6 Game Development Timeline

The following schedule provides a framework to work toward a September launch of the game:

Dates	Name	Description
Jan-Feb	Brainstorming / Conceptualization	Study results of the previous seasons and generate ideas for a new game concept.
March	Concept Selection	• Choose one or two concepts for detailed study. • Outline the game rules and scoring systems as a basis for the manual.

April	Prototyping	• Implement components of the game rules as modules. • Start assembling a complete game.
May	Prototype Complete	• Complete the first prototype of the game. • Perform preliminary balancing analysis. • Ideally, the prototype should complete with a brief design review to present the current design. • A draft manual should be completed.
June	Game Updates	• Based on feedback from the design review or lessons from the initial analysis, implement any additional updates to the game. • Aim for a full game draft by the end of the month.
July	Standard Players and Final Tuning	• Develop standard players. • Perform final balancing analysis with standard players. • Finalize manual details. • End month with completed game and final readiness review.

August	Code Freeze	• In preparation for the game launch, freeze code development as much as possible. • Deploy first game to website and test functionality. • Address any last-minute bugs from final testing.

A.1.7 Code Preparation for ISS

It is the responsibility of the Zero Robotics team to prepare the final code package sent to the ISS for the competition season. Preparation of the code package typically starts immediately after the finalist code submission deadline. At least one full round-robin simulation should be conducted using the code in its flight configuration. Several simulations from each competitor should be examined carefully for any potential anomalies.

Since the users' solutions are full 6-DOF programs, the final checkout operations are mostly limited to verifying that the software runs until the game times out. It is usually possible to check that the satellites attempt to move their appropriate initial conditions and that the satellite attempts to execute the first action specific to the user program.

A.2 Simulation Details

A.2.1 S-Function Interface

Each of the satellites are configured with a set of parameters that are static for the duration of the simulation. They are used by the S-Function interface and lower layers to model components of the satellite that are not part of SPHERES Core but

are accessed by the software.

userLibraryName, commLibraryName Name and path to the shared object containing SPHERES Core and user code and the communications simulation library. The library is dynamically loaded by the S-Function interface.

hardwareId Uniquely identifies the satellite. The hardware identifier is normally stored in flash on the satellite, so this value is written to the virtual flash memory in the SPHERES Core Wrapper.

logicalId An integer identifying the logical role of the satellite for the current program, usually 1, 2, or 3. The logical ID is also compiled into the SPHERES Core and User code layer, but it is passed to the S-Function layer so it can identify which satellite to update in a simulation step.

test_number The test number selected by the operator for the current test run. At the start of a simulation, the test number is passed to the simulated communications manager, which initiates a "start test" command packet from the virtual ground station.

sphActive Indicates if the selected satellite is active in the simulation. Since the Simulink simulation does not allow for a dynamic number of satellites, all satellites are compiled into the simulation model. The sphActive flag turns off the software model for inactive satellites so the S-Function interface does not attempt to load their libraries.

nSph The total number of satellites in the simulation.

bcnPos, bcnDir Matrices containing the position and normal vector of the ultrasound beacons in the global frame. The beacon locations are normally transmitted via communications packets to the satellite upon bootup and prior to tests, but in the simulation, the SPHERES Core Wrapper skips the communication and calls the function *padsBeaconLocationSet* directly to set the locations and directions.

randSeed Among the random seeds computed in Section 3.3.2 is a value to initialize random number generators within the C/C++ code of the simulation. This value is useful if the code being simulated must produce some form of randomness such as in a Zero Robotics game.

A.2.2 S-Function Interface Inputs and Outputs

At each step of the simulation, the following inputs and outputs are passed from Simulink to the S-Function interface for each satellite. The following input and output elements are defined in the interface:

gyroCounts [in] The 32-bit values $\mathbf{z}_\omega$ computed in the gyro measurement model. The SPHERES Core Wrapper loads these values into the virtual FPGA register A2D_ADDR.

accelCounts [in] The 32-bit values $\mathbf{z}_a$ computed in the accelerometer measurement model. The SPHERES Core Wrapper loads these values into the virtual FPGA register A2D_ADDR.

usFlag [in] Set to 1 when a new global metrology distance vector is available from Line 12 of Algorithm 3.2. The SPHERES Core Wrapper uses this value to set the FGPA_STAT_FRAME flag in the virtual FPGA register FPGA_STAT_ADDR.

usMatrix [in] Configured with the distance vector computed on Line 9 of Algorithm 3.3. When *usFlag* is set, the matrix will be read from the FPGA register MATRIX_ADDR when the wrapper runs the SPHERES Core function *HWI_pads_rcv()*.

runtimeCmd [in] Models a command sent from the ground station laptop keyboard. In the simulation the runtime command is static for the full simulation and is sent in a command packet originating with the virtual ground station in the communications simulator.

UARTRx [in] Incoming UART data from the SPHERES expansion port. The data stream is byte limited to reflect the baud rate of this interface.

thrPtr [out] Array of values for activating each of the 12 satellite thrusters. This array is the main input to the dynamics module.

irTx [out] Flag for activating a global metrology cycle described section 3.3.4.3. In addition to passing the signal out of the software model, if at the end of a time step any of the satellites have set irTx, the S-Function interface will signal each SPHERES Core Wrapper to immediately fire the IR interrupt *HWI_IR_rcv()*.

UARTTx [out] Outgoing serial data. Like the UARTRx channel, it is byte limited to reflect the baud rate of the interface.

terminate [out] Set to 1 if the software model has encountered an internal error and needs to signal the simulation to terminate immediately.

SPHERES Core Status Variables [out] Several standard values from SPHERES Core are passed out of the software model for logging purposes, including the current test time, current maneuver number, maneuver time, the estimated state, and the latest test result number.

A.2.3 Thruster Transient Modeling

So far most SPHERES simulations have assumed a negligible transient when opening and closing the thrusters. It is not entirely clear if this is a poor assumption because the transient has been very difficult to measure with lab equipment. SPHERES thruster characterization performed in [9] indicates an exponential rise time of approximately 10ms, though the estimate is very rough due to noise and vibrations on the test stand. Future simulations might use a better thruster model with a delay and transient as shown in Figure A.1 because it is very easy to implement and models a varying impulse for very short thrust durations. To match an exponential rise time with a delay and slope, an equivalent impulse can be used. The following derivation matches the impulse of a linear slop and delay model to an exponential model for a

period of n time constants.

$$\begin{aligned} \tau &\approx \frac{t_{rise}}{2.197} \\ J_{exp} &= \int_0^{n\tau} (1 - e^{-t/\tau})\, dt = (n-1)\tau + \tau e^{-n} \qquad \text{(A.2)} \\ J_{linear} &= \int_0^{t_r} \frac{1}{t_r} dt + n\tau - (t_r + t_d) = n\tau - \frac{t_r}{2} + t_d \qquad \text{(A.3)} \end{aligned}$$

Equating J_{exp} and J_{linear} and solving for the linear rise time in terms of the delay results in

$$t_r = 2\tau(1 - e^{-n}) - 2t_d.$$

Taking the limit as $n \to \infty$, the best approximation is

$$t_r = 2(\tau - t_d).$$

Note that this means the delay time must be shorter than the time constant of the exponential, or the linear transient will provide less impulse than the exponential rise.

Example: suppose the thruster transient rise is indeed 10ms, resulting in an exponential time constant of $\tau = 0.0046s$. Suppose we assume $t_d = 1ms$ for the opening delay

$$t_r = 2(0.0046s - 0.001s) = 0.0036s.$$

To decide if the new model should be used, both approaches should be compared in simulation to see if changing the delay model has a significant effect on fuel consumption, dynamics, or control performance. It is likely that the approximation error is small compared to other uncertainties such as the magnitude of the thruster forces at each firing time.

