Toward Consistent Robotics Simulation Through Validation

by James R. Taylor

B.S. in Computer Science, May 2010, University of Memphis

A Dissertation submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Doctor of Philosophy

August 31, 2018

Dissertation directed by

Evan Drumwright
Senior Research Scientist

The School of Engineering and Applied Science of The George Washington University certifies that James Rankin Taylor has passed the Final Examination for the degree of Doctor of Philosophy as of July 23, 2018. This is the final and approved form of the dissertation.

Toward Consistent Robotics Simulation Through Validation

James R. Taylor

Dissertation Research Committee:

Evan Drumwright, Senior Research Scientist, Toyota Research Institute, Dissertation Director

Robert Pless, Professor of Computer Science, Committee Member

Bhagirath Narahari, Professor of Computer Science, Committee Member

Gabriel Parmer, Associate Professor of Computer Science, Committee Member

Paul Mitiguy, Adjunct Professor of Mechanical Engineering, Stanford University, Committe Member

Dedication

This work is dedicated to my mother, Margaret.

Abstract

Toward Consistent Robotics Simulation Through Validation

Multi-rigid body dynamics simulation supports design, prototype, test, and evolutionary stages of robotic development. However, the reliability of simulating tasks critical to robotics, *i.e.* grasping and locomotion, remains low and represents a barrier to future robotics applications. A lack of simulation reliability is typically exhibited by a simulated robot performing a task while under similar conditions the real system fails to perform that same task. Simulations that are not reliable may mislead roboticists into costly, infeasible designs or give false safety assurances of systems where safety is paramount, *e.g.* autonomous vehicles. Roboticists therefore need assurances that simulation will either consistently predict real performance or indicate that it can not make sound predictions. General software engineering proposes verification and validation as practices for quality assurance. In simulation software, verification checks that implementation is consistent with theory, and validation checks that simulation is consistent with real behaviors. While verification is critical, validation is necessary to objectively demonstrate simulation consistency.

This thesis studies validation of multi-body rigid body dynamics simulations with contact and friction in an effort to improve the state of the art in robotics simulation. This work identifies scenarios relevant to robotics tasks, examines simulators performing these tasks, and identifies ways that current simulators should be used and future simulators should be improved. Additionally, this work identifies temporal consistency as a critical requirement for robotics simulators, proposes a temporally consistent simulator architecture, and studies the performance difference between temporally consistent and existing robotics simulators. Finally, this work studies the motion of a minimally simple, yet still sophisticated, real world robot that we can model and examines a means to validate simulation of the robot.

Table of Contents

Dedication . . . . **iv**

Abstract . . . . **v**

List of Figures . . . . **ix**

List of Tables . . . . **xi**

List of Acronyms . . . . **xii**

Chapter 1- Introduction **1**

Chapter 2- Background for Robotics Simulator Validation **8**

2.1 Multi-Rigid Body Simulation Design 8
   2.1.1 Simulation Paradigm 8
   2.1.2 Coordinate Type 9
   2.1.3 Solver Type 10
   2.1.4 Collision Detection 11
   2.1.5 Quasi-Rigid DVI-Based Contact 11
   2.1.6 Modeling Specification 11
   2.1.7 Real World Constraints 12
2.2 Validation Studies of Multi-Rigid Body Simulations 14
   2.2.1 Studies of Simulation Performance 16
   2.2.2 Real Data Studies for Simulation Validation 19

Chapter 3- Comparative Evaluaton of Algorithms Using Quasi-Rigid Grasping **21**

3.1 Failure Modes for Quasi-Rigid Grasping 23
3.2 Quasi-Rigid Grasping Experimental Specifications 24
   3.2.1 Simulating Quasi-Rigid Grasping 25
   3.2.2 Quasi-Rigid Grasping Scenarios 27
   3.2.3 Measuring Quasi-Rigid Grasping 28
   3.2.4 Quasi-Rigid Grasping Experimental Controls 30
3.3 Testing Hypotheses with Quasi-Rigid Grasping Scenarios 30
   3.3.1 Hypothesis 1: Modifying Inertia Properties Could Lead to Improvements in Grasping Performance 30
   3.3.2 Hypothesis 2: Increasing the Friction Coefficient to Very Large Values Will Not Reduce Grasping Performance 31
   3.3.3 Hypothesis 3: Using Primitive Geometries in Place of Mesh Geometries Yields Improved Grasping Performance 32
   3.3.4 Hypothesis 4: Pivoting LCP Solvers Are More Effective at Simulating Grasping than Iterative Matrix Splitting Solvers 32
3.4 Comparative Analysis of Quasi-Rigid Simulation 32

3.5 Conclusions and Future Work with Grasping Scenarios 39

Chapter 4- Validating Controllers Through Temporally Consistent Simulation 40
4.1 A Contrast Between Robot and Simulation Architectures 42
4.2 Background for Temporally Consistent Simulation 45
4.2.1 Requirements for Temporally Consistent Simulation 46
4.2.2 System Scheduling Toward Temporally Consistent Simulation . . . 47
4.3 OS Facilities for Temporally Consistent Simulation 48
4.3.1 Temporal Accounting of Processes 48
4.3.2 Scheduling Simulation Processes 51
4.3.3 Communication Between Simulation Processes 52
4.4 Temporally Consistent Simulator Design 52
4.4.1 Hierarchical Scheduling and Threads 52
4.4.2 Coordinator Design 54
4.4.3 Client Process Implementation 55
4.5 Comparisons Between Conventional and Temporally Consistent Simulators 56
4.5.1 Experimental Comparison of a Pendulum Swing-Up 56
4.5.2 A Demonstration of Temporal Consistency: Predator and Prey . . 60
4.6 Future Work in Temporally Consistent Simulation 64

Chapter 5- Validating Simulations Involving Contact and Friction with a Wild Robot 66
5.1 The Weazelball is a Wild Robot 67
5.1.1 Anatomy of a Weazelball 68
5.1.2 Modeling the Weazelball 70
5.2 Measuring Weazelball State 71
5.2.1 Estimating Weazelball Shell State Using Vicon Motion Capture 71
5.2.2 Measuring a Weazelball's Hidden Actuator State 75
5.3 Additional Instruments for Weazelball State Measurement 77
5.3.1 Calibrating the Weazelball to Improve Vicon Tracking 77
5.3.2 Enclosing the Weazelball During Motion Tracking 80
5.3.3 Visualizing Weazelball State Data in Gazebo 82
5.4 Capturing Weazelball State 85
5.4.1 Registering the Weazelball Shell with Vicon 85
5.4.2 Capturing Weazelball Shell State Using Vicon 86
5.4.3 Video Recording Hidden Weazelball Actuator State 87
5.5 Estimating Weazelball State 88
5.5.1 Synchronizing Vicon Data and Video Data 88
5.5.2 Qualitative Assessment of Weazelball Behavior and Actuator Data 89
5.5.3 Identifying Error in Weazelball State Data 91
5.5.4 Correcting Weazelball Orientation Error in Vicon Data 91
5.5.5 Correcting Weazelball Position Error in Vicon Data 93
5.5.6 Estimating Weazelball Actuator State 98

5.6 Validating WEAZELBALL Simulation Through Support Vector Regression 100
5.6.1 Simulating the WEAZELBALL in GAZEBO 104
5.6.2 Generating Training Data from WEAZELBALL Simulation 105
5.6.3 Training the Regression Model 107
5.6.4 Analysis of Regression 108
5.6.5 Conclusions and Future WEAZELBALL Work 109

References **115**

List of Figures

1 Continuum of Simulation Realism 3

2 Formal Relationships Between Verification and Validation . . 5

3 Simulated Grasping Robots 22

4 Industrial Arm Grasp Control Analysis 33

5 Multiple Block Grasp Control Analysis 34

6 Industrial Arm Inertial Modification Analysis 35

7 Multiple Block Friction Analysis 36

8 Multiple Block Solver Analysis 38

9 Processing Time for a Large Number of Collisions 41

10 Architecture of a Physically Situated Robotic System 43

11 Architecture of a Typical Simulated Robot 44

12 Temporal Comparsion of Conventional and Temporally Consistent Simulation 58

13 Instantaneous Temporal Disparity of Conventional Simulation 58

14 Aggregate Relative Error in Temporally Consistent Simulation 59

15 Instantaneous Distance Between Predator and Prey 63

16 The Weazelball 67

17 Interior View of the Weazelball 69

18 The Weazelball as modeled by Solidworks 70

19 Vicon Configuration For Weazelball Motion Capture 72

20 The Weazelball Signal Circuit 76

21 Stand Placement in Registration Array 78

22 Illustration of the Stand Alignment Procedure 79

23 Front View of the Weazelball Mounting Process 81

24 Illustration of the Tracking Model Reference Frame 82
25 Enclosure Model 83
26 Illustration of the Motion Capture Configuration 84
27 Releasing the Weazelball 87
28 Synchronized Weazelball Data 89
29 Error Analysis of Weazelball Motion Capture Data 92
30 Interpolated Weazelball Actuator Frequency 99
31 Trajectory from a Situated Weazelball 101
32 Offset Varied Weazelball Trajectories Simulated by ODE . . . 102
33 Offset Varied Weazelball Trajectories Simulated by DART . . 103
34 Simulated Weazelball Trajectories with Varied Integration Step105
35 Regression Analysis for 30s of Weazelball Simulation 111
36 Regression Analysis for 120s of Weazelball Simulation 112
37 Regression Analysis for 300s of Weazelball Simulation 113

List of Tables

1 Process Priority Assignment. 55

2 HD video LED signal detection 90

3 Frequency of Vicon orientation snap change error events 94

4 Regression error on training and testing data sets 110

List of Acronyms

CoM— Center-of-Mass

DARPA— Defense Advanced Research Projects Agency

DART— Dynamic Animation and Robotics Toolkit

DVI— Differential Variational Inequality

HD— High Definition

IMU— Inertial Measurement Unit

IR— Infra-Red

LCP— Linear Complimentarity Problem

LED— Light-Emitting Diode

ODE— Open Dynamics Engine

OMPL— Open Motion Planning Library

OS— Operating System

OSRF— Open Source Robotics Foundation

PAL— Physics Abstraction Layer

POSIX— Portable Operating System Interface

ROS— Robot Operating System

RTOS— Real-Time Operating System

SDF— Simulation Description Format

URDF— Unified Robot Description Format

Wb— Weazelball

Chapter 1- Introduction

This thesis examines validation as a means to improve the predictive consistency of robotics simulation. This examination is carried out using a number of scenarios and metrics relevant to robotic systems and relies on comparisons between simulations to draw conclusions. This work demonstrates how validation can facilitate assessment of the algorithms, models, and architectures used in robotics simulation, identifies temporal consistency as a requirement for robotics simulation and describes a temporally consistent architecture, and proposes a validation approach that accommodates complex robotic systems and allows for fair comparisons among simulations. This work also provides a comprehensive study of data collection and simulation validation involving a real world robot and provides this data for the validation of other simulators.

Brooks and Mataric [9] define three abstract yet fundamental problems associated with robotics simulation:

1. Simulations are doomed to succeed

2. Simulations cannot be made sufficiently realistic

3. Many simulation artifacts do not exist in the real world

The first point argues that a simulation can be made to produce a plausible result, but simulated behavior may not predict real system behavior. For example, simulation of a robot may produce a walking or climbing behavior where the real robot falls instead [69]. The second point argues that simulations rely on models which are approximations based on provisional physical understanding and technological limitations. For example, robotic simulators model volumes as unbreakable, rigid bodies to facilitate fast computation, yet volumes composed of physical matter are not unbreakable. The third point is an observation that physical laws cannot be violated in the real world; however, unrealistic behaviors will arise in simulations due to the second point. For example, simulated rigid bodies may interpenetrate one another, *i.e.* occupy the same volume, where solids in the real world cannot occupy the same volume. In spite of these problems, roboticists rely on simulation to design, model and predict the performance of robots participating in interactive tasks.

Robotic applications are primarily interactive in nature. An ultimate goal of robotics is to safely assist humans performing physical tasks in human environments [62]. For a robot to be a fully cooperative partner, a robot must be able to move through the environment, *i.e.* locomotion, and manipulate objects in the environment, *i.e.* grasping, without failure. Locomotion and grasping are contingent upon contact and the resultant friction forces; therefore, robotics requires simulations that model contact and friction consistent with the contact and friction experienced by real systems. Unfortunately, simulations involving contact and friction have not been demonstrated to be consistent enough with reality to be reliably predictive.

Realism might be represented as a continuum ranging from unrealistic to realistic, refer to Figure 1, and every simulator exists somewhere along this continuum. Closer to the unrealistic end of the continuum lies a point where simulation is *realistically plausible* while closer to the realistic end lies a point where simulation is *qualitatively predictive*. Realistically plausible simulations are visually convincing to users, but plausible simulations do not necessarily model or simulate a real system with enough fidelity to predict real performance with a reasonable amount of accuracy. Plausible simulations often avoid modeling some aspects that are non-trivial considerations or they substitute simplified, unrealistic models for complex computations to guarantee real-time performance. Qualitatively predictive simulations do not provide perfect fidelity but the results that they produce are qualitatively similar enough to real world system performance that simulated behaviors are indistinguishable at a high level from and consistent with real world behaviors. Realistically plausible simulations may have no more predictive value than random guessing while qualitatively predictive simulations should produce feasible predictions. Decisons made by the simulator developer, such as selection of models, selection of simulation paradigm, and amount of real-time optimization, generally determine where along the continuum the simulation falls.

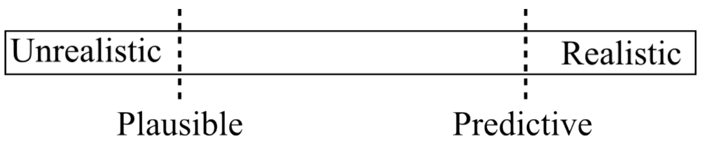


Figure 1: Continuum of Simulation Realism

Plausible marks the point on the continuum where simulation has apparant realism, *i.e.* quasi-realism. Apparant realism can mislead users to believe that simulation is realistic while the predictive value may be no better than a random guess. Predictive marks the point on the continuum where simulation is accurate enough, *i.e.* pseudo-realistic, for viable prediction.

Adding a real-time requirement to simulation constrains how realistic the simulation may be. A currently accepted approach to modeling and simulating a large class of robotic systems is multi-rigid body dynamics. Multi-rigid body dynamics is the extension of rigid body dynamics to actuated, articulated systems. Rigid body dynamics has been proven to be consistently predictive for modeling ballistics and space flight, and rigid body dynamics has also proven to be fast enough to operate at real-time on current hardware as demonstrated by computer gaming simulations. These two statements suggest that multi-rigid body dynamics is both fast enough and accurate enough for robotics simulation, but there are enough differences between these application domains that multi-rigid body dynamics robotics simulations have not been proven to be qualitatively predictive. For example, simulations of ballistics and space flight are able to ignore contact, and computer gaming simulations avoid contact and friction by generating plausible locomotion through animation using precompted trajectories and plausible grasping through fixing objects to an end effector. Game simulations generally use first order integrators that compute approximate solutions, *i.e.* forces, velocities, and positions, and integrate dynamical equations using a large time step. Reformulation of differential equations might avoid computational stiffness and potential, obvious instabilities that might develop in the simulation. Regardless, solution approximation introduces error that accumulates over the course of the entire simulation, and while the simulation appears to produce a plausible result, the accumulated error may reduce that simulation's predictive value. Game simulations may also introduce arbitrary parameters, *e.g* damping, restitution, and contact parameters, to maintain plausibility that cannot be measured from a real system and might not fit any accepted physical model. Multi-rigid body dynamics is attractive to roboticists because it is demonstrably real-time and requires relatively little system identification investment, but an open question remains as to whether there exists a multi-rigid body dynamics simulation that is consistent enough to be qualitatively predictive for simulations involving contact and friction.

Verification and validation are proposed means to assess the quality of simulation. The definitions and roles of verification and validation for simulation were formalized by committee agreement [56], refer to Figure 2. Informally, verification is checking that the implementation matches mathematical models and validation is checking the fidelity of a simulation.

Verification is a bottom-up approach to assessing simulation, which is based on the assumption that if all of the foundational models are correct, then errors in the system will be reduced. Verification does eliminate errors and inconsistencies in simulation, but analysis of models alone does not establish that a simulation as a whole has sufficient fidelity to be qualitatively predictive. A model may simply not capture all targeted phenomena and therefore may not be qualitatively predictive, so verification of that model may have no effect on the realism of the simulator. Verification is also only capable of assessing what is implemented but a model necessary for predictive simulation might be omitted from the system, so verification can not be relied

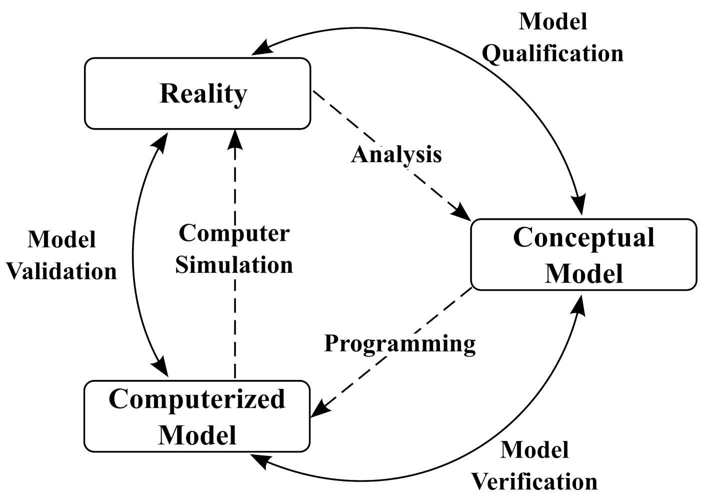


Figure 2: Formal Relationships Between Verification and Validation

The formalized diagram [56] defining the relationships and roles of simulation verification and validation. The inner arrows describe the processes which relate the elements to each other, and the outer arrows refer to the procedures which evaluate the credibilty of these processes.

upon to identify the absence of a model. Simulation is composed of a large number of individual physical models which must interact, and verification alone can not address whether the interactions between physical models has sufficient fidelity. For example, boundary modeling interacts with contact modeling, which interacts with friction modeling. Friction model verification assesses the correctness of the friction model implementation with respect to theory but does not assess the correctness of the friction model with respect to its dependency on boundary and contact model implementations.

Validation is a top-down approach to assessing simulation which is based on the assumption that comparing simulation outputs with experimental outputs gives an overall measure of the consistency between outputs. Validation alone cannot improve simulation because it does not directly identify the cause of differences between two outputs. However, if simulation is validated against real world physical experiments, measuring the fidelity of a simulation and its predictive capacity appears to be feasible.

Validation allows for the comparison of any experimental data against simulated data. To evaluate whether a simulation has sufficient fidelity, data from real world experimental studies is required. However, acquiring data from real world experiments can be problematic because not all experimental data is publicly available. Not all real world experiments are usable because many experiments do not focus enough on particular phenomena relevant to robotics and where roboticists suspect a simulation may not have sufficient fidelity. Because validation represents an analysis of the aggregate system behavior, it is difficult to identify individual components or models as sources of error within complex systems which may lead to misinterpretation. Different sources of error may also cancel each other out in particular scenarios so that a system may pass a set of validation tests[49]; however, these sources of error may still contribute error that is not cancelled in other contexts or scenarios. Real world experimental scenarios that are useful should be designed with the intent to focus study on a limited number of aspects. Identifying and measuring these scenarios can prove difficult in and of itself.

Rigid body dynamics is subject to the interaction of a number of phenomena that can not be isolated in physical tests, *i.e.* contact and friction, and surface variances are difficult to fully identify and parameterize; therefore, real world experiments intended for rigid body dynamics validation scenarios are difficult to develop and measure. Real world scenarios that have been studied for rigid body dynamics validation include the study of collisions between a steel bar and a surface [65], the motion of ball bearings on a spinning plate [42], and the contact of ball bearings dropped onto a plate [60].

While real data is critical to identifying whether simulation has sufficient fidelity, validation is not limited to using data gathered from real world experiments. Because validation is a comparative approach to measuring consistency, validation can also be performed between the output of simulations to determine how consistent simulations are with one another. If many simulators are demonstrated to consistently simulate the same task and one of those simulators is demonstrated to have sufficient fidelity, then it is reasonable to infer that all those simulators have sufficient fidelity to be qualitatively predictive. Regardless, comparisons between simulators is a necessary verification practice for scientific applications [33] and can be valuable for hypotheses testing of models applied in different simulators or for hypothesis testing of different modes that may be supported by a single simulator.

Validation may also produce indirect information about a simulator that may help researchers develop an intuition about why a simulation does not have sufficient fidelity. Simulators are complex engineering systems that are constrained by hardware and domain requirements which can lead to design compromises for better performance, *e.g.* real-time requirements. Compromises built into the core of the simulation architecture may be inherited by future generations of simulators without sufficient re-evaluation of the impact on how these designs implicitly alter modeling of the simulated systems. The bottom up approach of verification can not give any information about these compromises and may only reinforce assumptions of correctness. Validation may not directly expose these compromises, but validation may give an indication that there is an error in the system that must be explored.

Chapter 2- Background for Robotics Simulator Validation

Dynamic robotic simulators are one of the most widely used software tools in the field of Robotics today. Some of the recent focus on these simulations has been making them faster (*e.g.*, as with Gazebo in the DARPA Robotics Challenge), but one ongoing goal for rigid body simulations in general has been greater physical accuracy. The desire is that the simulations should evince physical behavior as close as possible to the real world, whether that closeness is measured quantitatively (as shown in recent experiments by Vose *et al.* [68]) or qualitatively (as can be seen in the unusual behavior observed in rigid "toys" like the Rattleback [44]). Clearly, Robotics will benefit as the physical accuracy of these systems becomes more faithful to reality: planning, optimization, and validation are just a few areas that can reap substantial improvements with better physical fidelity.

This chapter provides a background for further discussion of robotic simulation validation. The chapter begins with a discussion of various design considerations, algorithms, and elements involved in the development of multi-rigid body simulations with contact and friction and ends with a discussion of other research efforts focused on multi-rigid body simulation validation.

2.1 Multi-Rigid Body Simulation Design

Every simulation library is the product of many design decisions. This section discusses the most salient such decisions, as judged by Gazebo's currently supported multi-rigid body simulators and other available open source simulation software.

2.1.1 Simulation Paradigm

Brogliato *et al.* [8] broadly categorize rigid body dynamics simulation software into three paradigms: event driven, time stepping, and penalty-based. Penalty methods treat contact using virtual springs and dampers. The resulting system of equations of motion can be integrated forward directly using arbitrary integration schemes, though implicit integrators are often recommended to avoid issues with the “stiff” equations that can result. Event driven methods (i.e., piecewise differential algebraic equation-based approaches) integrate the equations forward in time until an impact (“the event”) occurs; the impact is modeled and the simulator resumes integrating the equations forward until further events are detected. Time-stepping methods, which are based around differential variational inequality (DVI) mathematical models, can avoid the need to locate events precisely by collecting all of the events that may occur during a time interval into a single complementarity problem. By solving a single such problem, time-stepping can allow systems with many contacts or impacts happening over a small period of time (e.g., a pool break) to be simulated more quickly.

2.1.2 Coordinate Type

Articulated bodies can be defined using absolute coordinates (i.e., six coordinates per link) or independent coordinates (i.e., six coordinates per link minus the number of bilateral joint constraint equations). The former is used in several pieces of open source software (including ODE and Bullet), likely because of ease of implementation. However, absolute coordinates require constraint stabilization (e.g., Baumgarte stabilization [4]) to be used to minimize robot links from unrealistically separating at joints.

Constraint stabilization is also used to separate interpenetrating bodies. The bodies may interpenetrate due to non-convergent iterative solves (see Section 2.1) or inability of the collision detection system (see Section 2.1) to find precise times of contact. Interpenetration is also an organic byproduct of time stepping simulation processes.

2.1.3 Solver Type

The bilateral (joint) and unilateral (contact and joint limit) constraint equations must be solved for constraint forces that obey compressive, non-interpenetration, frictional, and complementarity constraints; Gazebo's supported simulators pose these constrained optimization problems as either nonlinear or linear complementarity problems. If the latter, either a direct (pivoting [11]) or iterative (matrix splitting method [46]) can be used. The only available solver for NCPs at this time is PATH [12]. For LCPs, the iterative Projected Gauss Seidel method seems to work reasonably in practice. However, convergence to a solution is not guaranteed; both Lacoursieré [40] and Drumwright and Shell [16] found these methods yield inaccurate solutions; Lacoursieré showed that for matrix condition numbers greater than 10^7, the method is unusable and that the method also "stagnates".

Direct solvers are limited too. Accuracy is significantly higher than with iterative methods (see [16]), but the linear system solves required of pivoting operations limit the practical number of variables to on the order of 10,000, both due to compuational complexity and available memory. The maximum number of variables is limited even further by the rounding error that accumulates during the pivoting process; the LEMKE [19] library limits the number of pivots to $\min(50n, 1000)$ for this reason.

Some simulation approaches enforce constraints by solving optimization problems rather than complementarity problems: see [17, 67] for examples. While these approaches are being actively investigated, Gazebo does not currently support any simulators using such techniques, which removes a point of comparison. Our intimate knowledge of these approaches leads us to postulate that grasping performance would be affected little using an optimization based approach, but such a comparison will require future investigation.

2.1.4 Collision Detection

There are few collision detection libraries that provide contact points and normal data; most such libraries only test whether two shapes are intersecting. There are multiple ways to compute contact points and normals (see, e.g., [31, 15]), which can depend on the geometric representation (primitive types, constructive solid geometry, polyhedron, implicit surface, or triangle mesh).

Finding correct contact data is complicated because two geometries are unlikely to be "kissing" due to floating point arithmetic, and because contact points and normals can only be estimated when bodies are interpenetrating. The estimation can conceivably be prevented by tracking the movement of the contact surface over time, but we are unaware of an existing open-source simulator that has implemented this idea.

2.1.5 Quasi-Rigid DVI-Based Contact

Lacoursieré's dissertation [41] described a relationship between the complementarity problem regularization and constraint stabilization parameters and a first-order spring and damper model. This relationship indicates that contact constraints between rigid bodies can be effectively modeled as *quasi-rigid* in place of the fully rigid contact previously considered by DVI-based methods. We use the term quasi-rigidity to indicate that contacts between bodies are treated, at least partially, using spring-and-damper terms. We label those models with deformation along the tangent plane of the contacting surface between two bodies as fully compliant; this term would encompass both traditional finite element models and the penalty type model of Song [61].

2.1.6 Modeling Specification

Everything that is to be modeled by a simulator must be provided through a modeling specification which is generally defined in one or more configuration files.

The semantics and structure of the specification language varies from simulator to simulator and may be defined in a closed, proprietary format such that the same specification can not be used by another simulator. There have been a number of efforts to produce XML based specification languages [29] [24] [23] in an attempt to unify simulation specification so that one configuration can be used across many simulators.

The COLLADA format was initially developed by Sony but is currently maintained by the Khronos Group. The COLLADA format was initially developed for entertainment applications; however, due to the common requirements for multi-rigid body simulators and its continual development, the COLLADA format is used in robotics applications as well.

The Open Source Robotics Foundation (OSRF) developed the Unified Robot Description Format (URDF) for the Robot Operating System (ROS); however, OSRF also developed the Simulation Description Format (SDF) for the Gazebo simulator. URDF is primarily designed to support the distributed nature of robotic systems while SDF is primarily designed to support the need for virtual parameters.

Each of these formats is viable for robotic simulation; however, the development and maintenance of two different specification languages by one organization illustrates that adoption of a single modeling specification remains a challenge.

2.1.7 Real World Constraints

Multi-rigid body dynamics simulation has varied applications which have different requirements. Game simulations are generally virtual and are not required to model the real world aside from generating a plausible aesthetic. Robotic simulations model real world systems and must account for the real world constraints inherent to physical phenomena such as physical and temporal limitations. For example, an opaque barrier between a laser sensor and an object physically prevents the sensor from detecting that object. A predictive simulation of the sensor must prevent detection of the object; otherwise, the simulation violates real world constraints.

Plausible multi-rigid body dynamics simulation does not require modeling of real world constraints because modeling of real world constraints increases complexity and impinges on real-time performance. According to the principle of economical models [52], *i.e.* good models include everything necessary to achieve goals but no more, plausible simulations should omit real world constraints. However, if a plausible simulation is used for robotics simulation, the absence of real world constraints may significantly impact simulated behavior in esoteric ways.

If real world constraints are not modeled by a simulation, the simulation may not satisfy requirements of the robotics application domain. For simulations in the entertainment domain, many real world constraints can be relaxed as long as the simulation remains real-time and realistically plausible. On the other hand, robotics simulations require sufficient predictive capability to model real world scenarios, so roboticists need to be aware of real world constraints and whether or not real world constraints are modeled in a preferred simulator. For this reason, roboticists must use great care when using game engines as a basis for robotics simulations.

Three real world constraints that should be considered in robotics simulation are limited observability, sensor uncertainty, and temporal consistency.

Limited Observability Limited observability describes the real world constraint where sensors used by robotic systems are only able to detect objects in the immediate surroundings subject to physical limitations of the sensor. Objects that are occluded to the sensor or objects that are too far away from the sensor may not be observable depending on the underlying physics governing the sensor (illustrated by the example above). Depending on simulator design, the entire state of the world may be accessible to a simulated robot which may allow that robot to observe objects that would be unobservable in a similar real world scenario.

Sensor Uncertainty Sensor uncertainty describes the real world constraint that sensor signals are imperfect measurements and contain noise in the signals generated by the sensors. For example, image based sensors capture discrete pixel data which represents combined information subject to resolution while range based sensors may not return accurate distance measurements. State within a simulation is perfect with respect to the simulated world, but real world sensors are affected to varying degrees by noise. Simulated sensing may contain no uncertainty while real sensors contain significant uncertainty for similar observations.

Temporal Consistency Temporal consistency describes the case that simulated robots must honor the same timing constraints that real world robots are subject to. Robots are not homogeneous systems; instead, robots are the composition of a number of independent processing units, sensors, and actuators which operate at different frequencies and are networked with different degrees of latency. Independent process freqeuencies and intercommunication delays between simulated robotic components are often overlooked and insufficiently modeled. Many robotic processes, such as planners and controllers, require input from other processes and the timing at which input is received dictates the outputs generated. The processes themselves also take time which may only be loosely constrained by the simulation. Real world robots are inherently constrained to a real time environment, and if a simulation does not simulate the timing of robotic components, the simulation will not reflect the behavior of a comparable situated robot.

2.2 Validation Studies of Multi-Rigid Body Simulations

Simulation validation relies on the comparison of data from at least two independent experiments where the data that forms the basis of comparison may originate from analytic solutions, purely simulated experiments, or real world experiments.

The experimental setup is often described as the *scenario* which consists of a description of the experimental system and all the parameters necessary to model the system in simulation. The number of parameters may be large and quantifying these parameters may require additional experimentation. State data of a robotic system typically has high dimensionality with mixed units and the behavior of the system may be nonholonomic which makes direct comparison of state data infeasible. Researchers use one or more *metrics* to reduce state data dimensionality and objectively quantify behavior; however, a metric may not capture relevant information. The scenario, state data, and metric together form the basis of comparison for simulation.

Multi-body dynamics scenarios may range from classical dynamics inspired ideal cases such as a simple pendulum or block on plane to complex, real world inspired examples such as a walking humanoid. Scenarios derived from classical dynamics are usually evaluated against data derived from an analytic solution, but these scenarios are often too simplistic to yield significant insight into the performance of a multi-rigid body system subject to contact and friction, *e.g.* a robot. Scenarios inspired by complex, real world systems are generally infeasible to solve using an analytic approach; however, the goal of robotic simulation is to predict the behavior of these systems. For example, predicting the trajectory of a nonholonomic system like a car involves computing the evolution of the system subject to precise command input timing. A critical step toward demonstrating the consistency of robotic simulation is validation against real world data.

This section surveys research into multi-rigid body simulation validation and are grouped into two categories: large scale comparisons and data collection from real world scenarios. Large scale comparisons generally deal with creating a large framework in order to compare many simulators using the same model specification. Real world scenarios primarily focus on gathering and comparing data from real world experiments to the performance of one or more simulators.

2.2.1 Studies of Simulation Performance

Motivated by the need to increase immersion through realism in an interactive therapeutic application, Seugling *et al.* [57] focused on examining friction, conservation of energy in rotation and collisions, stability in constraints and collisions, and overall correctness using a mixture of quantitative and qualitative metrics. Their scenarios are generally simple cases for which we have analytical solutions: a box on an inclined plane, a box in zero gravity, a ball and plane in zero gravity, a simple pendulum, a compound pendulum, a stack of boxes with small initial seperation, and a set of scenes consisting of various geometries. They measured the ratio between tangential and normal force with respect to inclination angle in the box on inclined plane scenario to evaluate both Coulomb friction and whether a simulator uses an isotropic or anisotropic friction model. They measured the error between initial and final momentum of the box in zero gravity given an initial angular velocity to the box in all axes to evaluate the conservation of angular momentum. They measured the kinetic energy of the ball in the ball and plane scenario to evaluate whether energy is conserved in interpenetration and collision. They measured the distance between bob and anchor over time using the simple pendulum to study constraint stability by varying the mass of the bob and they measured the pendulum trajectory, period error, and conservation of energy with respect to an ideal pendulum to evaluate correctness. They measured the distance between links over time and the average execution time in the compound pendulum to evaluate elasticity of constraints and computational overhead. They measured the average time per colliding pair in the stack of boxes to evaluate the scalability of contacts and they used a qualitative metric to evaluate the stability of piling. Finally, they used a qualitative metric to compare contact behavior between combinations of standard primitives, nonconvex triangle meshes, and convex triangle meshes.

Boeing *et al.* [7] were motivated to identify which dynamics engines available are best for gaming applications. To carry out their study, they developed a middleware abstraction layer, *i.e.* Physics Abstraction Layer (PAL), which is intended to provide a unified interface for dynamics simulators. Their work focused on integrator performance, material properties such as friction and restitution, constraint stability, and collision performance using a combination of quantitative and qualitative metrics. Their scenarios consisted of a sphere falling under gravity, a sphere falling onto a box, a box on an inclined plane, a chain of spheres linked to one another and constrained to two fixed pylons, a number of spheres falling into an inverted pyramid, and stacks of spheres or cubes. They measured the positional error of the sphere falling under gravity to evaluate integrator performance against the ideal case. They measured the bounce height of the sphere falling onto a box to evaluate energy dissipation from collision with respect to the ideal case. They measured the angle at which the box on an inclined plane begins to slide to evaluate Coulomb friction implementations against the ideal case. They measured the displacement of spheres in the chain of spheres to compare constraint stability between simulators. They measured the number of spheres interpenetrating into the inverted pyramid to compare simulator collision performance. Finally, they used a qualitative metric to evaluate the stacking of spheres or cubes as a further test of collision performance.

Gonzalez *et al.* [27] [28] were motivated to develop multi-rigid body validation scenarios by needs in mechanical engineering applications; however, they also recognized the need for a historical database of simulator performance as advocated by Oberkampf [49]. Their approach allows for scenarios that are more complex than ideal cases by allowing for comparison against reference solutions rather than against analytical formulas. They computed reference solutions in commercial software and validated the reference solutions against their own Fortran based simulations. Their system attempts to balance accuracy with real-time requirements by bounding the computational efficiency of simulation. They also permit the tuning of parameters to achieve a required precision with respect to the reference solutions and use the minimum processing time as the metric of comparison. Their database maintains a problem set, *i.e.* scenario definitions, a set of reference solutions, and a description of the parameters used to produce the scored values. The problem sets that they developed consist of two classes of problems: "basic problems" and "industrial-like applications." Their basic problems consist of a simple pendulum, a one degree-of-freedom four-bar linkage, a planar mechanism consisting of seven bodies known as "Andrew's mechanism", an overconstrained system of six links with one degree-of-freedom known as "Bricard's mechanism", and a flyball governer. Their industrial-like applications are complex machines for which an analytical solution is infeasible.

Hummel *et al.* [35] were motivated to study validation of multi-rigid body simulation to determine the best candidate for virtual reality based haptic simulations in zero gravity training. Like Boeing *et al*, they developed their own abstraction layer. They measured computation time using a scenario involving the collision of large number of spheres. They evaluated collision restitution by measuring the bounce height of an object with respect to an ideal analytical solution. They evaluated constraint stability using the same scenario and metric defined by Boeing *et al*; however, they extended this work by varying the number of spheres and measured the number of spheres at which constraints began to become unstable. They evaluated one degree-of-freedom constraint stability by simulating the throw of a toggle switch and measuring switch vibration and the amount of constraint limit violation. They evaluated interpenetration during collision by perturbing a box constrained on all sides and measuring the translational and rotational displacement of the box. Finally, they evaluated the behaviors of a realistic screw which they modeled using solids by measuring the position of the screw head when screwed into the nut.

Erez *et al* [18] used their own simulation framework, MuJoCo, to compare a number of dynamics engines for robotics applications. Their goal was to test dynamics engines on identical models to characterize speed, accuracy, and stability. Their approach involved complex robotics scenarios, similar to the industrial-like applications of [27], and excluded simple systems where analytic solutions are available. Their accuracy metric is that of "self-consistency" where they measure how much a simulated trajectory deviates from a reference trajectory after a number of timesteps and they argue that this accuracy metric is also a stability metric. They generate a reference trajectory from each simulator using very small timesteps, execute the same trajectory using larger timesteps, and measure deviation from the reference trajectory. They limit their scenarios to use only revolute joints, sphere and capsule collision geometries, Coulomb friction contact dynamics, and only forces from gravity, joint constraints, joint torques, and joint stiffness and damping. Their scenarios involved: a 35 degree-of-freedom robotic arm, a 25 degree-of-freedom humanoid, a 5 degree-of-freedom planar kinematic chain, and 27 capsules falling onto a ground plane.