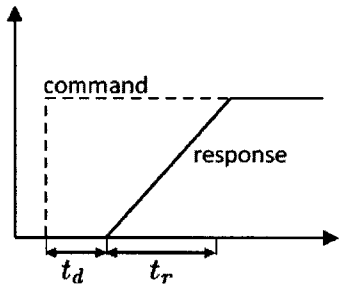


Figure A.1: A useful approximation for thruster opening and closing dynamics is a delay t_d followed by a linear transient with a specified rise time t_r.

Appendix B

SPHERES Parameters and Uncertainty Quantification

B.1 SPHERES Thruster Geometry

The configuration table from Hilstad [32] is reproduced below.

Thr #	Thruster Position [cm]			Resultant Force Direction			Resultant Torque Direction		
	x	y	z	x	y	z	x	y	z
1	−5.2	0.0	9.7	1	0	0	0	1	0
2	−5.2	0.0	-9.7	1	0	0	0	−1	0
3	9.7	−5.2	0.0	0	1	0	0	0	1
4	−9.7	5.2	0.0	0	1	0	0	0	−1
5	0.0	9.7	−5.2	0	0	1	1	0	0
6	0.0	−9.7	−5.2	0	0	1	−1	0	0
7	5.2	0.0	9.7	−1	0	0	0	−1	0
8	5.2	0.0	-9.7	−1	0	0	0	1	0
9	9.7	5.2	0.0	0	−1	0	0	0	−1
10	−9.7	5.2	0.0	0	−1	0	0	0	1
11	0.0	9.7	5.2	0	0	−1	−1	0	0
12	0.0	−9.7	5.2	0	0	−1	1	0	0

B.2 Sources of Uncertainty in ISS Testing

The first step toward implementing the tool is creating a traditional Monte Carlo system that interfaces with the Zero Robotics platform. A standard approach is to identify a set of important simulation parameters and their uncertainties, then generate random samples to apply to simulations. The resulting state trajectories or *dispersions* are analyzed for conditions that violate constraints.

During the design and initial operations of SPHERES, both the ultrasound global metrology system and the propulsion system were carefully characterized. Most of these characterizations have been incorporated into the SPHERES simulation with both deterministic and stochastic components, but the parameter are set to static,

nominal values (see Table 3.3.1 for a summary of the parameters). For Monte Carlo analysis, it is necessary to determine which of these parameters to vary and how much. This section identifies several sources of parameter variation that occur during the hardware testing phase aboard the ISS.

B.2.1 Mass Properties

The mass of the satellite is well known based on measurements of the hardware before launch to the ISS. The only variation in mass is due to the use of CO_2 propellant, totaling $170\,g$. In 2012, an expansion port was added to the satellites, bringing the empty mass with a tank and batteries to $4.365\,kg$ The mass range is therefore physically limited to $4.365\,kg \leq m \leq 4.535\,kg$.

The nominal inertia matrix for SPHERES is

$$\mathbf{J} = \begin{bmatrix} 0.0285 & -8.37 \cdot 10^{-5} & 1.4 \cdot 10^{-5} \\ -8.37 \cdot 10^{-5} & 0.0283 & -2.9 \cdot 10^{-4} \\ 1.4 \cdot 10^{-5} & -2.9 \cdot 10^{-4} & 0.0245 \end{bmatrix}$$

The uncertainty bounds on the inertia matrix $\mathbf{J}$ have not been well characterized to date. Recent additions of expansion items has prompted additional study of the satellite mass properties, so this information may be further refined in the near future.

B.2.2 Thruster Performance

The preliminary SPHERES thruster analysis conducted by Chen in [9] includes theoretical and bench testing results and discusses the main factors affecting variations in the thruster performance:

- **Pressure regulator settings**. Based on a linear fit of bench data, the pressure regulator setting is related the thrust by $T = 0.0033 \cdot psia - 0.0049$. The setting on the pressure dial can vary as much as $\pm 2\,psi$ from the actual value. In the worst case we might also assume the crew may erroneously set the dial within a $\pm 5\,psi$ range. This results in a thrust variation of up to 0.2N.

- **Nozzle area variation.** Changes in the profile of the thruster nozzle can result in differences in thrust between individual thrusters. The overall standard deviation is $\sigma_{area} = 0.004434\,N$. Differences between the thrusters are not explicitly modeled in the simulation, but a uniform random thrust level of $\pm 5\%$ is applied each time the thruster opens. This captures up to $2\sigma_{area}$ of the variation, but it is averaged over the time history of the simulation. To better capture the potential variation of nozzle areas, a random bias could applied to all the thrusters once at the beginning of the test, or to more clearly relate to a Monte Carlo parameter, a small torque bias could be applied to each thruster pair.

- **Number of thrusters activated.** This is one of the most significant effects, though a deterministic model is used in the simulation. The thrust is reduced by about 6% for each thruster thruster opened after the first. See section 3.3.3 for details.

Though these known variations are incorporated in the simulation, multiple test sessions have shown qualitative differences in thruster performance between simulation and ISS. To better anchor the simulation, we can examine test data from the 2012 RetroSPHERES final competition. In this test session the telemetry included information about the commanded forces to be applied to the satellites at each time step. By matching up the force commands with background telemetry information from the SPHERES estimator, we can determine a rough approximation of the ratio between commanded and applied $\Delta\mathbf{v}$ at each time step. The approximation procedure follows this procedure at each time step n in the data:

1. Interpolate background telemetry data at the time step corresponding to the force command telemetry. Extra care must be taken when interpolating the velocity data because the background telemetry may fall in the middle of a firing period when the velocity is changing rapidly. To avoid a mistake, the most recent telemetry packet before the current timestep is used for the velocity measurement.

2. Approximate the torques. Torque data was not present in the 2012 data (for best results torques should be included in future telemetry), but the torque (impulses) can be roughly approximated with the inertia matrix $\mathbf{J}$ and the angular velocity changes:

$$\tau_n = \mathbf{J}\Delta\omega. \tag{B.1}$$

3. Run a model of the mixer. After extracting forces and torques, the commands are passed through the standard SPHERES mixer[1]. The mixer compensates for thruster saturation and coupling between force and torque. After the mixer model, the resulting firing times can be converted back into impulses:

$$\Delta\mathbf{v}_{cmd} = \frac{F_{thruster}}{m}\mathbf{M}\mathbf{u} \tag{B.2}$$

where $\mathbf{u}$ is a vector of thruster pulse widths, $\mathbf{M}$ is the *mixing matrix*, m is the assumed satellite mass, and $F_{thruster}$ is the assumed nominal thrust value. Note that the thruster force and mass are not individually observable without knowledge of one of the quantities.

4. Extract the velocity measurement from the next time step. Find the velocity change

$$\Delta\mathbf{v}_{act} = \mathbf{v}_{n+1} - \mathbf{v}_n. \tag{B.3}$$

5. Find the ratio of commanded to actual velocity. The ratio can be viewed as a thrust attenuation factor k_T

$$k_{T,n} = \frac{\|\Delta\mathbf{v}_{command}\|}{\|\Delta\mathbf{v}_{cmd}\|}. \tag{B.4}$$

Figure B.1 shows all thrust command events from the US section of the 2012 finals, excluding individual trajectories where the satellites were predicted to have run out of propellant. The fit line is an approximation performed with the MATLAB Statis-

[1]Instead of using force and torque telemetry values, it is also possible to send the thruster pulse widths directly to avoid this step.