2.2.2 Real Data Studies for Simulation Validation

While comparisons of simulations with analytic solutions and simulated experiments are valuable, a simulator's predictive capacity is most clearly demonstrated by a simulator's ability to recreate real behavior, so there are an increasing number of research projects dedicated to capturing data from real world scenarios.

Zhang *et al* [72] gathered data from planar grasping experiments and used the data to calibrate a simulator capable of recreating the behavior exhibited by the real system. Their apparatus recorded the state of an actuator pushing an object into a single degree of freedom gripper. They used data gathered from the apparatus to determine uncertain parameters such as friction for a calibrated simulation of the gripper and validated the calibrated simulation against additional data generated by the apparatus. Lu *et al* [43] developed a benchmarking framework in an effort standardize simulator validation. They demonstrate their framework by using data generated from the Zhang *et al* apparatus and by producing a validation comparison of complementarity problem solvers.

Yu *et al* [71] investigated variable contact geometry and friction in a 2D plane using a robot arm and an automated data gathering process. Their process allowed for unattended reset of the experiment and enabled them to gather a large dataset of real data. The same lab investigated contact modeling accuracy by recording the pose of an object being pushed by a robot [39]. They recorded the telemetry of an object maintained in the grasp of a robot arm as the object was pushed across a surface. They compared forces and state from simulation with the recorded data. This same lab also investigated improving the accuracy of impact models by analyzing data gathered from the tossing of a two dimensional object onto a hard surface [20].

Frigerio *et al* [25] studied an extremely sophisticated robotic platforms for validating simulation. They compared actuator torque profiles between virtual and *in situ* HyQ robots. The challenge with validating simulations of such complicated machines is attribution for discrepancies, i.e., what modeling aspect should be blamed when the simulation does not match recorded telemetry data to the desired accuracy. Additionally, we have found that ensuring that telemetry data is collected without error is a challenge for even much simpler robots.

Chapter 3- Comparative Evaluaton of Algorithms Using Quasi-Rigid Grasping

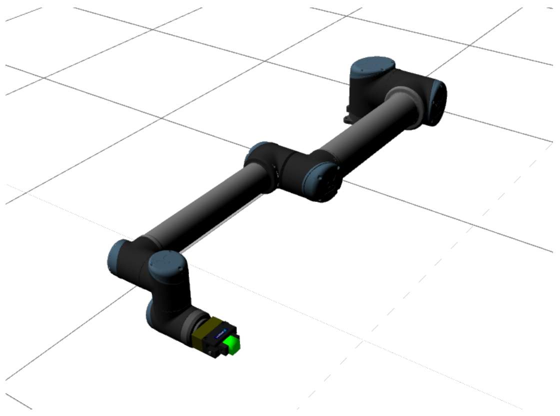
Every simulation library is the product of a series of design decisions, including direct *vs.* iterative mathematical programing/optimization solver, geometric representation, coordinate type (reduced or absolute), and simulation paradigm (penalty-based, piecewise differential algebraic equation-based, time stepping). Exactly how any of these decisions affects a particular scenario or task or a class of scenarios or tasks has yet to be established, making objective comparison and evaluation of simulators extremely difficult. This chapter focuses on comparing the performance of a number of simulators using a scenario and task that is ubiquitous in robotics: grasping of quasi-rigid objects with rigid robots.

Stable quasi-rigid grasping is an anecdotally challenging task to simulate, and potential causes of failure are numerous: constraints are subject to numerical drift; truncation error arising from terminating an iterative method early can cause slip, and "jitter" due to correcting bilateral constraint errors and interpenetration can transmit excessive forces through the multi-body *system*

The work described in this chapter was carried out using an open test platform, Gazebo version 4.0, to directly compare the performance of four existing multi-rigid body simulators on the aforementioned grasping task. Gazebo is developed by the Open Source Robotics Foundation (OSRF) which received funding for Gazebo from the Defense Advanced Research Projects Agency (DARPA) to serve as the simulation platform for the DARPA Robotics Challenge. As a result, Gazebo currently has broad adoption and supports a number of dynamics libraries which make it well-suited as a fair platform for simulator comparison, *i.e.* an experimenter can ensure that the same model is used across all simulators. As of this publication, Gazebo supports the Open Dynamics Engine (ODE), Bullet physics, the Dynamic Animation and

Robotics Toolkit (DART), and Simbody.

This chapter considers two grasping scenarios: (1) a simulated robot arm moving through a trajectory while maintaining a grasp using a parallel gripper; and (2) a simulated fixed parallel gripper attempting to maintain a grasp on multiple objects simultaneously. The arm model used in the former scenario is a hybrid of the Universal Robotics UR10 arm, sourced from an existing ROS package and mated to a model of the Schunk MPG-80 parallel gripper. This configuration represents a reasonably accurate model of an industrial arm with respect to kinematics, geometry, and dynamics. On the other hand, the model used in Scenario (2) is contrived: a pair of boxes represent the parallel jaw gripper. The "robots" used in both scenarios are depicted in Figure 3.


(a)

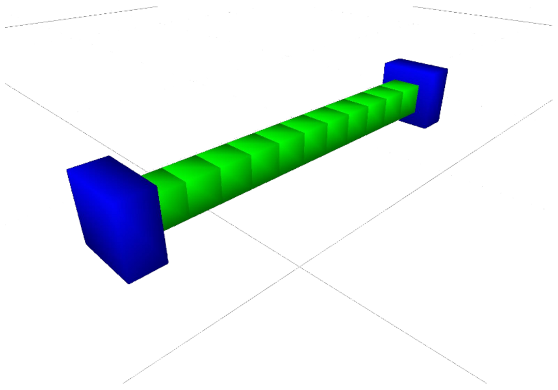

(b)

Figure 3: Simulated Grasping Robots

(a) The UR10 arm with Schunk hand used in the grasp with industrial arm experiment and (b) the block gripper with eleven objects to be grasped used in the multi-block experiment. In both images, grasped objects are shaded in green and in (b) the grippers are shaded in blue.

Finally, we note that this research does *not* try to answer the question of which software library may be superior. All libraries tested are actively developed and performance is subject to change, often dramatically, as bugs are introduced or corrected and new techniques are tested. It is also conceivable that Gazebo contains bugs in the interfaces to one or more libraries. This research instead focuses on understanding the factors that can contribute to grasping performance in simulation and providing a metric for evaluating grasp performance in simulation.

3.1 Failure Modes for Quasi-Rigid Grasping

Objects slipping from a simulated robot's grasp can occur for several reasons, described below.

Slip If the grasping force or friction coefficient is not large enough for force closure. This case is the only one that is not an artifact of the simulation software.

Iterative Method Non-Convergence Iterative methods are often terminated early; non-convergence can cause slip to occur at points of contact, joints to move apart unrealistically (thereby, for example, causing one or more gripper fingers to separate from the grasped object), or interpenetration to occur (which could cause objects to drop or become entangled [54]).

Rounding Error Rounding error can occur even with direct solvers, leading to all of the problems described immediately above, though with lesser severity expected in this case.

Regularization Error Regularizing the complementarity problem (via, e.g., the "constraint force mixing" parameter in ODE), causes constraint violation, which can lead to all of the issues described under *iterative method non-convergence*.

Constraint Stabilization Projection methods (used in RPI-Sim, ODE, Bullet, and DART) for correcting joint constraint errors and interpenetration are prone to adding energy to the multi-body system [63] (which, among other issues, can compromise simulation stability).

Imprecise Contact Information As Section 2.1 intimates, correct algorithms—and even precise problem specifications—for determining contact data have yet to be established. The general effects of inaccurate points of contact, normals, or both is currently unknown. The effects of approximating by point samples a polygonal or nonlinear manifold corresponding to the locus of intersection between two contacting bodies is also unknown. *Contacting shapes between the bodies considered in the present research are convex polyhedra, meaning that bodies will not interpenetrate if the contact constraints are not violated on the convex hull of the contact manifold.* This fact permits ignoring the effects of point sample approximation in the present work.

Tangential Drift Although interpenetration and bilateral constraint errors are addressed by constraint stabilization, we are unaware of any multi-rigid body simulation libraries that stabilize constraint drift in the tangential direction, which would require tracking the evolution of the contact manifold as two bodies move. Without addressing such drift, we expect manipulated objects to gradually slip out of grasp.

We determined the factors above from reasoning about how multi-rigid body simulation software functions. However, teasing apart which of these factors is responsible is not the focus of this research. Indeed, evaluating even a single factor independently is impractical with most of the simulators that we evaluated. Instead, this research applies multiple simulation libraries to test hypotheses (see Section 3.3) using the spectrum of available multi-rigid body dynamics software libraries.

3.2 Quasi-Rigid Grasping Experimental Specifications

This section covers the evaluated simulators, the tasks, the performance metric used, and the description of the experimental controls.

3.2.1 Simulating Quasi-Rigid Grasping

We assessed every simulation using an integration step size of 1ms. We found anecdotally that smaller step sizes produced better grasping performance (as expected), though our focus in this work is on factors that affect grasping performance *independently of integration step*. We conducted this investigation because reducing the step size leads to considerably slower simulations (e.g., simulations integrated at 10^{-4}s generally run an order of magnitude more slowly than those running at 10^{-3}s).

ODE The version of OPEN DYNAMICS ENGINE (ODE) that we tested has been modified by Open Source Robotics Foundation to correct interpenetration by attempting to project bodies to a disjoint configuration (see [34] for details). ODE uses a slightly modified version of the first-order time stepping method described by Stewart and Trinkle [64] and Anitescu and Potra [2] with constraint stabilization procedure following the approach described in [32]. ODE uses absolute coordinates and supports both primitive and triangle mesh geometric types. GAZEBO 4.0's interface to ODE uses an iterative solver only, though "vanilla" ODE supports a direct solver.

Bullet The version of BULLET we tested (2.82) also uses a time stepping approach; BULLET supports independent coordinates, though only absolute coordinates are supported in the version of GAZEBO used in this study. Like ODE, BULLET also uses an iterative solver and supports both primitives and meshes; our examination of the source code, interface, and discussion groups indicate that ODE and BULLET function very similarly at the multi-rigid body dynamics level. However, BULLET also includes code for simulating soft bodies, fluids, and particles (none of which are used in the present study).

DART We tested DART version *core-4 1.1.0*, which uses an independent coordinate formulation and the Stewart-Trinkle/Anitescu-Potra time stepping approach.

DART uses a direct, mixed linear complementarity problem solver (specifically, ODE's Dantiz-Cottle LCP solver) and a triangle mesh geometry representation. DART uses a polygonalized friction cone for contact, thereby following the Stewart-Trinkle/Anitescu-Potra contact model more faithfully than the approximate pyramidal friction model used by both ODE and BULLET. We had to modify the DART source code to allow us to change the Coulomb friction coefficient for DART experiments.

Simbody SIMBODY version 3.4 uses a considerably different paradigm than the other physics engines in these experiments. SIMBODY uses a quasi-rigid contact model (to permit modeling the human skeleton, muscles, and skin) that follows the event-driven paradigm in place of the time stepping approaches used by ODE, BULLET, and DART. SIMBODY proved to be too slow to conduct the numerous requisite experiments (simulating a single grasping scenario required on the order of days, i.e., in the same timeframe as FEM-based simulations). This elimination is unfortunate because SIMBODY's focus has been on accuracy, while the other simulations described above have targeted low running time. SIMBODY's performance does make it clear why the present work focuses on rigid body dynamics in place of more accurate FEM-type deformable codes, for which the remainder of the libraries described in this section can simulate the scenarios in minutes: considerably longer simulation times preclude model predictive control, fast edit-compile-test cycles, motion planning, and similar applications of multi-body dynamics simulation to robotics.

RPI-Sim The multi-block grasping example was sufficiently simple that we were also able to set up the example in RPI-SIM (Subversion revision 463). RPI-SIM supports pivoting solvers (Lemke's Algorithm); a popular iterative matrix splitting method (Projected Gauss Seidel); and PATH [12], a popular library for solving complementarity problems (which operates using Lemke's Algorithm for LCPs with few variables and via a Newton-based method more generally). RPI-Sim uses the first-order Stewart-Trinkle/Anitescu-Potra method with the stabilization approach described in [32].

3.2.2 Quasi-Rigid Grasping Scenarios

The experiments using both grasping tasks, each of which is described immediately below, begin with the grasped object lodged firmly in the gripper: no grasp planning, pre-grasping, *etc.* were necessary in any experiment.

Grasp with Industrial Arm We acquired a model of the UR10 robotic arm as depicted in Figure 3(a) from the ROS Industrial repository and converted it to Gazebo's SDF format. We attached this model to a model that we constructed of the Schunk MPG-80 hand, using CAD files and technical schematics from the manufacturer. The mated UR10/MPG-80 model yields a 8-DoF system (6-DoF arm + 2-DoF gripper). The inertial properties for the Schunk hand were set by approximating the bounding volume of the palm assembly and the individual fingers with boxes and apportioning the mass of the palm to be 80% of the total mass (described in the hand schematics); each of the two fingers were assigned the remaining 20% of the total mass. The box approximations were used for collision geometries in the geometric primitive experiments and the CAD meshes were used for collision geometries in the tessellated mesh experiments. The gripper prismatic joints were constrained by joint limits derived from the stroke per finger (defined in the schematics). Each gripper is actuated toward closure with 100N of force, which should be sufficient to maintain force closure on even relatively massive objects.

The object to be grasped was modeled as a simple cube with equal dimensions of 100mm. This yields a primitive/primitive representation, which we hypothesized would yield more robust grasping than using primitive/mesh or mesh/mesh-based experiments. A "collision box" was used for the geometric primitive experiments; for the tessellated mesh experiments, a mesh was generated in Blender using the same dimensions as the cube. The mass of the object to be grasped was set to 1 kg, and its inertia tensor was derived from a box with given dimensions and mass.

The arm controller was designed to move the arm through randomized sinusoidal trajectories for all joints using PD-control. The amplitude of each sinusoid was tuned to be as large as possible without the arm colliding with either itself or the ground plane. As noted above, the prismatic joints controlling the grippers applied 100N of constant force.

Block Grasp The block grasp scenario consists of a simple manipulator with a fixed base and a pair of parallel grippers as depicted in Figure 3(b). Each gripper is modeled as a box with dimensions larger than the grasp object and with a mass of 1 kg. Each gripper is actuated toward closure with 100N of force *per block*.

The object to be grasped is the same as used in the industrial arm experiment, i.e. 1/1000m^3 with 1kg mass.

3.2.3 Measuring Quasi-Rigid Grasping

Selecting a metric for identifying effective grasping performance proved challenging because our focus is only on the qualitative behavior of keeping the object within the gripper. Comparing telemetry data against, e.g., control commands, would not be indicative of simulation performance as the robot might be observed to maintain a perfect grasp while the end-effector deviates from its commanded path. We devised a work-type metric to capture the qualitative behavior we sought. Specifically, we use a first-order approximation to the kinetic energy necessary to restore the grasped object to its *initial pose relative to the grippers.* This formula is described below:

$$T = \frac{1}{2} m \boldsymbol{v}^{\mathsf{T}} \boldsymbol{v} + \frac{1}{2} \boldsymbol{\omega}^{\mathsf{T}} \mathbf{J} \boldsymbol{\omega} \tag{1}$$

$$\boldsymbol{v} \equiv \frac{\boldsymbol{x}^* - \boldsymbol{x}}{\Delta t} + (\dot{\boldsymbol{x}}^* - \dot{\boldsymbol{x}}) \tag{2}$$

$$\boldsymbol{\omega} \equiv 2\mathbf{G}(\frac{\boldsymbol{q}^* - \boldsymbol{q}}{\Delta t}) + (\dot{\boldsymbol{\theta}}^* - \dot{\boldsymbol{\theta}}) \tag{3}$$

$\boldsymbol{x}^*$ and $\boldsymbol{x}$ in Equation (2) are the desired and current position of the grasped object, $\dot{\boldsymbol{x}}^*$ and $\dot{\boldsymbol{x}}$ in Equation (2) are the desired and current linear velocity of the grasped object, $\boldsymbol{q}^*$ and $\boldsymbol{q}$ in Equation (3) are the desired and current orientation of the grasped object (in unit quaternions), and $\dot{\boldsymbol{\theta}}^*$ and $\dot{\boldsymbol{\theta}}$ in Equation (3) are the desired and current angular velocity of the grasped object. The difference between the unit quaternions is converted to an angular differential using well known formulas; we use one such formula to determine $\mathbf{G}$, which is defined with respect to $\boldsymbol{q}$ (see, [48], p. 175).

Each desired configuration is defined with respect to the grasped object's initial configuration. Velocities are defined relative to the gripper's velocities. The inertia tensor of the block, $\mathbf{J}'$ is defined relative to the grasped object body frame and is transformed to the world frame by operation $\mathbf{J} = \mathbf{R}\mathbf{J}'\mathbf{R}^{\mathsf{T}}$ where $\mathbf{R}(\boldsymbol{q})$ is the grasped object's orientation matrix. All other variables described in the equations above are defined relative to the global frame.

For each iteration, we compute the metric T of all grasped objects with respect to both the left and right grippers; those values are then averaged to yield a single value. The simulation was terminated if the simulation time reached 100 seconds or the metric exceeded 10^7 units (the simulation configurations lack a ground plane, so an object that slips from a grasp causes the metric to quickly exceed 10^7). Other invalid configurations, such as possible locking of models due to interpenetration (see [54] also) were noted, but otherwise not objectively measured; however, our metric does penalize interpenetration in a stable grasp compared to stable grasps with no interpenetration. Finally, note that $T = 0$ when the relative configuration between the object and the grippers matches the initial relative configuration.

3.2.4 Quasi-Rigid Grasping Experimental Controls

The experimental controls consisted of each of the three simulators (four in the case of the multi-block experiment) listed previously using GAZEBO's default parameters. We do tune one simulator—ODE, which was GAZEBO 4.0's best supported simulator—to show that it is possible to improve on these parameters significantly for this task; however, we also wished to establish a performance baseline. GAZEBO's default parameters have been determined using domain knowledge to maximize performance (speed and accuracy) over numerous simulation models and environments.

The models used in the experimental controls used realistic inertial values for all rigid bodies. Primitive geometric boxes were used to model the grippers and grasped objects. Coulomb friction coefficients were set to 100.0, which is significantly higher than that considered to be natural but ensures that the models contact without slip (in theory, if not implementation). Examination of multiple published friction tables indicates that 1.0 acts as an effective upper limit, though values larger than 1.0 are compatible with Coulomb's friction model.

3.3 Testing Hypotheses with Quasi-Rigid Grasping Scenarios

This section describes the hypotheses that we formulated and examined using the industrial arm and multiple block grasping scenarios.