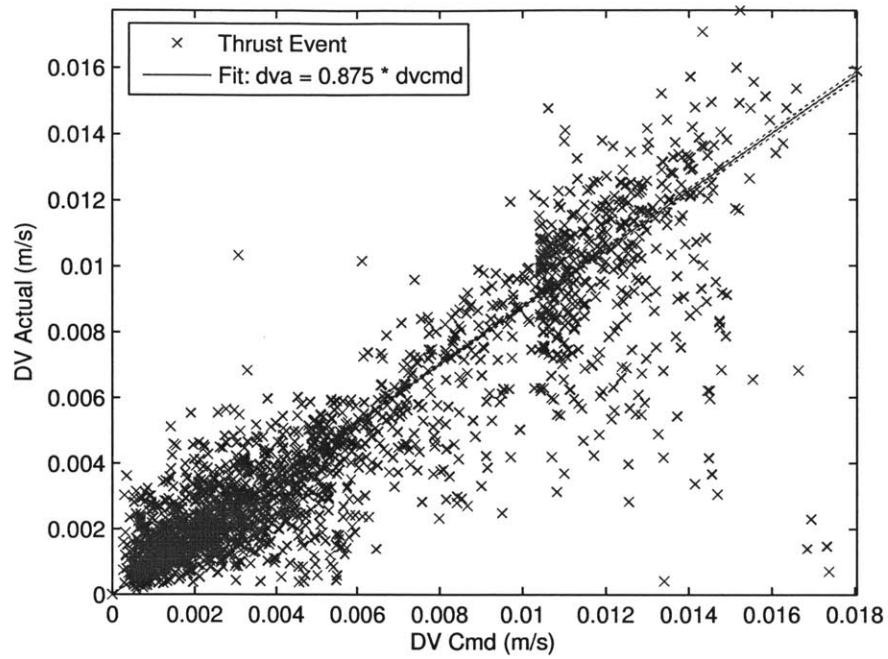


Figure B.1: Comparison of commanded velocity change to actual velocity change for ISS data. Each data point represents an independent thrust event during the 2012 ISS US finals. The fit line shows a rough approximation of the thrust attenuation factor.

tics Toolbox with a forced 0-intercept and a "Robust" linear regression that uses an iterative method to re-weight outlier data points. The resulting line roughly predicts the attenuation factor as $k_{T,ISS} = 0.875$ with $R^2 = 0.93$ and a standard deviation of $\sigma = 0.0054 \frac{m}{s}$ based on evenly weighted data points (traditional least squares). The low value of k_T suggests that the on-orbit thrust performance was significantly lower for this test session.

The result on its own is not meaningful without comparing to simulation results where the true thruster performance is known. Figure shows a scatter plot of the same matches executed in simulation overlaid with the ISS results after iteratively adjusting the parameter controlling the thruster strength until the fit lines match. The required scaling ratio was

$$F_{thrust} = 0.847 \cdot 0.112N,$$

to achieve an equivalent attenuation $k_{T,sim} = 0.877$ in the simulation with $R^2 = 0.95$ and $\sigma = 0.0038$. The difference between the thruster scaling ratio and attenuation factor indicates is likely an additional component contributing to the thrust variations.[2] However, for the purposes of Monte Carlo testing, the results are close enough to use the variations in attenuation factor as a range over which to vary the thruster performance. To suggest the bounds, the attenuation factor was estimated using a least squares estimator at each time step in the ISS tests. The results are shown in Figure B.3. Most of the variation in the ISS tests is captured in the region of $0.6 \leq k_T \leq 1$ for a thruster strength of $0.112\,N$, the current simulation default. Based on this analysis, the default thrust value and parameter ranges should change to:

$$\begin{aligned} F_T &= 0.875 \cdot 0.112 = 0.098\,N \\ & 0.69 \leq k_T \leq 1.24 \end{aligned}$$

B.2.3 Metrology Errors

In addition to the noise model applied by default in the simulation, the metrology system is subject to other variations between test sessions.

Beacon Normal Misalignment Starting in Test Session 22, a bug in the SPHERES Flight GUI prevented the beacon normal vector from being calculated correctly on the ground station from readings of the two stage angles on the base of the beacons. After analyzing beacon angle information from previous test sessions, it was determined that crew members, instructed to point the beacons toward the center of the volume, usually managed to do so within about 7 degrees. This was deemed suffi-

[2]Note that we did not include variations in the thrust related to the number of open thrusters. This effect was tested but because the internal process model of the estimator does not account for changes in thruster force, and the velocity estimate is heavily weighted on this process model, so attempting to account for the thrust reduction in the commanded $\Delta \mathbf{v}$ analysis will introduce more error. While performing this analysis, it became apparent that this missing factor in the process model accounts for a significant amount of velocity error in the state estimate and should be corrected in the estimator.

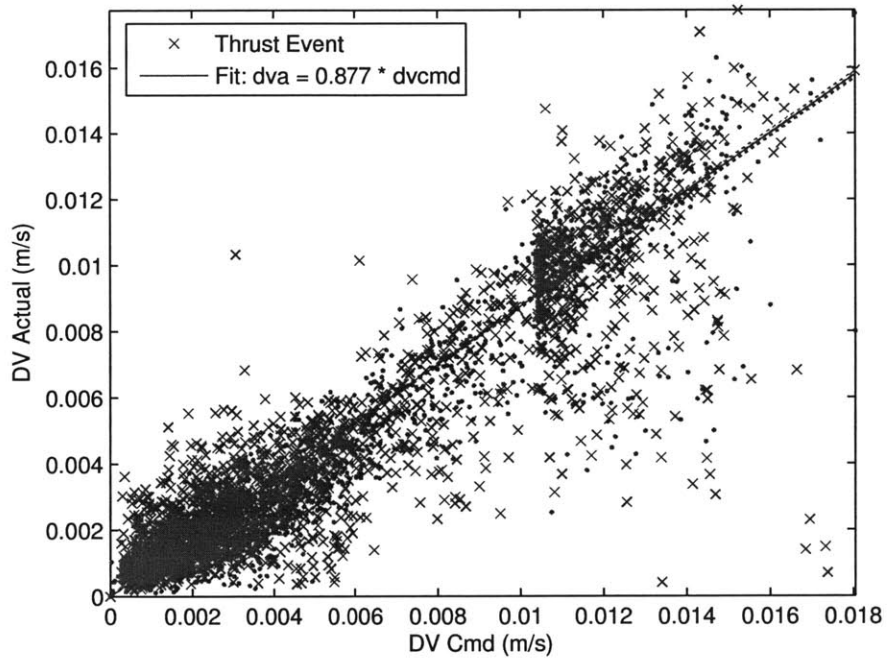


Figure B.2: Comparison of commanded velocity change to actual velocity change for simulation (red) and ISS data (blue). The level of thrust in the simulation was adjusted to achieve the same attenuation factor.

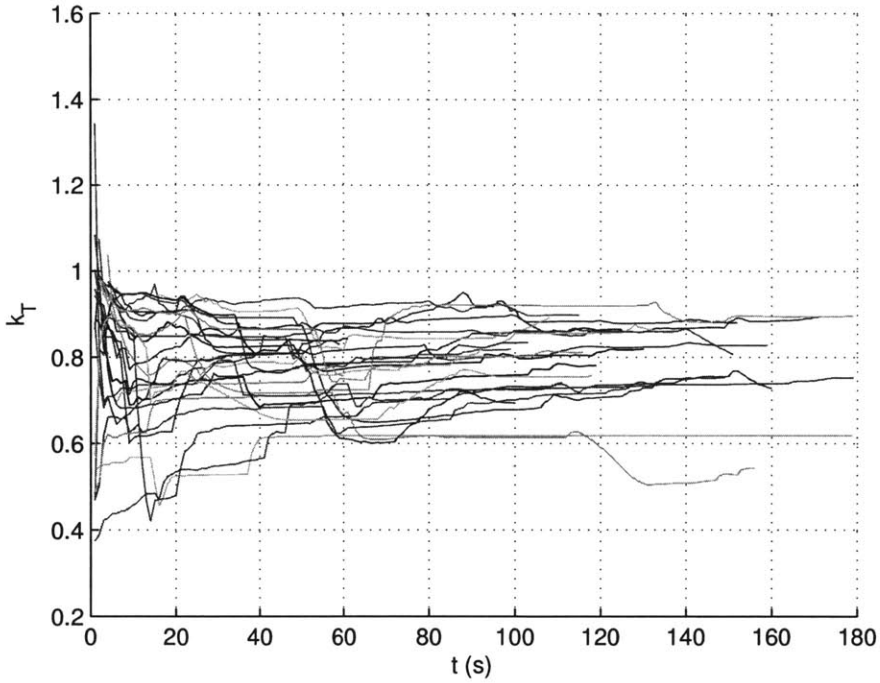


Figure B.3: The thruster attenuation factor as estimated by a least squares filter at each time step for all tests during the 2012 ISS finals. The family of time histories suggests a reasonable bound on the thruster strength variation is $0.6 \leq k_T \leq 1$ for a thruster strength of 0.112N.