3.3.1 Hypothesis 1: Modifying Inertia Properties Could Lead to Improvements in Grasping Performance

It is accepted in the multi-body dynamics simulation community that large discrepancies in mass ratios lead to lower simulation performance. Such discrepancies cause ill-conditioning of the manipulator inertia matrix[1], given Featherstone's analysis [21] that showed that the condition number of this matrix grows up to $O(n^4)$ in the number of links in a robot and is also dependent on variations in link inertia (among other factors). This ill-conditioned inertia matrix may be viewed as a source of stiffness in the underlying differential equations [53], thereby affecting simulation stability. However, we guessed that grasping performance might be affected as well.

We tested this hypothesis by comparing performance for the experimental control against a robot model with modified robot arm and gripper link inertias (for the UR10). We assigned the shoulder link a mass of 1kg and inertia matrix of identity. Each subsequent link in the chain would have its inertial properties scaled geometrically (i.e., the second link would be scaled by $1/x$, the third link would be scaled by $1/x^2$, *etc.*) We experimented with scaling factors between 1.0 and 10.0.

3.3.2 Hypothesis 2: Increasing the Friction Coefficient to Very Large Values Will Not Reduce Grasping Performance

While setting the Coulomb friction coefficient to a large value (i.e., larger than 1.0) will not generally yield a realistic simulation, there exists numerous anecdotal accounts of simulation users setting friction coefficients to particularly large values to prevent slipping at contacts. We tested the hypothesis above to see whether particularly large values of μ would reduce the numerical stability of the pivoting LCP solvers; we would expect the condition number of the LCP matrix (see [2] for the matrix's structure) to increase proportionally with μ. For testing the hypothesis, we modified the friction coefficients between contacting models in our experimental group by using a friction value of $\mu = 10^8$.

[1]Although such a matrix is not explicitly constructed within all simulators, the conditioning of that matrix should yield a measure of numerical stability.

3.3.3 Hypothesis 3: Using Primitive Geometries in Place of Mesh Geometries Yields Improved Grasping Performance

Historically, modeling contacts between bodies represented using ubiquitous mesh-based geometries (e.g., triangle mesh) has proven challenging. We wanted to test whether this effect was present in the evaluated simulation software. In cases where multi-rigid body dynamics libraries supported both mesh and polyhedral representations (i.e., ODE and Bullet), we varied the representation of the grippers, of the object, and both. The evaluated groups were then (1) polyhedral grippers, polyhedral object; (2) polyhedral grippers, mesh object; (3) mesh grippers, polyhedral object; and (4) mesh grippers, mesh object.

3.3.4 Hypothesis 4: Pivoting LCP Solvers Are More Effective at Simulating Grasping than Iterative Matrix Splitting Solvers

This hypothesis follows from past research that has found that pivoting solvers are considerably more effective at solving LCPs (corresponding to multi-rigid body contact problems) with tens of variables to high degrees of accuracy [16]. We speculated that a scenario with low admissibility for error would be more likely to expose this difference. Figure 3(b) illustrates this scenario, for which even one block slipping would likely lead to catastrophic failure. The experimental variable for testing this hypothesis was the number of blocks in the grasp, which varied from one to eleven.

3.4 Comparative Analysis of Quasi-Rigid Simulation

The experiments generated significant data, including numerous plots and videos, which are available at our repository (`https://github.com/PositronicsLab/grasp-data`). Plots of control performance for the industrial arm and multi-block performance are depicted in Figures 4 and 5.

The data indicates the following responses to the hypothesis presented in the previous section.

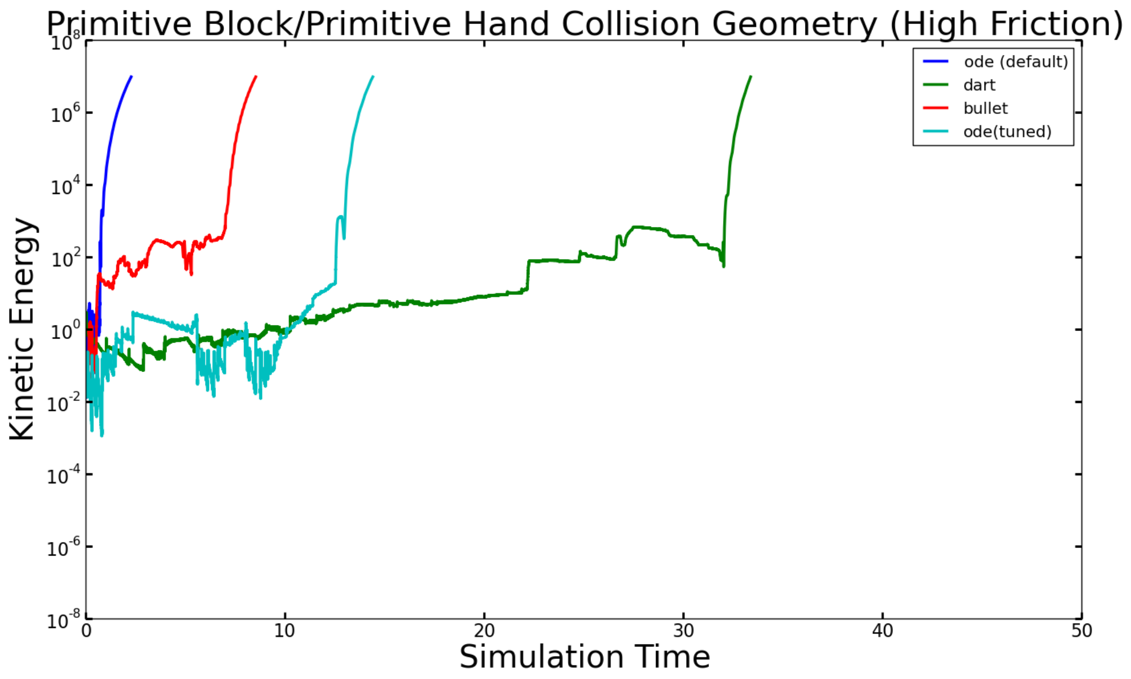


Figure 4: Industrial Arm Grasp Control Analysis

Metric performance for the industrial arm grasping experimental control over three simulators (ODE, Bullet, and DART). Bullet becomes unstable immediately. ODE is able to maintain a stable grasp for nearly a second of virtual time. DART is able to maintain a stable grasp for nearly 35 seconds.

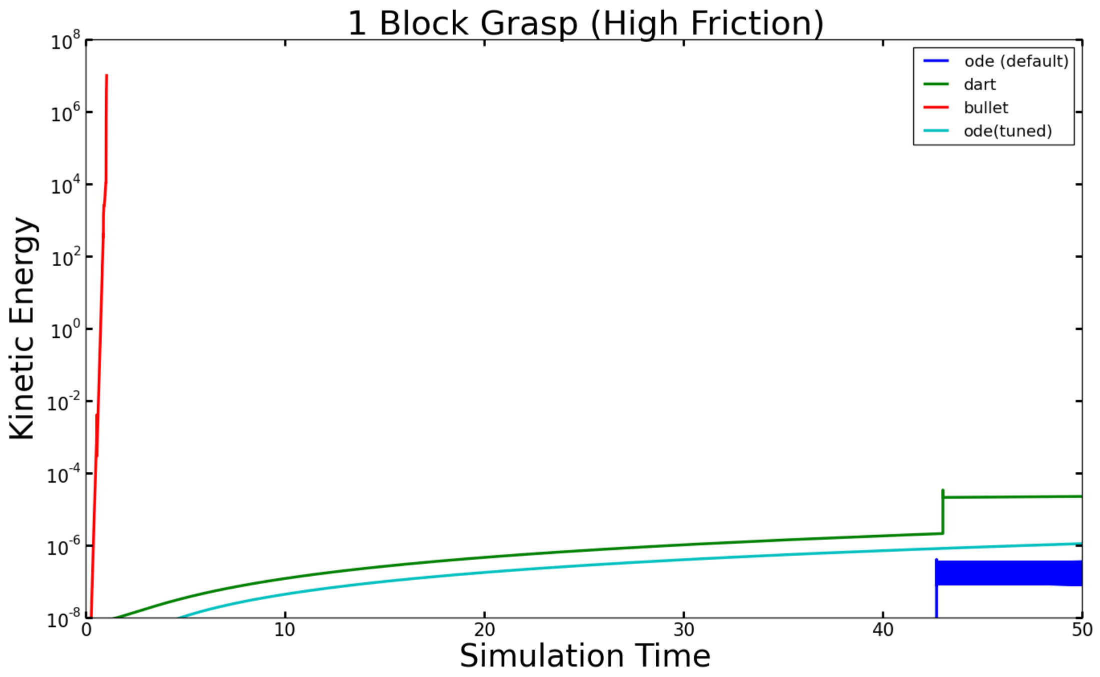


Figure 5: Multiple Block Grasp Control Analysis

Metric performance for the multi-block grasping experimental control (using a single block) over three simulators (ODE, Bullet, DART). Bullet becomes unstable immediately. DART (and to a lesser extent, ODE) exhibits a concerning discontinuity in performance, but generally is able to maintain a grasp for the length of the experiment. ODE maintains a stable grasp.

Does altering inertial values affect grasping performance (Hypothesis 1)? *Yes.* Unit mass / inertia yields significantly better grasping performance compared to the control group with realistic inertias (as depicted in Figure 6). Any integer scaling factor reduces the performance significantly worse than the control. These results indicate that simulation users must take care in setting inertial values, *even for this task for which simulation stability does not seem to be a factor.*

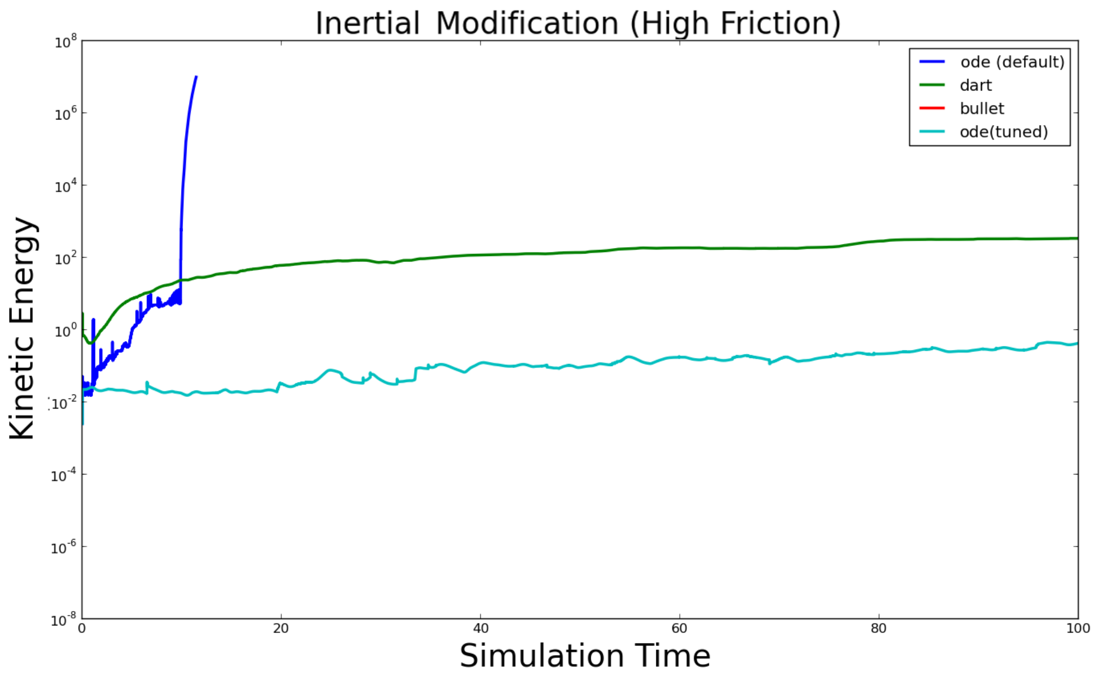


Figure 6: Industrial Arm Inertial Modification Analysis

Metric performance for the industrial arm grasping experiment over three simulators (ODE, Bullet, and DART) using modifications to inertial values. Grasping performance is improved significantly in two simulators compared to the depiction in Figure 4.

Do very high friction coefficients yield less stable simulations (Hypothesis 2)? *No.* Our experiments (and Figure 7) indicate that a high friction coefficient ($\mu = 10^8$) does not affect the stability of the simulations.

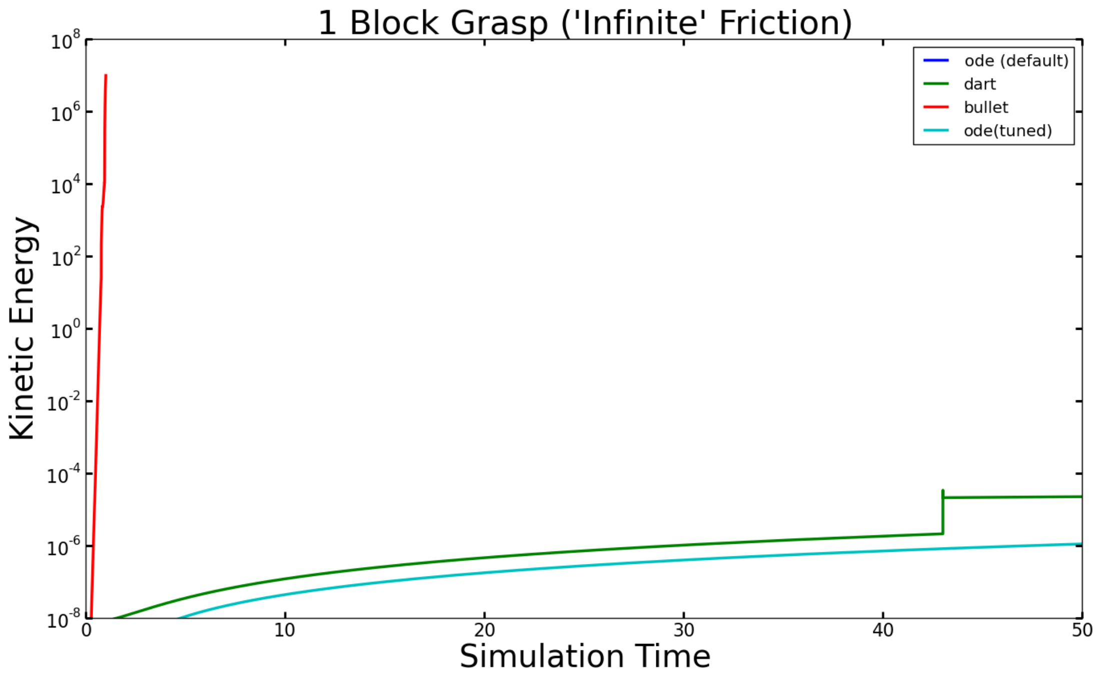


Figure 7: Multiple Block Friction Analysis

Metric performance for the multi-block grasping experiment (using a single block) and very high friction value over three simulators (ODE, Bullet, and DART). Performance appears identical to Figure 5.

Does using mesh geometries impact grasping performance (Hypothesis 3)? *Yes.* Both tested versions of DART and BULLET "segfaulted" with tessellated geometry. ODE drops the grasped object nearly immediately. We recommend using primitive type geometric representations in these simulation libraries whenever possible; aside from maximizing the performance metric in our experiments, the software implementations for primitive geometries have anecdotally proven to be far more robust.

Does the iterative solver fail readily on the grasping tasks (Hypothesis 4)? *No.* Surprisingly, the lack of convergence proofs for the iterative LCP solver [40] and the much poorer solution performance [16] does not translate to challenges with grasping. Instead, *we have found that the iterative solver performs better than the pivoting solver on the sensitive grasping scenario that we devised* (see Figure 8).

Does a particular simulation library outperform others across the board? While we prefer not to address this question for reasons described in the introduction to this chapter, we understand that readers will ask it. With the caveats in that section in mind, DART generally outperformed ODE under our metric; this performance differential could be due to many factors (pivoting method vs. iterative, accuracy of friction cone approximation, generalized coordinates vs. absolute coordinates). One key observation: ODE was able to simulate all scenarios, even if it does not always yield the highest performance; the library crashed less frequently than others, as the software has remained quite stable. BULLET performed surprisingly poorly, which Erwin Coumins (the lead developer) attributed to the particular version we examined (BULLET undergoes a fairly aggressive release schedule) as well as possible bugs in the GAZEBO-BULLET interface, in personal communication.

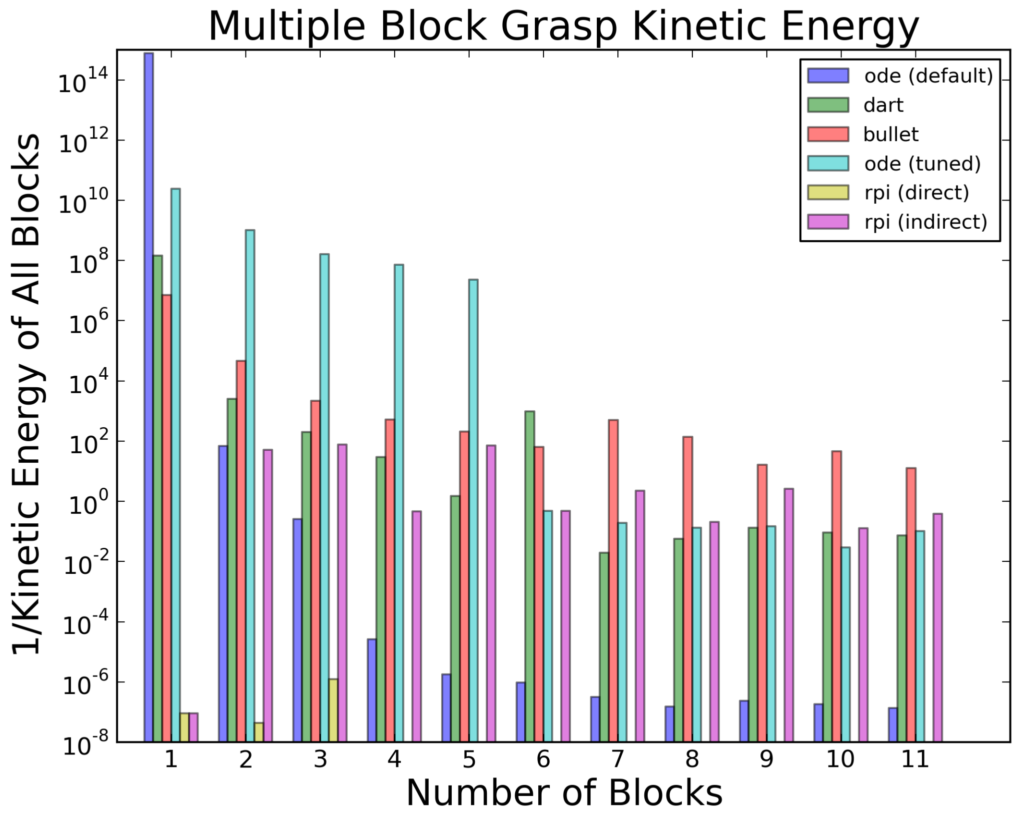


Figure 8: Multiple Block Solver Analysis

Logarithmic inverse metric performance for the multi-block grasping experiment, illustrating direct vs. iterative solver performance at the end of 0.3s of virtual time. Taller bars indicate better performance. A missing bar (see, e.g. `rpi (direct)` for 11 blocks) indicates failure.

3.5 Conclusions and Future Work with Grasping Scenarios

Our experimental results make clear that analyzing quasi-rigid grasp failures in multi-rigid body simulation software is not a trivial problem. Software quality and design decisions can both affect grasping performance; the latter's effect is not necessarily predictable, as, for example, the results from testing our fourth hypothesis indicate. Our results point to new research efforts as well. Why do splitting matrix iterative LCP solution methods work better on the multi-block grasping scenario? What is the mechanism by which "regularizing" the inertia matrix values leads to more stable grasping performance? Can there be an effective middle group between primitive geometric representations (which are fast to evaluate and less challenging to code) of objects and mesh based representations (which are slow and challenging to code but for which formats are ubiquitous and shapes are generalizable)?

Chapter 4- Validating Controllers Through Temporally Consistent Simulation

As dynamic simulation becomes increasingly prevalent in roboticists' software development cycle, new needs are beginning to emerge. This chapter addresses one such nascent need: validation that controllers for robots modeled with physically accurate dynamic simulation will function as desired when transferred to physically situated robots.

There exist numerous technical reasons at the systems level that make the above goal surprisingly difficult, including architectural challenges (adapting existing simulation software toward meeting the goal), scheduling challenges (slowing the rate of execution of the controllers to match the time evolution of simulations), and timing challenges (timing processes with sufficient precision to measure high frequency control loops).

The technical challenges become particularly tortuous when accounting for simulations and controllers that may run on symmetric multi-processing, distributed processing systems, or GPUs and when supporting simulations that use higher-order, adaptive, or implicit integrators. This chapter does not address these considerations and instead focuses on the time consistency between control software and the simulation under simpler conditions.

We also avoid consideration of simulations that run significantly faster than real-time (we currently wish to steer clear of investigations into how to "drop" cycles from controllers yet still achieve some minimum level of simulated robot performance). This omission is reasonable due to the common desire for high accuracy in robotic simulations, which often require on the order of a second of computation to simulate a second of time. Robotics simulations to-date typically focus on physical accuracy (*i.e.*, attempting to produce output that matches real world phenomena). Our work focuses on temporal consistency by slowing the execution of user-level software such that it proceeds proportionately to the advancement of virtual (simulated) time just as it would on an actual robot.

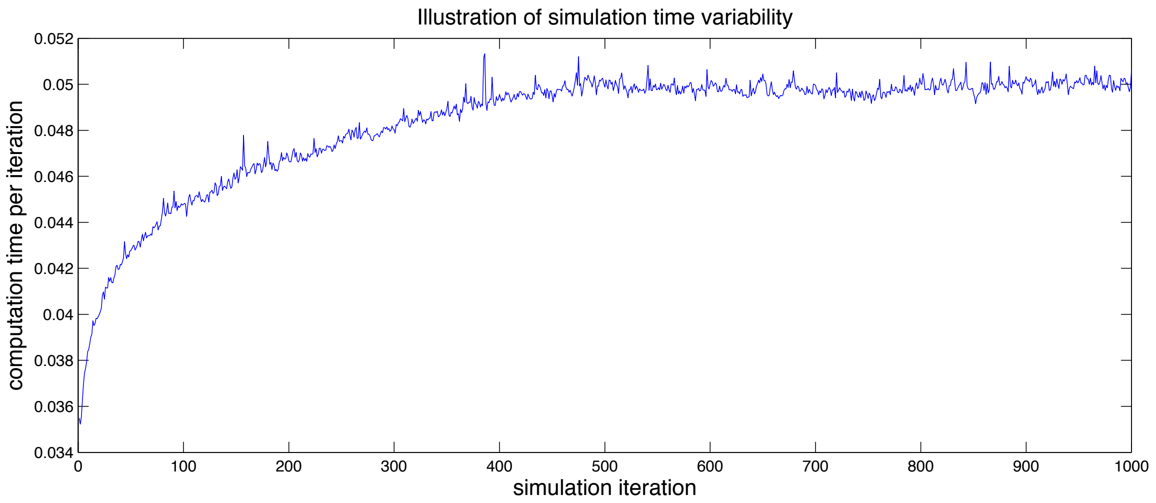


Figure 9: Processing Time for a Large Number of Collisions

Plot of experimental data modeling 1,000 boxes moving on an enclosed planar surface using ODE [59]. The plot shows that the time required to compute a simulation iteration is highly variable: the maximum time is over 50% higher than the minimum time in this experiment. Thus, the execution time granted to the robot controller changes over time proportionately to the rate of simulated time.

For a simple example, if the simulation advances one second of virtual time [37] for every ten seconds that pass in the virtual world, we would want to run the controller at 1/10 its nominal speed: if the nominal speed were 100Hz, we would want the controller to run at 10Hz. As experimental data in Figure 9 shows, the disparity between virtual and real time does not remain constant, so our strategy must correspondingly adapt to such fluctuations dynamically. The practical effect of ignoring this disparity in time is that controllers which run (even slightly) more slowly than their nominal frequency may cause robots or other controlled systems to operate one way in simulation and another way on physically situated systems, *even if the fidelity of the simulation to reality is nearly perfect.*

The *temporal consistency* of simulating dynamics is safely ignorable when simulating dynamics for computer gaming and computer animation applications (the original focus of some popular simulation libraries like Gazebo and ODE). However, the temporal consistency between the simulation library and the "user level" software (controllers, planners, perception loops) is a relevant concern when simulating dynamics for robotics applications. This issue arises because the simulation software does not simulate at the rate of "wall clock" time; as a result, the user level software—which may be expected to feed commands to and pull state from the simulation at roughly the same rate it would perform those operations on a physically situated robot—can not be expected to exhibit similar performance in simulation as *in situ*.

This chapter investigates the issues of temporally consistent simulation and describes a framework that ensures robotic simulations produce temporally consistent results. This objective is achieved through the scheduling and timing of the dynamics process and the other user level processes that are necessary for robotics simulation.

4.1 A Contrast Between Robot and Simulation Architectures

We depict the architecture of a physically situated robotic system in Figure 10 as a reference point. Real-time components such as controllers generally operate at regular, independent frequencies while non-real-time components deliver inputs to other modules at variable frequencies.

A controller that has not received updated input from a non-real-time module will continue to control motor servos using feedback and stale input until new input is received. As a result, a controller can drive the system away from the desired trajectory if planning outputs are not received with precise timing.

In controlled environments, planning can be carried out in advance so that planning outputs can be delivered to controllers with precise timing. In uncertain environ-

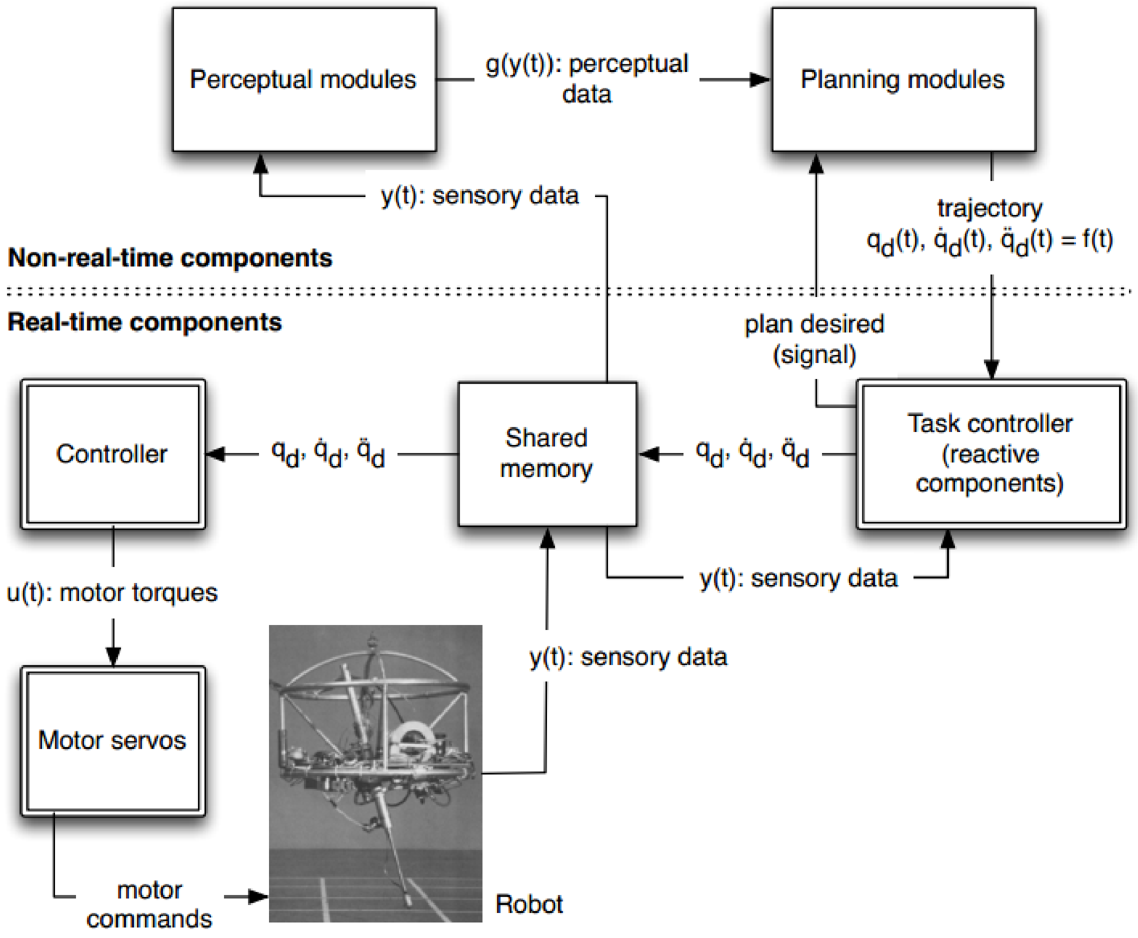


Figure 10: Architecture of a Physically Situated Robotic System

Named variables include $\boldsymbol{q}_d, \dot{\boldsymbol{q}}_d, \ddot{\boldsymbol{q}}_d$ (desired robot position, velocity, acceleration), $\boldsymbol{y}(t)$ ("raw" sensory data, *e.g.*, point clouds), and $u(t)$ (motor torques). Modules in double stroke (controller, servos, *etc.*) generally run at a regular, independent frequency while non-real-time processes (perception loops, planners, *etc.*) deliver outputs at a variable frequency.

ments where advance planning is impossible, the state of the environment may change significantly during planning such that the system can not act without entering into an uncertain and potentially hazardous situation.

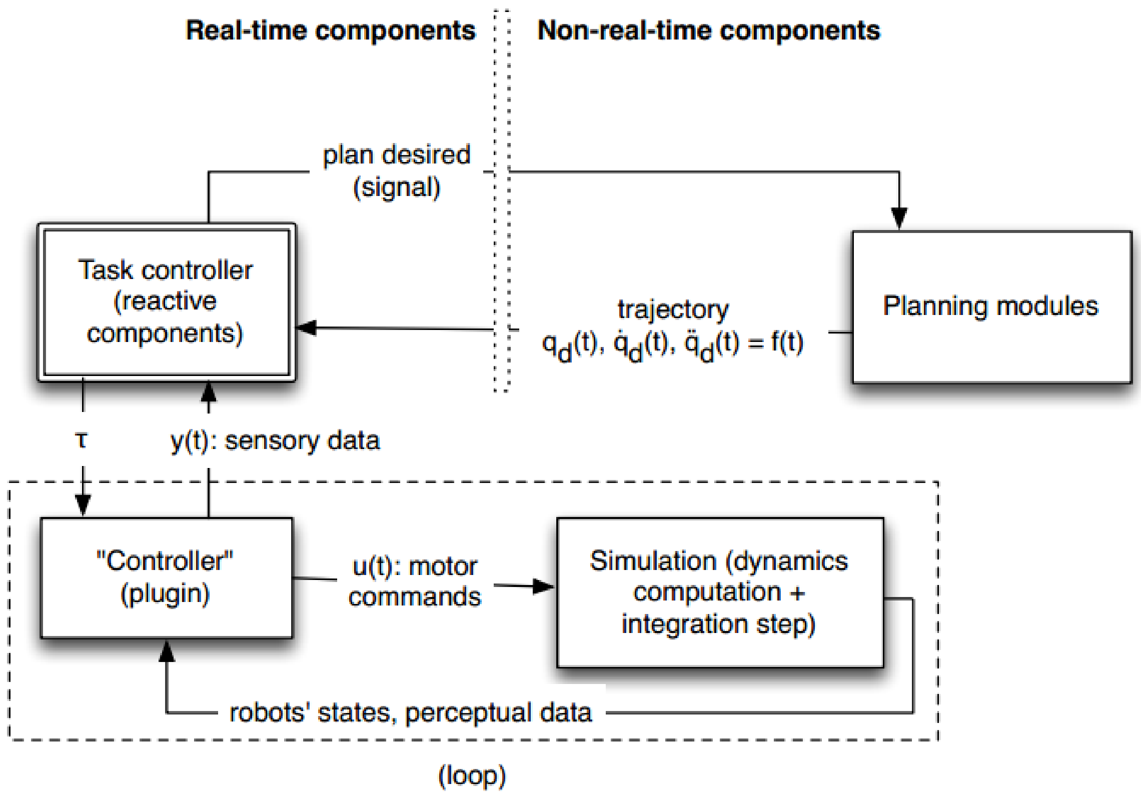

Real-time components
Non-real-time components
plan desired (signal)
Task controller (reactive components)
Planning modules
trajectory
$q_d(t), \dot{q}_d(t), \ddot{q}_d(t) = f(t)$
τ
y(t): sensory data
"Controller" (plugin)
u(t): motor commands
Simulation (dynamics computation + integration step)
robots' states, perceptual data
(loop)

Figure 11: Architecture of a Typical Simulated Robot

This illustrates the implicit architecture of a simulated robot system if the simulator and the control loop run in lockstep. This architecture eliminates the possibility of race conditions (the control loop only operates while the simulation is “frozen” in time, and *vice versa*), but is not representative of actual robot control architectures (embedded in Figure 10).

Current simulation architectures leverage a simple model whereby the dynamics executes for steps of time, and after each step the simulator invokes a call-back function which defines the planning and control for the system. This function can access dynamics state and it can actuate the system by adding virtual forces and torques. Importantly, the callback function can execute for an unbounded amount of time, without the time in the simulator progressing at all. The implication of this is that a significantly intelligent planner with correspondingly large computation times will execute on the *exact*, unchanging environment it used as input, thus ignoring planner computation time. If this system were transplanted into a real environment the physical state of the system would diverge from the plan, possibly irreparably, by the time such a planner returned a sequence of controls for the system.

We term the model typically employed by robotic simulators (*e.g.*, GAZEBO [38], SL [55], MOBY [14]) the *callback model*. This model, which is illustrated in Figure 11, is simple: the simulation requests motor commands from a controller (via a callback function) for some time t and state $\boldsymbol{x}(t)$.

In general, as is the case when higher order, adaptive (variable step), or implicit integrators are used (which facilitate more accurate, faster, and more stable integration, respectively) the controller may be called with non-monotonically increasing times t. The controller's loop can also run at any rate it desires, because the simulation blocks until the callback returns. This architecture allows the inexperienced roboticist to inadvertently employ computationally intensive algorithms in software components that require reactive behavior; such algorithms would only be feasible if physically situated robots were able to freeze the world, plan, and then resume the world's dynamics. Both of these issues lead to substantial effort when transferring software from simulation to real robots.

4.2 Background for Temporally Consistent Simulation

Simulators integrate user defined client components together with dynamics libraries into a single framework such that a particular scenario may be evaluated. We define *client processes* to be the set of controllers and planners evaluated in such a system. Multiple robots might be simulated, each with sets of client processes. Collections of robots and environmental obstacles form a scenario. Each of the robots has a set of sensors and actuators that are controlled by the client processes in the system.

4.2.1 Requirements for Temporally Consistent Simulation

Temporally consistent simulation attempts to ensure temporal consistency between different software components, and a simulation environment (*cf.* the notion of consistency for real-time aware, distributed shared memory accesses in Singla *et al.* [58]). The goal of temporally consistent simulation is not adhering execution to real-time, but rather ensuring consistency between the virtual progress of the dynamics, and the computational progress of the robotics software.

A difficulty in providing a temporally consistent simulation infrastructure is that the timing requirements for the client processes vary significantly, and that they require accurate timing from the system. A system's sensors and actuators often have a natural frequency at which they provide environmental data or take actuation commands. Correspondingly, controllers are often executed at a rate closely tied to those sensor and actuator frequencies (i.e. their rates of input/output), and thus execute in a *periodic* manner. Specifically, they are activated by the OS every N milliseconds, at which points they do their computation, and block waiting for the next periodic activation. Note that if a controller *overruns* its computation, it might finish execution only after an activation. This is often called *missing a deadline*, and can result in instability. In contrast, planners often execute irregularly, do as much computation as is required, and provide their output as fast as they can, but not on a tight schedule as with controllers. These computations are often called *best-effort*. Planners often also provide higher-level sets of commands to the controllers via Inter-Process Communication (IPC) channels.

In a temporally consistent simulation, the activations of periodic controllers must be as time-accurate as possible, and the computation for the planners must not interfere adversely with the controller's ability to meet deadlines. The most difficult timing requirements, however, derive from the interactions between client processes and the external world, and between client processes. If the simulated time within the dynamics gets too far ahead of the amount of time that the client processes have executed, then sensor data will reference data "in the future". Alternatively, if the simulated time in the dynamics lags behind computation, then actuator commands will be sent to stale dynamics state. Comparably, the planner and controllers must be temporally kept in sync for the same reason. This is at the core of temporal consistency: all aspects of the simulation environment must proceed at rates that are realistic, and in sync.

4.2.2 System Scheduling Toward Temporally Consistent Simulation

In attempting to address temporally consistent simulation, this research requires an infrastructure that can control not only the rate of progress of a dynamics engine (which is often provided naturally by it's API), but also of the *execution progress* of multiple client process computations. Put another way, the temporally consistent system infrastructure requires control over the scheduling of the system. Unfortunately, there is a *semantic gap* between what scheduling facilities the kernel of the system provides (often a black-box), and what is required by the simulation architecture. Many previous research projects in the area of operating systems have attempted to solve this problem. For example, [50] and [22] provide system infrastructures that are extensible, enabling normal user-level processes to define their own scheduling policies. However, they both require drastic system changes to provide these infrastructures. Alternatively, [3] and [1] attempt to create an environment in existing systems in which some of the timing characteristics can be controlled by user-level code. Our research continues with this trend by creating a multi-process simulation environment in which the *coordinator* plays the role that the kernel traditionally takes: it controls system scheduling (i.e. the interleaving of different client processes, and the dynamics), and communication. However, it is designed to execute at user-level using only the facilities and APIs provided by a POSIX-compliant OS such as Linux. Thus, users need not modify their underlying systems at all.