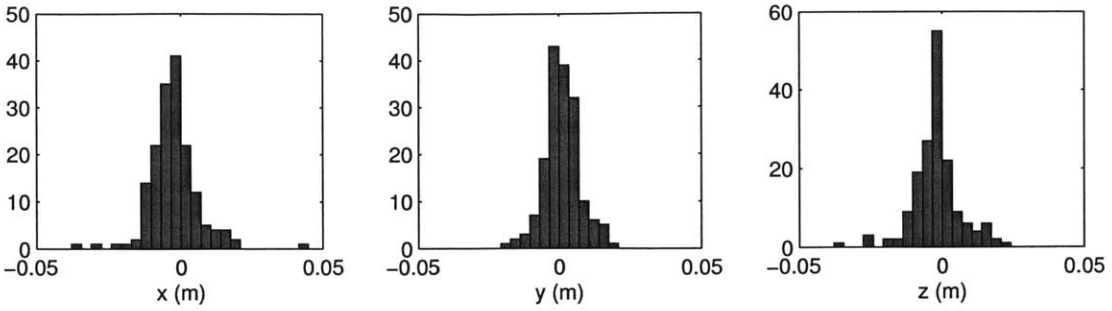


Figure B.4: The initial positioning distribution around the starting target from 168 Zero Robotics ISS matches. The standard deviations are $\sigma_{xyz} = \begin{bmatrix} 0.0085 & 0.0062 & 0.0081 \end{bmatrix}$.

ciently accurate to assume the beacons were pointed toward the center of the volume. A full Monte Carlo analysis should include the beacon misalignment as a source of potential variation.

ISS Temperature Variation Temperature affects the speed of sound in the estimator's time of flight calculations. Starting in Test Session 22, the satellite temperature has been assumed to remain constant at $21°C$. Data from sessions up to this point show that the temperature can range from $21°$ to $25°C$.

B.2.3.1 Initial Positioning

Figure B.4 shows the expected positioning error based on the estimated state values from all Zero Robotics test sessions except the 2010 Summer of Innovation finals where initial positioning time was not sufficient to achieve the targets reliably. Standard deviations on the order of 9 mm indicate that the positioning error is somewhat higher than the usually assumed 1 cm positioning accuracy of the satellites. The suggested Monte Carlo range is the 3σ positioning error.

Initial velocity errors are more problematic to include in the simulation because the 10 second estimator convergence period is still preserved to ensure the virtual estimator is stable at the start of a match. In the current simulation they will be ignored. One strategy for including velocity errors would be to model the crew's deployment errors and include an initial positioning phase in the simulation. This

Parameter	Description	Nominal	Range
m	satellite mass	$4.45\,kg$	$[4.37, 4.54]\ kg$, (true physical range)
F_T	thruster force	$0.098\,N$	(not varied, use k_T)
k_T	thruster force attenuation	1	$[0.69, 1.24]$
$(\Delta x_0,\ \Delta y_0,\ \Delta z_0)$	initial positioning error	$0\,m$	$[-0.027, 0.027]\ m$
α_{bcn}	beacon normal misalignment	$0°$	$[-7, 7]\,°$
T_{ISS}	ISS temperature	$21°C$	$[21, 25]\,°C$

Table B.2.1: Monte Carlo Parameters and Ranges

would be more accurate but would increase simulation times by about 15%. Another option would be to inject a velocity error following the estimator convergence period by directly modifying the true state of the simulation. This option would require careful study to prevent divergence of the estimator.

Appendix C

A Monte Carlo System for Open-Ended Robustness Analysis

C.1 Introduction

C.1.1 Motivation

The Zero Robotics simulation incorporates many sources of random variations, from modifications of thrust levels to sensor noise. In this way users experience variations between simulation runs and have some sense for the effects of uncertainty when developing programs. However, as highlighted by experiences with several final competitions, student implementations are not always robust to large uncertainties. This indicates that the fundamental update loop of write-test-update with the standard simulation environment does not always reflect the full scope of variations in on-orbit testing. This chapter presents a user-driven tool to better explore parameter variations.

Under *Efficient Inquiry*, there is a middle ground between using randomness to encourage robust implementations and causing frustration. Simply increasing the random variation in the simulation environment may affect the competition scoring and lead to a sense that game outcomes are decided by luck. Users see one simulation at a time, and their primary means of viewing results is through a 3D animation, so

trends in an ensemble of simulations may not be immediately clear. Further complicating analysis, all of the random components of the simulation are tied to time varying features, such as random errors in global metrology measurements, sensor noise from the inertial measurement units, and variation in thrust levels at each time step. When troubleshooting the causes of a program problem due to one of these variations, it is difficult to pinpoint the cause because the effects are spread out over the full time history of the simulation, and the random perturbations are different at every instant.

C.1.2 Requirements

The ideal solution is a dedicated tool for performing ensemble analysis of many simulations. The Leaderboard tool from Chapter 4 serves this purpose for comparing performance against other competing teams, but no component of the Zero Robotics platform is specifically targeted at assisting users in exploring how changes in parameters affect the performance of their programs. Such a tool should:

1. Highlight common variations found in ISS testing.

2. Allow users to define their own parameter ranges to vary.

3. Allow users to define custom constraints or success criteria to evaluate.

4. Provide feedback to a non-expert user about the relationship between parameter variations and success criteria.

The ability to define custom parameters and success criteria is a key requirement for ensuring the tool fulfills the principle of *Efficient Inquiry*. With this feature, users can make use of the tool in the design of their own programs to analyze a range of options for parameter tuning. The feature also complicates the implementation because the tool must be capable of answering more open-ended queries than traditional Monte Carlo systems, which are often analyzed by a topic expert.

C.2 Monte Carlo Robustness Testing

C.2.1 A Note About Parameter Sampling

One of the frequent reasons for turning to Monte Carlo simulation is the need to explore a high-dimensional parameter space where brute force gridding is not practical. When the samples are generated by high fidelity, long-running simulations even coverage of the space with a relatively small number of simulations becomes a significant concern. It is well known that uniform random sampling over the parameter ranges can lead to isolated clumps of sample points as demonstrated in Figure C.1. To capture the full parameter range, it is better to use a quasi-random sample generator also known as a *low discrepancy* generator. Popular choices include Halton [29] and Sobol [7] sequences.

This analysis uses a Halton sequence, which generates samples by reversing the digits of an integer expressed in the base of a prime number. Each dimension uses the next available prime number as a base. Better uniformity over high dimensions is achieved by choosing a *leap* factor for the sample set that skips L samples from the set at each draw. One method of choosing an appropriate leap factor is to choose L greater than all the prime bases, or in other words greater than the $(s+1)^{st}$ prime, where s is the dimension of the parameter space. A specific type of Halton sequence called a Hammersley set [30] chooses a better distribution over the parameters if the total number of samples is known *a priori*. Halton sets are used here because additional samples may be drawn during the Monte Carlo process.