Real-time operating systems (RTOS) could be used to provide the same functionality as our temporally consistent simulation design on Linux, but would surprisingly not provide many additional assurances. The focus of RTOSes is often to control and bound latency for I/O. As temporally consistent simulation replaces traditional I/O with interfaces for coordinator and dynamics interactions, such RTOS facilities are superfluous.

The coordinator must address three main challenges: (1) how can Linux's POSIX-like API be leveraged to control scheduling and communication; (2) how can abstractions be provided to client processes so that they read sensor data, send actuator commands, and communicate via IPC normally; and (3) how can client computation be scheduled alongside the dynamics engine's virtual time?

4.3 OS Facilities for Temporally Consistent Simulation

This section describes our use of the Linux API to time processes, control scheduling, and provide intercommunication between processes in order to support temporally consistent simulation.

4.3.1 Temporal Accounting of Processes

The goal of the temporally consistent system is to ensure that for a number of client processes, and the dynamics, time progresses consistently for all. A key concept for temporally consistent systems is the maximum deviation in this progress, which we define in the following. We denote each of the client processes and dynamics as $\{p_0, \ldots, p_n\} \in P$. $p_i \in P$ has executed (or had simulated time progress) an amount of time e_i^t by time t within a given simulation infrastructure. Each periodic process blocks waiting for its next activation, thus essentially moving its own computational progress forward by the amount of time it waits, w_i^t. This wait time is common for controllers that activate periodically, but don't use all execution time until their next activation. Thus we define the temporal drift of the system, Δ, as such:

$$\Delta = \max_{\forall t}\{\max_{\forall p_i \in P}(e_i^t + w_i^t) - \min_{\forall p_i \in P}(e_i^t + w_i^t)\}$$

Intuitively, Δ is the maximum deviation in temporal progress between any two parts of a simulation. A perfectly temporally consistent system is one in that $\Delta = 0$, while a traditional callback-driven simulator with a planner that always executes for more time than a step in the dynamics has $\Delta = \infty$ as $t \to \infty$.

Commodity hardware features timing facilities that are based on somewhat granular units of time. For example, timer ticks provide preemptive execution to prevent system starvation from a single process, which occur at a minimum fixed interval (for example, 100 or 1000 times a second). POSIX-based operating systems provide APIs for accessing timers and execution accounting facilities. Thus, the granularity for executing processes on modern systems is somewhat large. If this is bounded by 10ms (the timer inter-arrival on our system), then $\Delta \geq 10$ms. Thus our temporally consistent system attempts to minimize Δ within the confines of the hardware and OS provisions. However, we have found that the choice of the OS facility used for this timing has a large impact on the accuracy of timing in the system.

Accounting for execution time. To track the execution progress of a client process, traditional OS facilities (such as those used in the `time` and `top` programs) have a very large granularity, and an unbounded error. Instead of relying on these very coarse grained mechanisms, we use the cycle-accurate *time stamp counter* register that is available on most processors. It is a 64 bit value that counts the number of cycles elapsed in the processor, and is accessible on x86 and x86-64 processors through the `rdtsc` instruction. To maintain accurate time using `rdtsc`, the system must (1) know the processor speed; (2) maintain a consistent processor speed (or, alternatively use “invariant time stamps” in modern processors); and (3) all client processes and the coordinator must remain active on only one, shared processor core. For (1), we read the processor speed from the `/proc` file-system, for (2), we disable all power saving and throttling features, and for (3), we confine the measured process to run on a single processor core using the `sched_setaffinity(.)` system-call family. Thus our system can cycle-accurately account for the execution time (e_i^t) of each client process.

Accounting for wait time. To track the wait-time (w_i^t) for a process, our system provides an API similar to `setitimer` which enables recurring, periodic activations. Controllers use our API to schedule periodic activations, and after their computation for a specific activation is complete, they become inactive waiting for the next activation. The system tracks this elapsed wait time until the process is again executed. Notably, this wait time does signal some temporal progress for those controllers, even though it does not include computation time, thus why we consider it in the calculation of Δ.

Granularity of preemption. For any scheduling system to control the execution of unknown computation (that might, for example, contain an infinite loop), preemption is required. A significant flaw of the callback model is that it executes the planner non-preemptively – the simulator cannot stop the planner when it has executed for too long. The hardware provides timer interrupts as its basic mechanism for preemption. POSIX provides a number of facilities for notification of timer interrupts. Our coordinator uses signals associated with the timer to receive these notifications. When the hardware causes a timer tick, the OS vectors it into a user-level signal that switches away from the previously executing client process (*e.g.*, planner), and to the coordinator, where scheduling decisions can be made. This, combined with the accurate execution time explained above, provides the temporally consistent system with the main facilities it requires to manage timing.

4.3.2 Scheduling Simulation Processes

The default scheduling policy in Linux is `SCHED_OTHER` that makes no guarantees on when any thread in the system will make progress. In contrast, it also includes two real-time policies for first-in-first-out, non-premptive, fixed priority scheduling, and preemptive (round-robin), fixed priority scheduling—`SCHED_FIFO` and `SCHED_RR`, respectively. These policies are *predictable* in that if two processes both want to run on the CPU, the higher priority one will always be chosen to execute. A process can be set to be scheduled using any of these policies via the `sched_setparam` system call.

Context switching. We take advantage of the predictable behavior afforded by these kernel scheduling policies to implement our own scheduling policy in the coordinator. The coordinator itself always executes at the highest priority. The client process it wants to switch to will be given the next highest priority. To finish the switch to that process, the coordinator will block waiting for timer interrupts, actuator commands, blocking notifications, and IPC (blocking on multiple sources in POSIX can be done with `select`). If any of these are detected, it will wake, and immediately activate (as it is highest priority). Whenever the coordinator executes, it makes a scheduling decision about which client process/dynamics should run next to optimize for a minimal Δ.

Process blocking. In attempting to override the scheduling policies of the kernel, the coordinator must consider the case when a client process blocks. For simplicity, in this work, we assume that client processes only block waiting for timeouts (periodic controllers), or waiting for sensor data. Without accounting for blocking, the coordinator might lose control of the system: if the process that is supposed to be executing instead blocks, then the kernel will take over and choose the next highest thread to execute, which might not be in the time consistent system, thus invalidating the coordinator's execution accounting.

4.3.3 Communication Between Simulation Processes

Communication between client processes (*e.g.*, the planner sending commands to the controller), sensor data requests, and actuator commands require the coordinator to mediate the communication. Each client process is given access to two pipes that are used to (1) block the process waiting for an event (*e.g*,. IPC, sensor data), or (2) send a notification to the coordinator that the process is sending data (*e.g.*, IPC, actuator commands). The coordinator is awakened by such notifications and can decide where to copy the data (it is in shared memory) or how to manipulate the dynamics.

4.4 Temporally Consistent Simulator Design

The time consistent simulator must manage multiple conflicting goals. On the one hand, the timing requirements of controllers require the meeting of deadlines (*i.e.*, an accurate activation time), and on the other, temporal consistency is required to have a clear mapping between actual system execution, and simulated execution. As the simulated environment must support multiple robots and varied software infrastructures with rich communication structures, it must be highly flexible. This section covers the implementation of the infrastructure, and details how it integrates with the OS facilities from Section 4.3.

4.4.1 Hierarchical Scheduling and Threads

To handle the required generality, we use a hierarchical scheduling framework [50]. Such systems define a tree of schedulers. The leaves of the tree are client processes, and the dynamics. When activated, the *root* scheduler determines from its children which to execute (*i.e.* it makes a scheduling decision), and does so using a polymorphic method invocation to `dispatch` the child. If the child is a leaf, then the dispatch function will either (1) use the context switch mechanism described in Section 4.3, or (2) make an invocation into the dynamics to step the simulated time forward. However, as the system is hierarchical, the dispatched child could be a scheduler itself. The key insight here is that each of the schedulers in the hierarchy can define different scheduling policies.

Scheduling policies. There are two policies we use in the system, one to maintain minimal temporal consistency, and another to do strict priority-based scheduling. As a general rule, the schedulers close to the root are concerned with temporal consistency, while those that represent an actual scheduler on a robot are concerned with maintaining accurate timing for the system controllers, thus placing them at a higher priority than the planners. Though more scheduling policies could be added, we found that these are sufficient.

Example use of hierarchical scheduling. The system set-up we use in Section 4.5 includes a dynamics engine and two robots, one with both a planner and a controller, and the other with a simple controller. All of the three threads for the robots, and the dynamics must be scheduled. Thus, we organize the system with a single root scheduler for consistent timing between each robot, and the dynamics. The robot with both the planner and the controller has each under a scheduler with the fixed-priority policy, with priority going to the controller. The hierarchical arrangement of schedulers and of dynamics and client processes is essential to properly schedule given the different goals of different parts of the system, and to enable the simulation of complex, possibly multi-robot systems.

Maintaining proper accounting in the hierarchy. Just as scheduling decisions follow a chain from the root to a leaf, the accounting for execution time and progress must go from leaves down toward the root, so that scheduling decisions can be made at each scheduler based on an accurate rendition of how much temporal progress all of its children have made. For this, we aggregate the execution times of all children, unless they are all blocked waiting for sensor data, in which case we determine that child's progress to be the minimum of those block times.

4.4.2 Coordinator Design

The coordinator is the heart of the system and orchestrates all execution. The root scheduler is maintained by and the hierarchical scheduling policy is executed in the coordinator, and when a dispatch is made to a client process, the coordinator goes through the following steps: (1) swap the priorities of the previously active client process, and the one we want to switch to (Section 4.3), (2) take a time-stamp reading (Section 4.3), and (3) block the coordinator (on `select`) waiting for an event (timer tick or request for sensor data). As the coordinator (which is highest priority) blocks, the system naturally switches to the new process, thus completing a context switch. Unblocking the coordinator (and subsequent blocking of the active client) is accomplished by writing notifications to the pipes that will wake the main coordinator thread of execution. A client process explicitly notifies the coordinator of a servicing need and scheduling demands by sending timestamped `read`, `write`, and `idle` notifications to the coordinator which trigger a yield by the client process and rescheduling per its scheduling policy. The coordinator also implements a real-time monotonic timer via `timer_create` that periodically sends a timestamped `timer` notification. The `timer` notification ensures the coordinator interrupts a long running (typically *best-effort*) client process such that all client processes (especially *periodic*) are given fair access to the processor on a regular basis and no client process can starve all others.

Coordinator initialization. System boot-up is a delicate process that we detail here. Upon initialization, the coordinator is bound to the CPU, set as a real-time process with highest priority ℓ_0 (Table 1), opens pipes for IPC, opens the shared

Priority	Level	Process	Description
p_c	ℓ_0	coordinator	highest real-time priority for the OS
$p_c - 1$	ℓ_1	active client	the currently dispatched client
$p_c - 2$	ℓ_2	block detection	reserved for a block detection process
$p_c - 3$	ℓ_3	waiting clients	all other waiting (or blocked) clients

Table 1: Process Priority Assignment.

memory, intializes the dynamics system, creates all client processes, and initializes the timer. Creation of a client process involves wrapping the process with a client thread, forking a new process, scheduling with the system as a real-time process and with real-time priority ℓ_3, binding to the same CPU as the coordinator, disabling console interaction, and launching the executable file via `execl`. Console interaction is disabled after the fork by forwarding `stdin`, `stout`, and `stderr` to `/dev/null` to minimize blocking system calls within these processes, which might disturb the coordinator's control over timing. When a client process is dispatched, the coordinator raises the client system priority from ℓ_3 to ℓ_1, and yields to the dispatched child by `select`. When the client process publishes any notification to the coordinator, the coordinator unblocks, preempts the client process, and lowers the client system priority from ℓ_1 to ℓ_3.

4.4.3 Client Process Implementation

A client process is an external main function program that must provide facilities for opening the shared buffer, for sending `read`, `write`, and `idle` notifications on prescribed channels, and for executing its own computation code. A client process must be preregistered with the simulation such that it is linked to the corresponding dynamic body, is described as a controller or planner, is classified as either periodic or best-effort, and has IPC facilites prepared. Forking the coordinator and executing the external program, inserts the external program into the process space of the coordinator as a child process and inherits the established IPC channels. Notifications of process `read`s indicate requests for simulation state (*e.g.*, reading sensor data). The coordinator services these `read`s using shared memory to pass data. Client process `write`s correspond to either a controller sending commands to actuators, which are interpreted to manipulate dynamics state, or correspond to a planner sending the plan to the controller via the coordinator. Finally, client processes send `idle` notifications to the coordinator to yield until the next activation (*e.g.*, for periodic controllers).

4.5 Comparisons Between Conventional and Temporally Consistent Simulators

We carried out multiple experiments using our temporally consistent simulator at different phases of development. After we implemented support for real-time client processes but before we implemented support for non-real-time client processes we used a simple pendulum swing-up scenario to compare performance between a conventional callback simulator and our time consistent simulator. After implementing support for non-real-time client processes, we experimented with a reactive planning, predator-prey inspired scenario to demonstrate the true temporal cost of planning and the time sensitive relationship between planner and controller.

4.5.1 Experimental Comparison of a Pendulum Swing-Up

We assessed the time consistency of a callback simulator and our time consistent simulator using a PD controller to perform the swing-up and balancing tasks from the downward equilibrium point for a simple pendulum. Our experiments were run on Linux kernel 3.2.0 ("vanilla" Ubuntu 12.04) using a 2.80GHz Intel Xeon quad-core processor. We used *ROS Groovy Galapagos* for all ROS-based experiments and Moby [14] as the dynamics library; we did not use methods for simulating perception.

Conventional Callback Simulator Analysis. To gauge the performance of simulators that use the callback model, we developed a PD controller via a ROS service that interfaces to Gazebo. We then evaluated the system-time of the control code activation with respect to Gazebo simulation time. For each simulation iteration, the controller computes and submits motor torques as a function of pendulum joint position and velocity. We recorded the simulation time (*i.e.*, the virtual time according to the simulator) on each controller invocation, and we recorded the CPU time spent in the controller by querying system `/proc` statistics for the controller.

This experiment shows that for every second of system time that the controller runs, Gazebo advances simulation time approximately 1/10th of a second. Our results demonstrate that Gazebo and callback based simulations in general do not maintain real-time system requirements, and therefore controllers designed and tested exclusively in callback based simulations will not match real-world operation. The practical implication of this finding is that *controllers that need to synchronize actions to time (*i.e.*, controllers that are tightly coupled to time) cannot be expected to exhibit identical performance in simulation and* in situ.

Figure 12 shows the disparity between the system time made available to the controller and the time maintained by the simulator. Figure 13 shows the instantaneous change in observed time between the controller and the simulation. We note that not only is there a disparity between controller and simulator times, but that *this disparity changes over time* (indicated by the derivative being far from zero).

Temporally Consistent System Analysis. As a second experiment, we implemented the PD controller for the pendulum swing-up and balancing tasks under our temporally consistent system using the `TCS` branch of the Moby simulator. The PD controller was to operate at a frequency of 1000Hz over 10,000 simulation cycles. The simulation time was maintained by the Coordinator.

Figure 12 shows that over the course of the simulation, dynamic scheduling of the controller guaranteed control code to operate at the expected frequency within a

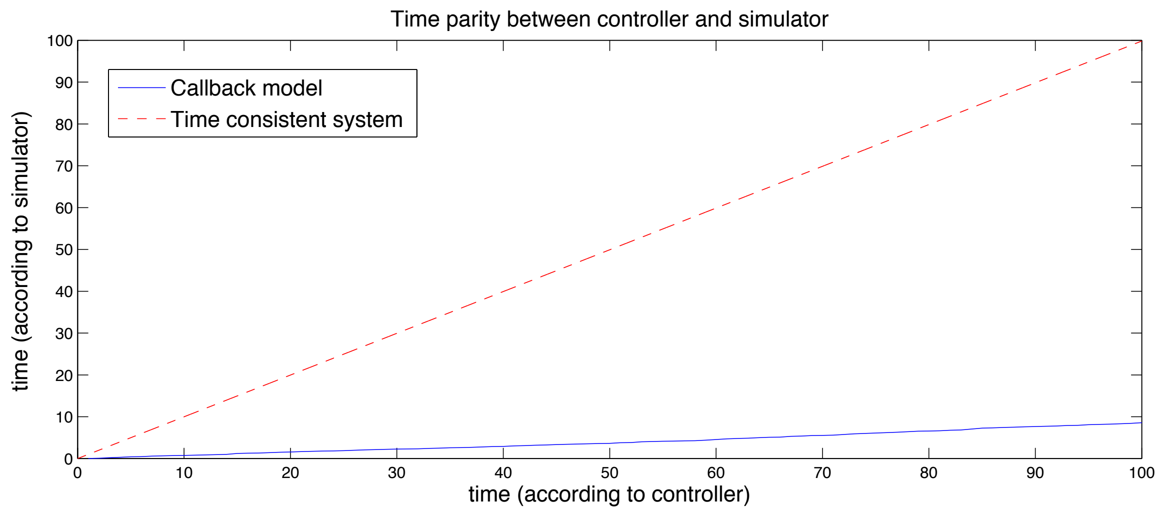


Figure 12: Temporal Comparsion of Conventional and Temporally Consistent Simulation

Data plot (see Section 4.5) using the callback model with GAZEBO (blue/solid) and our temporally consistent system (red/dashed); the latter exhibits exactly the behavior we were seeking. The callback model data shows that GAZEBO advances approximately 1/10th of a second for every second that the controller advances (and thus we should not expect simple transference from simulation to *in situ*).

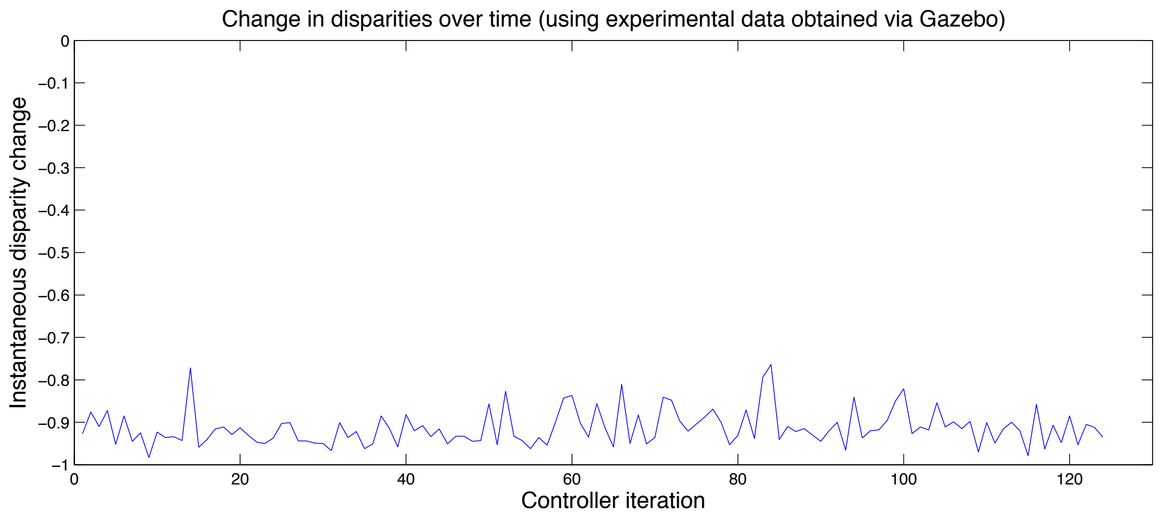


Figure 13: Instantaneous Temporal Disparity of Conventional Simulation

Plot depicting *instantaneous changes* in time disparities using data plotted in Figure 12; note that the disparities slowly decrease over time.