C.2.2 Overview of Method

The analysis tools presented here draw heavily on the framework introduced by Crespo, Kenny, and Giesy in [16], hereafter referred to as Crespo. This work shares the goal of analyzing controller robustness by examining the outputs of high fidelity simulations. The approach is not specific to Monte Carlo testing, but the implementation in a package called UQTools for MATLAB includes support for results generated by

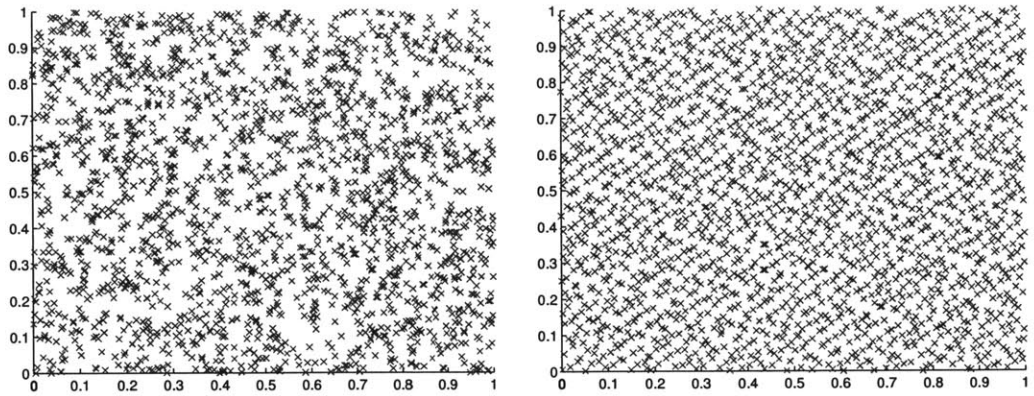


Figure C.1: (a) Uniform random sampling (2000 points) over the parameter space leads to clumps of data points. (b) Quasi-random sampling (2000 points, Halton, $L = 12$) tends to fill the parameter space more evenly.

random sampling. The Monte Carlo approach presented here parallels the UQTools approach for deterministic parameter models with several enhancements noted in the successive sections. The remainder of this section is a brief summary of Crespo's approach. It should be noted that the present discussion omits the extensive study of failure probability analysis that leverages the same techniques. These advanced methods might be appropriate for a future enhancement of the tools presented here.

The robustness analysis starts with the definition of a parameter vector $\mathbf{p}$ and one or more requirements functions, expressed in the form of a constraint $\mathbf{g}(\mathbf{p}) < \mathbf{0}$. The constraint function does not have to be explicitly specified and may be based on the processed output of a simulation. Crespo defines the range of parameter values as a hyper-rectangular set of the form

$$\mathcal{R}(\bar{\mathbf{p}}, \mathbf{m}) = \{\mathbf{p} : \bar{\mathbf{p}} - \mathbf{m} \leq \mathbf{p} \leq \bar{\mathbf{p}} + \mathbf{m}\}, \tag{C.1}$$

where $\mathbf{m} > \mathbf{0}$ are the half-lengths of the sides and $\bar{\mathbf{p}}$ are the nominal parameter values. Parameter robustness testing involves characterizing the size of the region where the constraint function is satisfied and comparing it to the size of the original parameter set. To aid this analysis, Crespo defines the concept of a *Homothetic Deformation* of

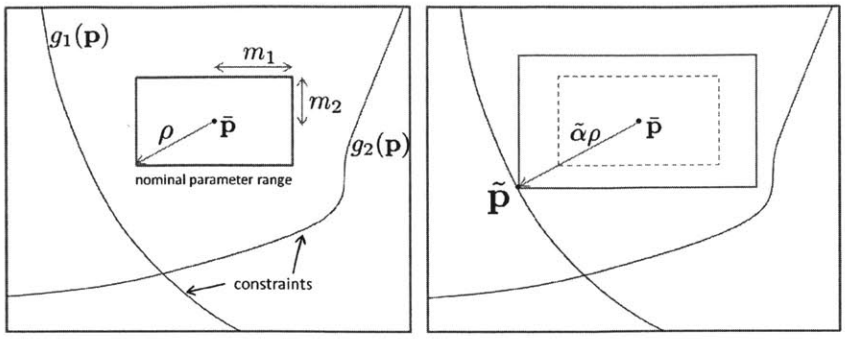


Figure C.2: A *Homothetic Deformation* expands or contracts a set with respect to its center $\bar{\mathbf{p}}$ by the similitude ratio α. To explore robustness we wish to find the largest deformation $\tilde{\alpha}$ such that the parameter region is fully contained within the constraints. The corresponding intersection point is the *Critical Parameter Value,* $\tilde{\mathbf{p}}$.

a set Ω as

$$\mathcal{H}(\Omega, \alpha) = \{\bar{\mathbf{p}} + \alpha(\mathbf{p} - \bar{\mathbf{p}}) : \mathbf{p} \in \Omega\}. \tag{C.2}$$

The *Similitude Ratio,* $\alpha \geq 0$, is a parameter that expands or contracts the reference set with respect to the center point $\bar{\mathbf{p}}$. This can be visualized as shrinking or growing the parameter set equally around its center point. We wish to find the largest homothetic deformation such that the deformed parameter space is fully contained within the constrained region. If the largest set, called the *Maximal Set,* $\mathcal{M}_{\mathbf{p}}$, corresponds to $\alpha \geq 1$, then the full expected parameter range is contained within the constraints, and the system is deemed to be robust. The value of α that produces $\mathcal{M}_{\mathbf{p}}$, also called the *Critical Similitude Ratio* (CSR), $\tilde{\alpha}$, is a measure of the robustness. The parameter value at the intersection of the constraint function and the maximal set is referred to as the *Critical Parameter Value* (CPV), $\tilde{\mathbf{p}}$. The CPV is not necessarily unique as the parameter set may intersect with the constraint function at multiple locations. Figure C.2 illustrates these quantities.

In the baseline approach, the critical parameter value is determined by solving the nonlinear optimization problem

$$\tilde{\mathbf{p}} = \arg\min_{\mathbf{p}} \{\|\mathbf{p} - \bar{\mathbf{p}}\|_{\mathbf{m}}^{\infty} : \mathbf{g}(\mathbf{p}) = \mathbf{0}\} \tag{C.3}$$

where $\|\mathbf{x}\|_{\mathbf{m}}^{\infty} \triangleq \max_{x_i} \frac{|x_i|}{m_i}$, is the m-scaled infinity norm. For better compatibility with numerical solvers, we can redefine the variable $\mathbf{p}$ as a normalized version $\mathbf{p}' = \text{diag}(\mathbf{m})^{-1}(\mathbf{p} - \bar{\mathbf{p}})$ and re-cast the optimization with a linear cost function involving α:

$$\langle \tilde{\mathbf{p}}', \tilde{\alpha} \rangle = \arg\min_{\alpha, \mathbf{p}'} \{\alpha : \mathbf{g}(\mathbf{p}') \geq \mathbf{0}, -\alpha \leq \mathbf{p}' \leq \alpha\} \tag{C.4}$$

After a critical parameter value is identified, additional samples of the constraint function may be taken around the surface of the maximal set to better characterize the constraint values. Several iterations of sampling and optimization will help to ensure a valid value has been found. The final value and corresponding simulation are highlighted to the user as a potential problem spot.

Two major caveats are important to highlight. First, the function $\mathbf{g}(\mathbf{p})$ is an arbitrary nonlinear constraint function, which means a gradient solver may converge to a local minimum before finding the true parameter boundary. Second, since $\mathbf{g}(\mathbf{p})$ is the output of a simulation, evaluations of the constraint function are very expensive. The local minimum issue can be partially addressed by additional sampling after a CPV is located, and careful choice of the constraint function can also help, though in general the choice should be left to the user to define constraints. The computational issues are discussed next.

C.2.3 Response Surface Fitting

C.2.3.1 Fitting with Radial Basis Functions

To reduce the cost of evaluating the constraint function, the approach in UQTools is to perform an initial Monte Carlo run, sampling the parameter space with a quasi-random generator, then fit a *response surface* to the resulting constraint values for each component of $\mathbf{g}(\mathbf{p})$. The response surface is a nonlinear function approximator that acts as a surrogate for the constraint function during the optimization process.

UQTools supplies a toolbox of Radial Basis Function (RBF) approximators, which

are of the form

$$f(\mathbf{x}) = \sum_i c_i \varphi\left(\|\mathbf{x} - \mathbf{x}_i\|\right), \tag{C.5}$$

where the function φ is a *kernel* that operates on the radius between the function input $\mathbf{x}$ and a feature point $\mathbf{x}_i$. c_i are the weights corresponding to each data point. A common kernel function is a Gaussian of the form

$$\varphi(r) = e^{-\epsilon r^2}. \tag{C.6}$$

The parameter ϵ controls the width over which each kernel function has influence on the overall function output.

RBFs can be used as generic function interpolators to exactly represent the function at the data points i by solving the linear system

$$\mathbf{Ac} = \mathbf{f}, \tag{C.7}$$

for the coefficient vector $\mathbf{c}$, where $\mathbf{A}$ is a matrix of kernel functions evaluated at the points $\mathbf{x_i}$ and $\mathbf{f}$ is the set of interpolation points. This is the approach suggested by UQTools, but initial experimentation with Gaussian RBFs showed that with noisy data from stochastic simulations, the approximation becomes increasingly "bumpy" as the approximation attempts to match every data point. The high variability of the surface with many local minima makes for a poor landscape over which to perform the optimization. The issues are compounded with a large number samples, where it is known that the linear system in Equation C.7 becomes ill-conditioned.

One approach to rectifying the problem is to use an alternative RBF fitting approach called Iterated Approximate Least Squares (AMLS) [22]. Instead of attempting to fit the interpolant points exactly, the algorithm performs an iteration that has the solution of the linear system above as its limit. The coefficient vector is computed as

$$\mathbf{c} = \sum_{i=0}^{n} (\mathbf{I} - \mathbf{A})^k \mathbf{f}.$$

If and only if $\|\mathbf{I} - \mathbf{A}\|_2 < 1$, the series converges to

$$\sum_{i=0}^{\infty} (\mathbf{I} - \mathbf{A})^k = \mathbf{A}^{-1},$$

though in practice it is stopped well before it approaches $\mathbf{A}^{-1}$. The norm condition on $\mathbf{A}$ can be satisfied for a family of kernel functions known as *Laguerre-Gaussians* (see [22]). A slightly modified version of Equation C.6 is a Laguerre-Gaussians:

$$\varphi(r) = \frac{\epsilon^s}{\sqrt{\pi^s}} e^{-\epsilon^2 r^2/h^2}$$

where s is the dimension of the parameter space, and h is a fill-distance $h = 1/\left(n^{1/s} - 1\right)$ for n points. The stopping point of the iteration is determined by running a process called at each iteration *Leave-One-Out Cross Validation* (LOOCV) and stopping the iteration when the change in error falls below a specified threshold. An outer line search optimization can be used to select the optimal shape parameter ϵ using the same error metric from the terminal point of the iteration. For more details, see [23].

AMLS serves to significantly smooth the surface in the presence noisy data, but it still encounters problems when the number of data points grows large. With careful pruning of the data points it may be possible to improve the performance of AMLS to robustly fit the response surface for a wide variety of cases. However, a preferable method is described next.

C.2.3.2 Fitting with Support Vector Regression

The optimization in Equation C.4 is primarily concerned with shape of the constraint at the transition point $\mathbf{g}(\mathbf{p}) = \mathbf{0}$. The remaining part of the constraint function should simply be a guide for the optimizer to locate the constraint boundary. Viewed from another perspective, we wish to use boundary line as a separator that divides the data points into two classes, one containing points that meet the constraints, and another that does not. Such a problem is the motivation behind many machine learning algorithms.

In particular, the Support Vector Machine (SVM) [14] shares similarities with the RBF fitting problem. A nonlinear SVM uses an RBF approximator to create a classification function separating input data into two or more classes. The fitting process, also called training, involves an optimization over labeled data points. In contrast to an RBF interpolator, the SVM has the goal of minimizing the RBF coefficients while simultaneously minimizing the level of misclassification. This tends to result in a surface that is as "flat" as possible within the tolerances defined for misclassification. The "support vectors" in the SVM are important data points identified by the optimization that help define the decision boundary. Due to the objective of minimizing the weights, the number of vectors is often smaller than the number of data points.

An initial approach is to simply train an SVM using the simulation data points, labeling points as

$$y_i = \begin{cases} 1 & g(\mathbf{p}_i) > \mathbf{0} \\ -1 & g(\mathbf{p}_i) < \mathbf{0} \end{cases},$$

then use the resulting RBF as an approximation of the constraint function. Due to the abrupt transition between failure and success, this method tends to discard important information about the shape of the decision boundary. Also if noisy simulation results place some valid simulations well into the infeasible region, they will be evenly weighted with failed simulations, regardless of the level of violation. This will push the boundary line farther into the infeasible region.

A related form of SVM is the Support Vector Regressor (SVR). SVRs span the gap between SVMs and RBF interpolators by attempting to approximate function values while maintaining the objective of minimizing the RBF coefficients. One version, known as $\epsilon - \text{SVR}$, attempts to minimize approximation error only outside of an ϵ deviation from the fitting surface and otherwise keep the surface as "flat" as possible [66]. This formulation is ideal for the problem at hand because it produces a relatively smooth function for optimization purposes while attempting to minimize approximation error to the full set of samples.

The last problem to solve is adding an additional level of conservatism to the

standard training procedure. The class of SVMs under consideration are called Soft-Margin SVMs because they tolerate a level of misclassification to reduce over-fitting. For the purpose of robustness characterization, these misclassifications are failed simulations that fall incorrectly into the valid parameter region. An additional ad-hoc procedure is required to compensate:

1. Train the $\epsilon - \text{SVR}$ with a very small value of ϵ. This will tend to result in a response surface well outside of the maximal set with many parameter points predicted to be violations even though the constraint function evaluation indicates otherwise.

2. For all predicted values $\hat{\mathbf{g}}(\mathbf{p}) > \mathbf{0}$ where $\mathbf{g}(\mathbf{p}) < 0$, discard the falsely identified parameters. Repeat until no more false predictions are made.

This procedure should generally reduce the number of false predictions. In a stochastic simulation it will be nearly impossible to cleanly separate the boundary because small perturbations can easily cause a constraint violation. Once the critical parameter value is identified, additional sampling around the edge of the maximal set can help better characterize the region with additional data points.

C.2.4 Choosing a Constraint Function

For the best fitting performance it is helpful to choose a constraint function with a range of $[-1, 1]$ and distinct transition between positive and negative values. One way to achieve this behavior with a general input is to pass the values through the hyperbolic tangent function

$$g(\mathbf{p}) = \tanh(f(\mathbf{p})).$$

A function with this shape will anchor the failure and and success regions while providing slope information at the boundary.

C.2.5 Additional Implementation Considerations

Parameter Scaling Because the optimization in Equation C.4 normalizes the parameters, it is not necessary to re-scale parameters in the optimizer. This also produce better fitting results for the SVM.