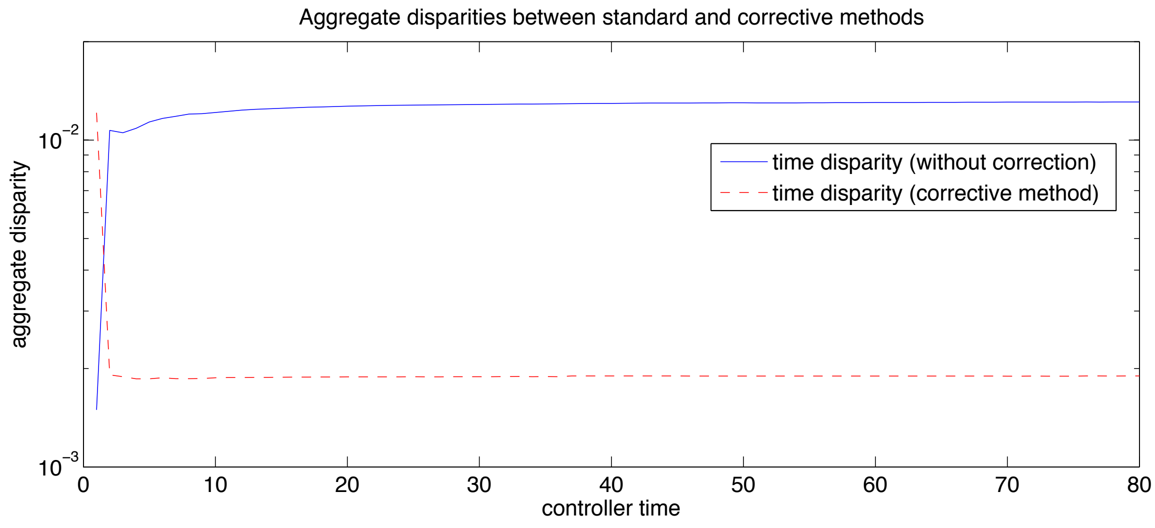


Figure 14: Aggregate Relative Error in Temporally Consistent Simulation

Plot depicting aggregate *relative* error (with respect to virtual time) with our temporally consistent system. Our initial system (without the correction mechanism described in [66]) exhibits some drift. Our system with the correction mechanism exhibits very low relative and absolute error (less than 2% absolute error over the entire 100s simulation).

small (under 2%) margin of error.[2] Figure 14 shows that the aggregate relative and absolute errors are small (although a controller may overrun its interval by as much as 2%); the plot shows that relative error is an error of magnitude greater without the corrective method.

The experimental data shows that the temporally consistent system is indeed consistent with time. Additionally, we note that the system allows us to effectively time control loops and thus, by testing, guarantee—again, to a 2% margin of error—that a control loop will run within its allotted time when moved to a real-time OS (assuming comparable hardware).

[2]In the initial temporally consistent simulator development, we had not implemented the hierarchical scheduler and instead used a relatively simple system to correct temporal drift [66]. The hierarchical scheduler described in Section 4.4 eliminates the need for drift compensention.

4.5.2 A Demonstration of Temporal Consistency: Predator and Prey

This experiment is intended to illustrate the performance discrepancy between systems using callback functions and our time consistent system when non-real-time client processes are integrated into the system. The experiment has been designed to reflect our experience with building software for both simulated and physically situated robots; this decision results in a few discrepancies between the time consistent and callback-based systems that will be noted below. The time consistent simulator described throughout the remainder of this section relies on hierarchical scheduling as described in Section 4.4 and does not require any drift compensation.

Our experimental scenario uses a predator-prey scenario with two identical "space ships" (*i.e.*, rigid bodies moving freely in SE(3) via application of forces). The ships are constrained to move within a cubic region of space; when a ship attempts to move out of this region, a spring-like penalty force pushes it back toward the free region.

Predator and Prey Behavior. The prey is driven by a simple control policy, which enacts either a random walk (we save the seed so that we can reproduce the walk across trials) or a fleeing behavior, depending on the distance of the predator. The prey flees by moving directly away from the predator using limited force.

The predator uses kinodynamic planning to chase the prey by exploiting the latter's deterministic behavior when the predator gets sufficiently close. Indeed, given ample planning time, the predator should be able to plan to intercept the prey by using the prey's deterministic movement model and an inverse dynamics model (that determines the requisite forces to achieve a target acceleration).

Planning and Control. We instituted our own kinodynamic planning mechanism which applies controls, integrates its models of the predator and prey forward in time, and finds a plan that brings the predator closer to the prey. Our initial efforts used OMPL, but the inherent multi-threaded nature of the library and its use of wall-clock time for determining when the planner should terminate confounded our system's efforts to schedule the planning process. Using wall-clock for process timing assumes that the process will not be scheduled-out, so the planning time parameter in running OMPL in the context of a real-time system can only be considered an idealized upper bound. The planner is allowed to execute for a maximum time (1.0s) and the resulting plan is not executed beyond a maximum duration (0.1s); beyond this point the open loop execution of the plan by the predator tends to lead to it becoming dynamically unstable.

The planner is called differently on the time consistent and callback-based systems. On the callback-based system, the planner is called only when all of the commands from a previous plan have been executed (or the plan has become stale by going over the maximum allowable duration). The time consistent system calls a planner in a manner analogous to operation on a real robot: (1) before a plan arrives, the predator executes "no-op" commands (*i.e.*, it applies no force and no torque); (2) the planner attempts to find a plan (the predator continues to execute "no-op"s at this time); (3) when a plan is found, the predator begins executing the plan and immediately calls the planner to begin planning again; (4) the planner keeps executing that plan until the sequence of commands is complete or the planner notifies the controller that a new plan is available.

The predator controller uses a composite feedforward (*i.e.*, the planned commands) and negative-feedback controller to account for error between its current state and the desired outcome. The prey uses a simple control policy only.

Temporally Consistent and Callback-Based System Configurations. The time consistent system was built on top of an otherwise unmodified version of MOBY. For comparison we used two callback-based systems, "vanilla" MOBY and GAZEBO/ODE.

Each simulation was run with a 0.01s dynamics time step, a maximum planning time of 1.0s, a planning step size of 0.01s, and both controllers running at a frequency of 100Hz. Our experiments were run on Linux kernel 3.2.0 ("vanilla" Ubuntu 12.04) using a 2.80GHz Intel Xeon quad-core processor.

Experimental Specifications. All scenarios start in the same configuration with the predator and prey halted and separated by ten meters and a flee triggering distance of five meters. Scenarios were simulated for twenty seconds of simulation time, and each experimental trial consisted of running the scenario with identical random seed for the prey using the three systems: Gazebo, "vanilla" Moby, and modified Moby (the time consistent system).

Experimental Analysis. The results of our experiments, depicted in Figure 15, show that the simulations based on the callback model yield virtually identical statistical behavior while the time consistent simulation exhibits dissimilar behavior. Because the simulation state is frozen during planning in the callback model simulations, the predator is consistently able to plan from its current state to the current state of the prey, which allows it to maintain close proximity at nearly all times. The statistical distributions for the callback-based systems are centered within the flee triggering distance with a maximum distance equal to the initial distance. In the time consistent simulations, the predator is able to approach the prey for only short durations and the statistical distribution is centered more closely to the starting distance and exhibits high variance. Animated renderings of the simulations show that the predator tracks the prey very closely in the Gazebo and "vanilla" Moby simulations while the predator generally undershoots or overshoots the prey's position in the time accurate system.

The predator in the callback-based systems is able to maintain much smaller distances to the prey solely because the predator's planner is able to execute propor-

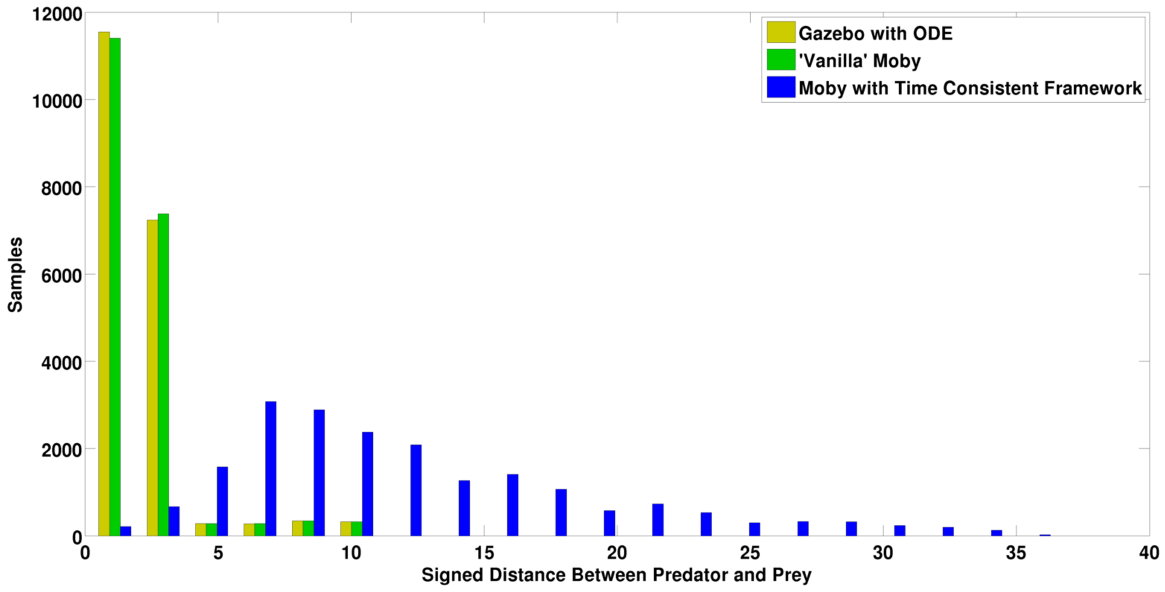


Figure 15: Instantaneous Distance Between Predator and Prey

Histogram showing the instantaneous distances between predator and prey over ten trials (2,000 samples per trial). Systems based on the callback model are effectively able to plan while the predator and prey are frozen in time, while the time consistent system must plan and act in real time. Consequently, the predator is able to stay much closer to the prey in the systems based on the callback model.

tionally for much longer without the danger of plans becoming stale. At the end of 20 simulated seconds, the planner in the traditional callback-based systems consumed on average 188 seconds, thus yielding $\Delta = 168.01$ seconds. The ratio of planner execution time (1.0s) to planner frequency (0.1s) indicates that the planner runs for 10 times longer per second than the simulated time progresses and approximates $\Delta = 10t$. In contrast, $\Delta = 0.003$ seconds for the time consistent system; though Δ is non-zero (due to intrinsic hardware limitations), its value is independent of the time that the simulation runs.

In a follow-up experiment, we measured the overhead of our components for enforcing temporal consistency. For a single trial, the overall system ran twenty seconds of simulation time in a real-time of 21.22 seconds during which the system spent 0.42 seconds coordinating, 20.21 seconds running planners and controllers, and 0.59 seconds stepping dynamics. From this result, we estimate our framework adds 2% of overhead, which we argue is acceptable.

4.6 Future Work in Temporally Consistent Simulation

The need for temporally consistent simulation has become a topic of discussion anecdotally and in published research. At least one research effort has implemented a temporally consistent framework [47] using the same fundamentals such as proportional progress and centralized coordination. Their research effort differs in that it is attempting to produce a temporally consistent distributed simulator.

For future work, we will increase time accounting accuracy by detecting unconstrained blocking system events (which most non-real-time software triggers) and client process unblocking events (which will allow us to better support existing libraries like `OMPL` without requiring modifications to the libraries themselves). To take full advantage of current system architectures, we will also scale the coordinator scheduling system to utilize multiple cores and multiple processors. We will support simulations for which time does not proceed monotonically (like those that use adaptive integration).

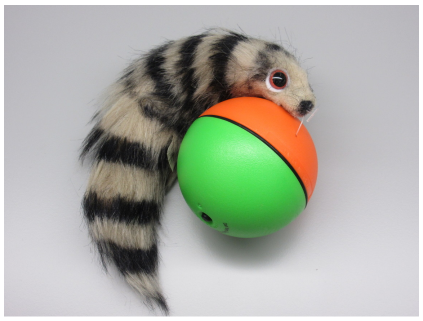
Chapter 5- Validating Simulations Involving Contact and Friction with a Wild Robot

Interactive robotics tasks such as grasping and locomotion are contingent upon contact and friction. While dynamics simulation of ballistic rigid body motion is well known to be generally consistent with natural behavior, simulations involving contact and friction are prone to producing physically implausible results. Given that the contact and friction models have been validated by mathematicians, physicists, and engineers on simple systems over centuries of study, one must wonder where the software that implements these models fails. Indeed, the past decade has seen the continual release of new simulation software libraries, each of which promises to eliminate the artifacts but in hindsight has improved the status quo little, if any.

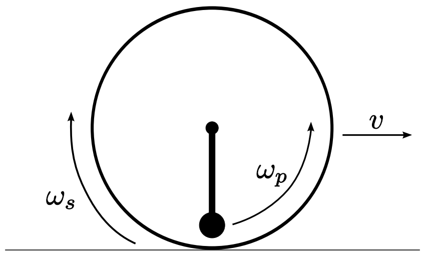
To validate simulations involving contact and friction, roboticists require data from real world systems that are subject to complex dynamics yet simple enough to comprehensively study. While a scenario involving a block on an inclined plane [57] [7] is a meaningful verification scenario or a potential validation scenario for coulomb dry friction, this scenario alone can not be relied upon to entirely validate contact and friction. Roboticists need simple enough scenarios, *i.e.* low state space and measurable, that focus on contact and friction but complex enough that they are difficult to simulate without accurate models, *i.e.* wild behavior. An ideal scenario consists of a robot with few degrees-of-freedom, has a primitive collision model (a sphere is the simplest and most reliable collision primitive), and can be modeled with a high degree of accuracy.

This chapter details a validation scenario for multi-rigid body dynamics by studying a mechanically simple but highly dynamic, real-world robot whose motion is primarily driven by contact and friction. This study involves examination and modeling of the robot, recording state data from the robot *in situ*, error correction of recorded data, simulation of the robot, and examination of a novel, machine learning validation approach for this scenario. We believe that successful simulation of real-world robot behaviors such as the robot in this scenario will lead to better robotic simulation, because we have found little qualitative agreement between the robot's observed behavior in simulation and in situ, *even though this robot seems well modeled by a multi-rigid body system.*

5.1 The Weazelball is a Wild Robot



(a)



(b)

Figure 16: The WEAZELBALL

(a) An off-the-shelf WEAZELBALL. (b) "Forward" velocity v develops from friction between shell and environment due to angular velocity ω_s, a response to the torque generating pendulum angular velocity ω_p.

This chapter studies the validation problem for the aforementioned domain using an inexpensive, widely available toy. Mechanical toys producing esoteric physical phenomena have influenced dynamics research for years. Examples include the drinking bird toy, which helped validate thermodynamic simulations [30], the woodpecker toy studied by Pfeiffer [51] that was used to construct and validate rigid body impact models [52], and the rattleback toy used to validate multibody dynamics simulations with contact and Coulomb friction [45].

The subject of our investigation is a spherical robot called a WEAZELBALL (WB), shown in Figure 16(a), which meets a variety of desirable criteria: wide availability, low cost, simple mechanism, and complex (wild) behavior. It has been the subject in a number of research efforts focused on controlling aggregate behavior of "wild bodies" [6] [5] [26]. The Wb basic principle and design was patented more than a century ago [70], operates by a combination counter-torque drive and offset center-of-mass, and is both underactuated and nonholonomic. A motor drives a bob around an axis fixed inside a spherical shell. Producing a torque to rotate the bob also produces a response counter-torque that rotates the shell in the opposite direction. Friction between the shell and environment causes the Wb to roll "forward" relative to the rotation of the bob. Figure 16(b) illustrates the relationship between bob angular velocity, shell angular velocity, and shell forward velocity. In an unconstrained environment, counter-torque drives the robot forward with respect to the rotational motion of the bob due to friction between the shell and the environment, and the robot wobbles from side-to-side due to rotation of the bob's offset center-of-mass around the shell's center. A situated Wb tends to roll forward when unobstructed and may undergo wild direction changes reminiscent of pirouetting, rocking, and tumbling upon collision with an obstacle.

Due to the wild behavior of the Wb, we do not expect simulations to be able to reproduce medium- or long-scale telemetry data of the physically situated robot, though we do believe it is possible to capture Wb behaviors qualitatively within simulation. Rather than comparing telemetry data, Wb simulation might instead be validated by comparisons such as whether the characteristic wobble is evinced and whether the exploration is ergodic. We propose a novel, regression-based validation scheme to avoid the need to classify and locate such behaviors.