Even Sampling with Constraints In some cases, there are limits to parameters either due to physical constraints such as positive mass, or to *a priori* known ranges. To support these constraints in the sampling method, it is possible to simply discard parameters that fall outside of the constraint limits. Due to the even sampling properties, the remaining region will continue to be filled in uniformly. When performing the CPV optimization, the upper and lower bounds must be added as inequality constraints in the optimization

$$\langle \tilde{\mathbf{p}}', \tilde{\alpha} \rangle = \underset{\alpha, \mathbf{p}'}{\arg\min} \left\{ \alpha : \mathbf{g}(\mathbf{p}') \geq \mathbf{0}, -\alpha \leq \mathbf{p}' \leq \alpha, \mathbf{A}\mathbf{p}' \leq \mathbf{b} \right\}.$$

C.2.6 Multi-Dimensional Data Display with Parallel Coordinates

For *Efficient Inquiry*, it will be important for students to be able to concisely review the results of Monte Carlo simulations. For several varied parameters, trends in the higher dimensions are often difficult to discern. One particularly attractive approach for Monte Carlo data visualization is the parallel coordinates method. In this method, each dimension of the data set is displayed as a separate column on a common scale. Specific parameter set realizations are traced out as lines that connect the variable columns on the plot. By grouping the lines and adding emphasis such as color data, overall trends become more clearly defined. Figure C.3 shows an example for a Monte Carlo simulation for a 2D circle-tracking satellite, where the angular velocity, circle radius and thruster saturation were varied. Simulations were considered to be a failure if the steady state error did not converge to within 3 cm in 60 seconds. From the figure, we can see immediately, that the dominant factor producing failed simulations

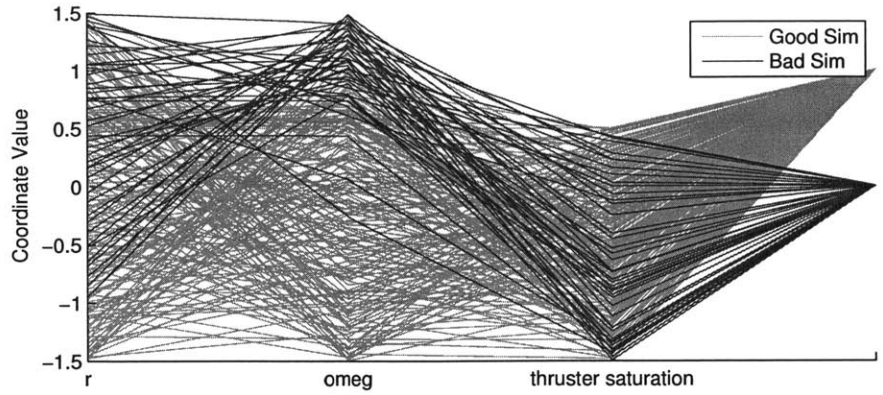


Figure C.3: Parallel Coordinates Example

is the angular velocity, while thruster saturation can become problematic at large radii.

When displaying on a web page, JavaScript data visualization libraries, such as D3.js[1], can add a useful element of interactivity to the display. In parallel coordinates, users can explore the data set by constraining parameters to specific ranges. This eliminates some of the clutter form the bulk of the set and allows new trends to surface. Also, when changing constraints the effects of changing parameters are perceived through progressive transitions, making incremental effects of the changes more visible.

C.2.7 Algorithm Summary

The complete procedure for identifying a critical parameter value follows:

1. Draw n random samples of $\mathbf{p} \in \mathbb{R}^s$ from a Halton set.

2. Run simulations for each of the samples.

3. Evaluate the constraint function $\mathbf{g}(\mathbf{p})$ based on the results of the simulations.

4. Fit a the constraint values with an $\epsilon - SVR$, and perform the steps to improve robustness of fit.

[1]See http://syntagmatic.github.io/parallel-coordinates/ for an example of parallel coordinates implemented in JavaScript

5. Perform the parameter optimization over $\mathbf{p}$ using the trained model of $\mathbf{g}(\mathbf{p})$.

6. Optionally: perform additional samples around the critical parameter point to find more accurate critical values.

C.2.8 Phased Deployment to Zero Robotics Platform:

Given the significant scale of implementing a Monte Carlo system on the Zero Robotics platform, here are several steps to incrementally deploy the functionality in manageable steps:

1. Enable users to run a batch of simulations over user-defined parameters. Simply show the list of resulting simulations.

2. Show parallel coordinates plot for exploration of raw simulation results.

3. Fit user-defined cost functions to response surfaces and use the simplified response model for rapid data exploration.

4. Add critical value optimization to highlight specific simulation instances and determine robustness metrics.

Bibliography

[1] Michael S Andrle and John L Crassidis. Geometric Integration of Quaternions. *Journal of Guidance, Control, and Dynamics*, pages 1–6, February 2013.

[2] Arduino.cc. Arduino API Reference, 2013.

[3] Albert Bandura. Self-efficacy: Toward a unifying theory of behavioral change. *Psychological Review*, 84(2):191–215, 1977.

[4] Albert Bandura. *Social foundations of thought and action: a social cognitive theory.* Prentice-Hall, 1986.

[5] M Bong. Predictive utility of subject-, task-, and problem-specific self-efficacy judgments for immediate and delayed academic performances. *The Journal of experimental education*, 70(2):133–162, 2002.

[6] Ralph Allan Bradley and Milton E Terry. Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons. *Biometrika*, 39(3/4):pp. 324–345, 1952.

[7] Paul Bratley and Bennett L Fox. Algorithm 659: Implementing Sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 14(1):88–100, March 1988.

[8] Louis Breger and Jonathan P How. Safe Trajectories for Autonomous Rendezvous of Spacecraft. In *AIAA Guidance, Navigation and Control Conference and Exhibit*, Keystone, Colorado, August 2006.

[9] Allen Chen. Propulsion System Characterization for the {SPHERES} Formation Flight and Docking Testbed. Master's thesis, Cambridge, MA, June 2002.

[10] H. Choset, R. Knepper, J. Flasher, S. Walker, A. Alford, D. Jackson, D. Kortenkamp, R. Burridge, and J. Fernandez. Path planning and control for AERCam, a free-flying inspection robot in space. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, number May, pages 1396–1403. IEEE, 1999.

[11] Committee on Science, Engineering and Public Policy. Rising above the gathering storm. 2005.

[12] Committee on Science, Engineering and Public Policy. Rising above the gathering storm, revisited: Rapidly approaching Category 5. 2010.

[13] James O. Coplien. Curiously Recurring Template Patterns. *C++ Report*, February 1995.

[14] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[15] Rémi Coulom. Whole-history rating: A bayesian rating system for players of time-varying strength. *Computers and games*, 2008.

[16] Luis G Crespo, Sean P Kenny, and Daniel P Giesy. A computational framework to control verification and robustness analysis. 2010.

[17] Pierre Dangauthier, Ralf Herbrich, Tom Minka, Thore Graepel, and Others. Trueskill through time: Revisiting the history of chess. *Advances in Neural Information Processing Systems*, 20:337–344, 2007.

[18] Beman Dawes, David Abrahams, and Rena Rivera. Boost C++ Libraries.

[19] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode : Enforcing Alias Analysis for Weakly Typed. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, New York, USA, 2006. ACM.

[20] Arpad E. Elo. *The Rating of Chess Players, Past and Present.* Arco Pub., New York, NY, 1978.

[21] John P Enright. *A Flight Software Development and Simulation Framework for Advanced Space Systems.* Phd, Massachusetts Institute of Technology, 2002.

[22] Gregory E Fasshauer and Jack G Zhang. Iterated approximate moving least squares approximation. In *Advances in Meshfree Techniques*, pages 221–239. Springer, 2007.

[23] Gregory E Fasshauer and Jack G Zhang. On choosing "optimal" shape parameters for RBF approximation. *Numerical Algorithms*, 45(1-4):345–368, 2007.

[24] FIRST. Rebound Rumble 2012 FRC Game Manual, http://www.usfirst.org/roboticsprograms/frc/2012-competition-manual-and-related-documents, 2012.

[25] David J. Gates. Properties of a Real-Time Guidance Method for Preventing a Collision. *Journal of Guidance, Control, and Dynamics*, 32(3):705–716, May 2009.

[26] Mark E Glickman. Paired Comparison Models with Time-Varying Parameters, 1993.

[27] Mark E Glickman and Shane T Jensen. Adaptive paired comparison design. *Journal of statistical planning and inference*, 127(1):279–293, 2005.

[28] Max Goldman, Greg Little, and Robert C. Miller. Real-Time Collaborative Coding in a Web IDE. In *UIST'11*, Santa Barbara, CA, 2011.

[29] John H Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 2(1):84–90, 1960.

[30] John M Hammersley. Monte Carlo methods for solving multivariable problems. *Annals of the New York Academy of Sciences*, 86(3):844–874, 1960.

[31] Ralf Herbrich, Tom Mnka, and Thore Graepel. TrueSkill: A Bayesian Skill Rating System. In *Advances in Neural Information 20*. MIT Press, 2007.

[32] Mark O Hilstad. A Multi-Vehicle Testbed and Interface Framework for the Development and Verification of Separated Spacecraft Control Algorithms. Master's thesis, Cambridge, MA, June 2002.

[33] Margaret A Honey and Margaret Hilton. *Learning Science Through Computer Games and Simulations*, volume 42. The National Academies Press, 2011.

[34] D Hsu, R Kindel, J.-C. Latombe, and S Rock. Randomized Kinodynamic Motion Planning with Moving Obstacles. *The International Journal of Robotics Research*, 21(3):233–255, March 2002.

[35] Http://www.bestinc.org. BEST Robotics: Middle and high school robotics competition.

[36] Http://www.botball.org. BotBall Educational Robotics Program.

[37] Http://www.usfirst.org. FIRST Robotics.

[38] David R Hunter. MM Algorithms for Generalized Bradley-Terry Models. *The Annals of Statistics*, 32(1):pp. 384–406, 2004.

[39] A. R. Johnson. International Space Station: National Laboratory Education Concept Development Report. Technical report, NASA, 2006.

[40] Jacob G Katz. *Estimation and control of flexible space structures for autonomous on-orbit assembly.* Master's, Massachusetts Institute of Technology, 2009.

[41] Firas Khatib, Seth Cooper, Michael D Tyka, Kefan Xu, Ilya Makedon, Zoran Popović, David Baker, and Foldit Players. Algorithm discovery by protein folding game players. *Proceedings of the National Academy of Sciences*, 108(47):18949–18953, 2011.

[42] S. M. LaValle. Randomized Kinodynamic Planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001.

[43] William A. Lucas, Sarah Y. Cooper, Tony Ward, and Frank Cave. Industry placement, authentic experience and the development of venturing and technology self-efficacy. *Technovation*, 29(11):738–752, November 2009.

[44] James E Lumpp, Daniel M Erb, Twyman S Clements, Jason T Rexroat, and Michael D Johnson. The CubeLab Standard for Improved Access to the International Space Station. In *Aerospace Conference, 2011 IEEE*, pages 1–6. IEEE, 2011.

[45] MATE. Marine Advanced Technology (MATE) Underwater Robotics Competitions, http://www.marinetech.org/rov-competition/.

[46] MATLAB/Simulink. *version 8.0.0 (R2012b)*. The MathWorks Inc., Natick, Massachusetts, 2012.

[47] Grant McDonald and Jennifer Lay. Social Networking In FIRST Robotics, 2010.

[48] Alan Melchior, Faye Cohen, Tracy Cutter, and Thomas Leavitt. More than Robots : An Evaluation of the FIRST Robotics Competition Participant and Institutional Impacts Heller School for Social Policy and Management. Technical Report April, Brandeis University, Walthma, MA, 2005.

[49] David P Miller, Illah R Nourbakhsh, and Roland Siegwart. Robots for Education. In *Springer Handbook for Robotics*, pages 1283–1301. Springer, 2008.

[50] MIT Space Systems Laboratory. SPHERES ISS Test Session 18. Technical report, 2009.

[51] MIT Space Systems Laboratory. SPHERES ISS Test Session 21. Technical report, Massachusets Institute of Technology, 2010.

[52] MIT Space Systems Laboratory. SPHERES ISS Test Session 33. Technical report, 2012.

[53] Sreeja Nag. *Collaborative Competition for Crowdsourcing Spaceflight Software and STEM Education using SPHERES Zero Robotics*. Master's thesis, Massachusetts Institute of Technology, 2012.

[54] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, October 2003.

[55] Simon Nolet. *Development of a Guidance, Navigation and Control Architecture and Validation Process Enabling Autonomous Docking to a Tumbling Satellite*. Sc.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2007.

[56] Simon Nolet, Alvar Saenz-otero, David W Miller, and Amer Fejzic. SPHERES Operations Aboard the ISS: Maturation of GN&C Algorithms in Microgravity. In *30th Annual AAS Guidance and Control Conference*, Breckenridge, Colorado, February 2007.

[57] William H Press, Brian P Flannery, Saul A Teukolsky, and William T Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. October 1992.

[58] Andrew Radcliffe. A Real-Time Simulator for the SPHERES Formation Flying Satellites Testbed. Master's, Massachusetts Institute of Technology, Cambridge, MA, June 2002.

[59] Casey Reas and Ben Fry. *Getting Started with Processing*. O'Reilly Meida, Inc., 2010.

[60] CW Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference*, pages 763–782, San Jose, California, 1999. Miller Freeman Game Group.

[61] Arthur George Richards. Trajectory Optimization using Mixed-Integer Linear Programming. Master's thesis, Cambridge, Massachusetts, June 2002.

[62] Alvar Saenz-Otero. The SPHERES Satellite Formation Flight Testbed: Design and Initial Control. Master's thesis, Cambridge, MA, August 2000.

[63] Alvar Saenz-Otero. *Design Principles for the Development of Space Technology Maturation Laboratories Aboard the International Space Station*. Phd thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2005.

[64] Alvar Saenz-Otero, Jacob Katz, Swati Mohan, David W Miller, and Gregory E Chamitoff. ZERO-Robotics: A student competition aboard the International Space Station. In *2010 IEEE Aerospace Conference*, pages 1–11. IEEE, March 2010.

[65] Jeanine Skorinko, Jennifer Lay, G McDonald, Brad Miller, Colleen Shaver, C. Randall, J.K. Doyle, G. Tryggvason, M. Gennert, and J. van de Ven. The Social Outcomes of Participating in the FIRST Robotics Competition Community. In *2010 ASEE Northeast Section Conference, May 7-8*, number 1, Boston, MA, 2010.

[66] Alex J Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222, 2004.

[67] Herb Sutter. Pimpls - Beauty Marks You Can Depend On. *C++ Report*, 10(5), May 1998.

[68] Tynan Sylvester. *Designing Games*. O'Reilly Meida, Inc., 2013.

[69] Texas Instruments. *TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide*. Houston, TX, 2002.

[70] Sonny Thai. *Collaborative editor environments for player programs*. M. eng., Massachusetts Institute of Technology, 2012.

[71] The Mathworks. MATLAB/Simulink Embedded Coder Documentation (v2013a), http://www.mathworks.com/help/ecoder/index.html, 2013.

[72] LLC YouTube. YouTube Space Lab Official Rules, 2011.