5.1.1 Anatomy of a Weazelball

The shell and bob are the two primary Wb components and consist of a number of smaller parts. The shell consists of two hollow hemispheres and an o-ring. The

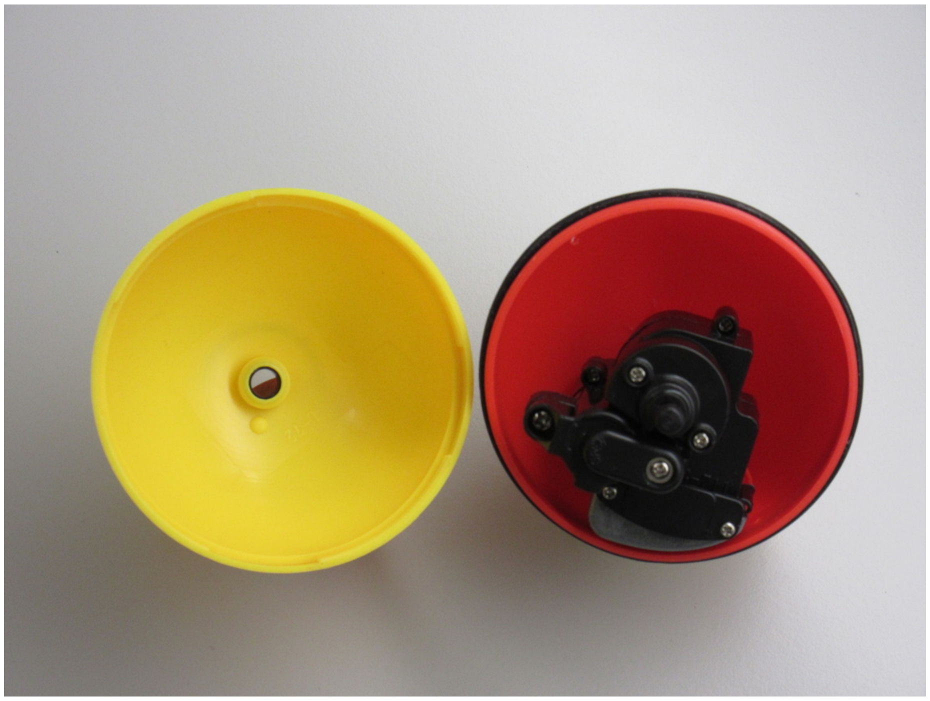
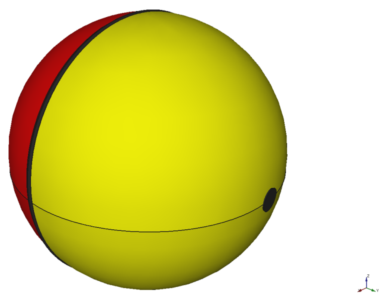


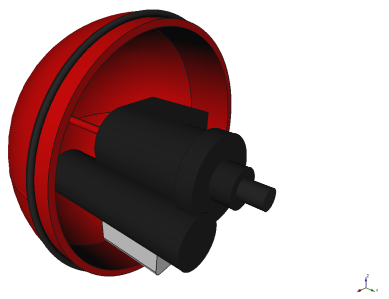
Figure 17: Interior View of the Weazelball

The interior view of the “free” hemisphere (left-yellow) and the “fixed” hemisphere (right-red)

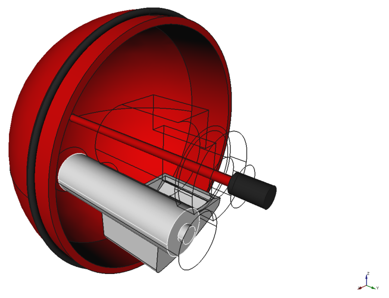
hemispheres thread together to form the shell and the o-ring sits in a groove along the seam formed by the hemispheres. The bob consists of plastic housing, AA battery, metal weight, plastic gearing, switch, small screws, wire, and DC motor. We call the Wb bob the *motor assembly*. Unthreading the shell separates the Wb into two pieces: one hemisphere joined to the motor assembly and a hollow hemisphere. We use the terms *fixed hemisphere* to refer to the hemisphere joined to the motor assembly and *free hemisphere* to refer to the hollow hemisphere. Figure 17 shows the separated Wb. A metal pin cast into the pole on the inside of the fixed hemisphere acts as mount for the motor assembly. The actuator is formed by a gear mounted on the pin and a simple gear train connected to the motor. A port cast into the pole of the free hemisphere accepts the power button on the motor assembly which enables the switch to be toggled without opening the Wb. The motor assembly rotates around



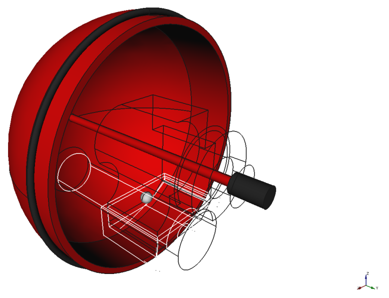
Shell Mesh



Motor Assembly Mesh



Battery and Weight



Combined Motor Assembly CoM

Figure 18: The Weazelball as modeled by Solidworks

the polar axis when the switch is closed.

5.1.2 Modeling the Weazelball

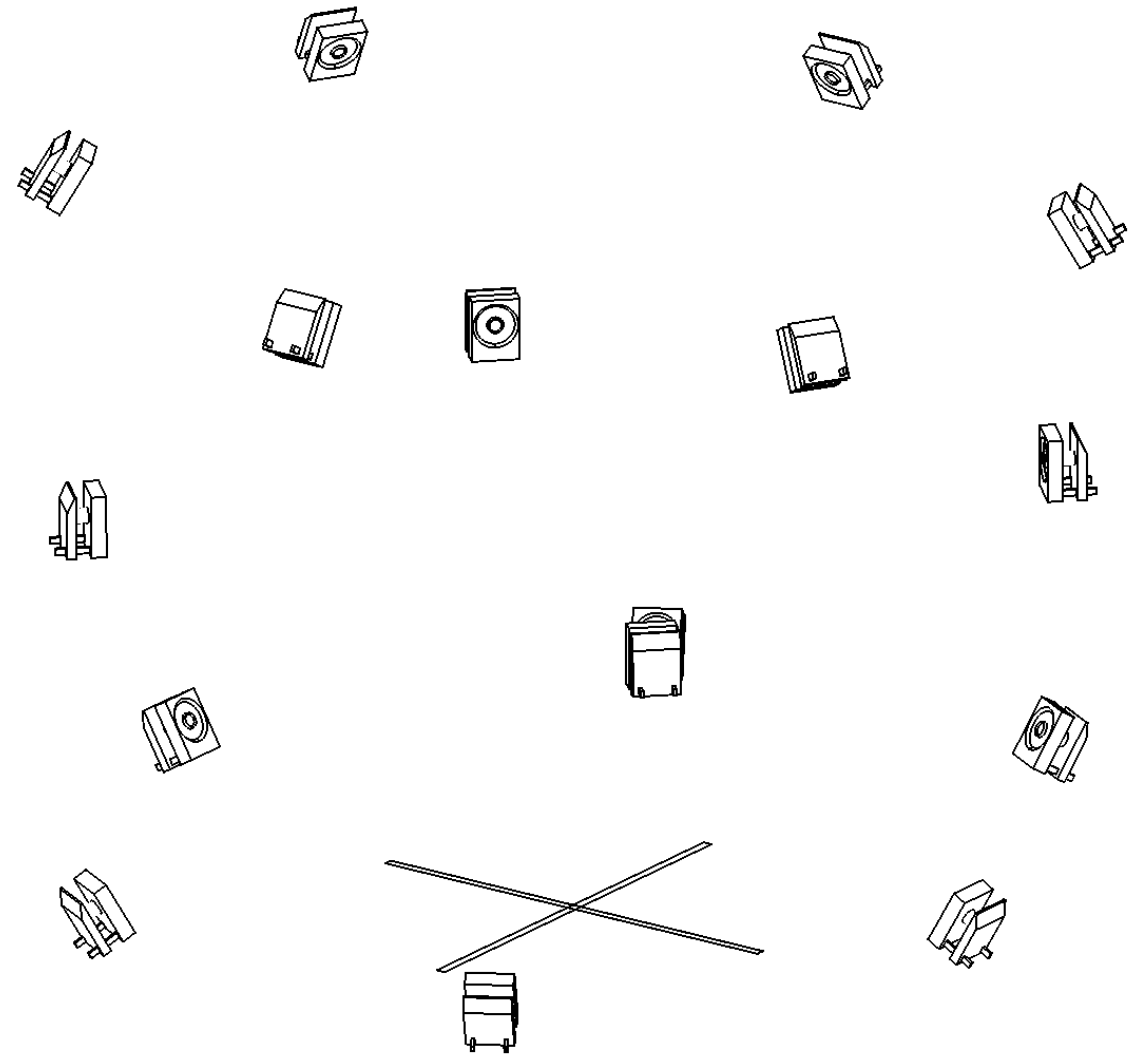
We model the Wb kinematically as two links constrained together by a continuous revolute joint. We model the shell's inertia as a hollow sphere with a diameter of 82mm, a thickness of 3mm, a mass of 46g, and a center-of-mass located at the center of the shell. Because the motor assembly consists of a number of components with odd shapes and significantly varying densities, modeling the motor assembly as a primitive with uniform density is insufficient. We model the motor assembly using Solidworks instead, refer to Figure 18. Solidworks allows parts with different material densities to be modeled individually and then combined into a single assembly. Solidworks also allows an assembly to be queried for a single inertial tensor and center-of-mass with respect to a user defined reference frame. Rather than modeling every part in the motor assembly, we simplify by categorizing parts into three groups of solid components: battery, weight, and housing. We measure the battery and weight and model them individually. We measure the housing, motor, and all other parts as a single component with homogeneous density. We measured the mass of the battery to be 24.2g, the mass of the weight to be 23.6g, and the mass of the motor assembly to be 38.7g. We assembled the components and queried SOLIDWORKS for the inertial tensor and center-of-mass using the center of rotation as the center of the inertial frame. We generated COLLADA files (DAE format) for visualization. We provide all modeling data generated through SOLIDWORKS in the repository.

5.2 Measuring Weazelball State

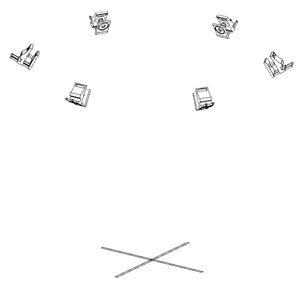
Simulation validation relies on analyzing state data, so accurate estimates of WB telemetry data are desirable. An off-the-shelf WB is not designed for state measurement. The shell has few features that can easily be tracked and the opaque shell obscures bob state. Modifications to the WB can be expected to alter the dynamic behavior. For example, adding instrumentation changes mass distribution and adding new features to the shell alters surface geometry and friction characteristics, which may be expected to modify contact behavior. We must accept that some modifications are necessary to measure WB shell and bob state, so we select modifications to minimize changes to mass distribution and surface features and to maximize reproducibility in the data gathering process. This section describes the measurements conducted to measure the state of the shell and the state of the bob.

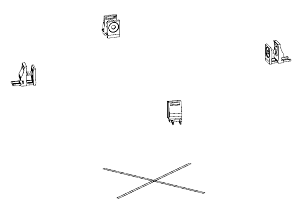
5.2.1 Estimating Weazelball Shell State Using Vicon Motion Capture

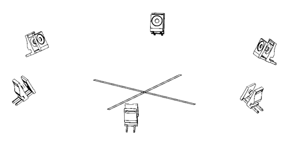
We use a VICON array to track the WB as it moves, which allows us to capture the pose of the WB shell over time. We set up a VICON array consisting of sixteen motion capture cameras focused into a cube of space two feet above the ground and roughly six feet on a side. Figure 19 illustrates the camera arrangement we use to support this



Full Vicon array







High ring Middle ring Low ring

Figure 19: Vicon Configuration For Weazelball Motion Capture

work. In the center of this space, we marked a large, orthogonal *central axis* on the floor. Around the central axis, we arranged cameras focused on a point approximately 4ft from the floor into three rings: a high-level ring with cameras focused downward, a mid-level ring with cameras focused level to the floor, and a low-level ring with cameras focused upward. We include six cameras in the high ring, four cameras in the mid ring, and six cameras in the low ring.

We selected our camera arrangement to share one camera calibration between two activities: *model registration* and *motion capture*. We define model registration to encompass all tasks involved with developing a Vicon *tracking model*, discussed in the following paragraph, and we define motion capture to encompass all tasks involved with gathering motion data from a subject. Calibrated cameras may be deactivated and reactivated into the active camera set without the need to recalibrate the array (as long as the cameras are not moved).

A Vicon "tracking model" consists of the set of markers registered with the Vicon system for a given subject and is used by Vicon to derive a subject's pose from the pattern of markers detected by the array. The reference frame for the tracking model, *i.e.* the *model frame*, is defined by a combination of the position of a manually assigned "center" point and the orientation of the array's origin at the time of model registration. A tracking model must be defined by a unique arrangement of markers in order for Vicon's Tracker software to be able to accurately localize the subject. If two pairs of markers have the same approximate Euclidean distance, Vicon will localize the tracking model with an arbitrary pose relative to one of the marker pairs. During registration, if Vicon detects a marker arrangement where more than two markers have the same approximate distance, the system will warn of a *symmetric* tracking model. A minimum of three markers must be visible to the array when a sample is recorded; otherwise, Vicon either will record a pose with an incorrect orientation or will fail to record a pose.

We activate all cameras when registering the Wb tracking model, and we activate only the high and mid cameras during motion capture. We call the cameras used for model registration the *registration array* and the associated reference frame the *registration frame* and the cameras used for motion capture the *capture array* and the associated reference frame the *capture frame*. The high and low cameras allow Vicon to detect markers on the Wb from above and below without the need to manipulate the Wb during model registration. The low cameras are deactivated in the capture array because they are occluded by the environment.

A standard, spherical motion tracking marker is unsuitable for the Wb as placing spheres, even small ones, on the surface of the Wb vastly alters its locomotion. We use flat retro-reflective tape instead. Tape does modify contact parameters but the impact on locomotion seems much less than that from using spherical markers. We observed that the tape marked Wb behaves similarly to an unmarked Wb.

Marker visibility is predominantly subject to occlusion, *i.e.* where a marker is occluded from a camera by an opaque object. Marker triangluation accuracy increases as the number of cameras detecting the marker increases. A Vicon array works best with a marker detected by at least three cameras but will still attempt to "triangulate" a marker detected by two cameras. If fewer than two markers are detected by the array, Vicon will not record state.

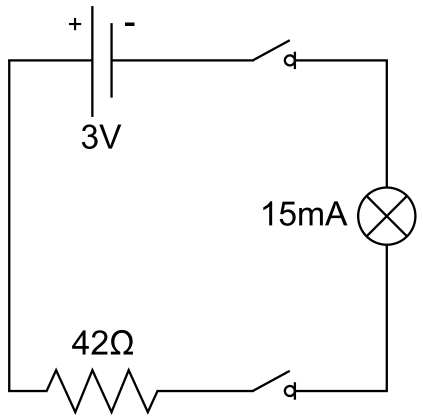
In lieu of spherical markers, we cut retro-reflective tape into circular markers 12mm in diameter and applied them to the shell. We attempted to minimize the markers' impact on contact properties by using small markers in small numbers. When a flat marker is applied to the shell, the marker forms a dome. We found that a dome marker is less visible than a spherical marker, and a given dome is very often occluded to most cameras. To compensate for reduced marker visibility, we posed cameras to maximize overlap between fields-of-view; we also distributed cameras to maximize viewing angles. Given a sufficient number and distribution of cameras and markers, we found that the array can maintain continuous localization of the shell with error that we can correct with minimal assumptions, see Section 5.5.

When we tested our initial motion capture process, we discovered a significant amount of error from the motion tracking model. Our initial array included top down perspective cameras only, which required rotating the WB to add new markers to the tracking model. Using this approach, we were only able to add five markers before VICON detected a symmetric tracking model. With such few markers and using only top-down perspectives, we found (1) that VICON recorded a large number of samples where the orientation of the WB was arbitrarily rotated about two markers and (2) that VICON frequently lost track of the WB entirely due to too few markers being visible. We also found that grasping the WB during registration did not allow us to reliably determine the transform from the registration frame to the model frame; this problem made it extremely difficult to objectively determine the pose of the WB from VICON data. To reduce these sources of error, we incorporated a rigid stand, detailed in Section 5.3, into the registration process, and we added the mid- and low-level camera rings.

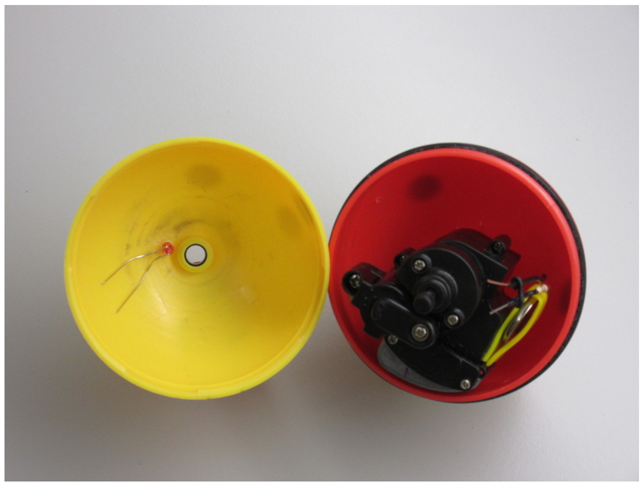
5.2.2 Measuring a Weazelball's Hidden Actuator State

WB behavior is a product of counter-torque generated by the bob/motor assembly, the rotating offset center-of-mass, and contact/impact between shell and environment. The state of the hidden motor assembly drives much of the WB behavior. We considered a number of methods to capture motor assembly state, including adding an IMU, inserting an encoder, and casting a transparent shell; however, we expected each of these alterations to affect dynamic behavior by significantly changing inertia and friction.

We chose to capture motor assembly state by adding a simple, lightweight circuit to the WB that generates a visible light signal each time the motor assembly completes



LED Circuit



Installed Circuit

Figure 20: The WEAZELBALL Signal Circuit

a cycle. We added a high definition (HD) camera to our experimental setup to record *signal events* (LED flashes) generated by the circuit. The *signal circuit* is an independent circuit powered by a 3V button-cell battery. The circuit consists of the button-cell battery, a red LED, a resistor, and a small length of copper wire. We attached the button cell to the motor assembly and rigged the circuit so that leads from the button cell contact the LED leads on each revolution of the motor assembly (and at a consistent position of the motor assembly). The circuit increases the mass of the motor assembly by 5.9% which we consider a minimal change to the motor assembly inertia. Figure 20 shows a diagram of the circuit and an image of the installed circuit.

We mounted the LED by boring a small hole through the free hemisphere at a dimple located near the power button port. We found the light emitted by the LED is directly observable when the shell is viewed from almost any angle where the switch is visible, and the light is indirectly observable due to LED light reflecting from surrounding surfaces.

During each motion capture session, we record video of the WB using the HD camera at 30 frames per second to capture LED signals. We then post-process our data to synchronize each video with VICON pose data by manually examining each frame for a signal event, recording the frame number of such a signal event, and classifying the signal event. We compute the angular velocity of the actuator from the interval between signal events and the video frame rate, and we use the frame of the initial signal in an interval to establish a zero joint angle. From this information, we interpolate the angle and velocity of the actuator over each interval where we detect a signal. Because the motor assembly is constrained to the shell by the joint formed by the actuator, we derive comprehensive state of the WB given state of the shell and state of the actuator.

5.3 Additional Instruments for Weazelball State Measurement

We used a number of additional instruments in our motion capture arrangement. To confine the area in which the WB could roam and consequently the amount of space covered by the VICON system, we constructed an enclosure that we can consistently align and orient to the array. After executing a preliminary motion capture session, we added a number of cameras to the array and a stand to which we mount the WB during model registration. The addition of the cameras and stand are to correct a number of errors associated with the marking method used. We also combined replay visualization of VICON data with HD video to synchronize between the two data streams.

5.3.1 Calibrating the Weazelball to Improve Vicon Tracking

Introducing a stand, *i.e.* a modified camera tripod, to suspend and fix the WB during model registration facilitates finding a non-symmetric tracking model through using an increased number of markers. Suspending the WB during registration allows us to add the low-level camera ring which improves the likelihood of establishing a tracking model that is detectable from any WB orientation. Fixing the WB allows us to determine the transform from model frame to capture frame. Figure 21 illustrates

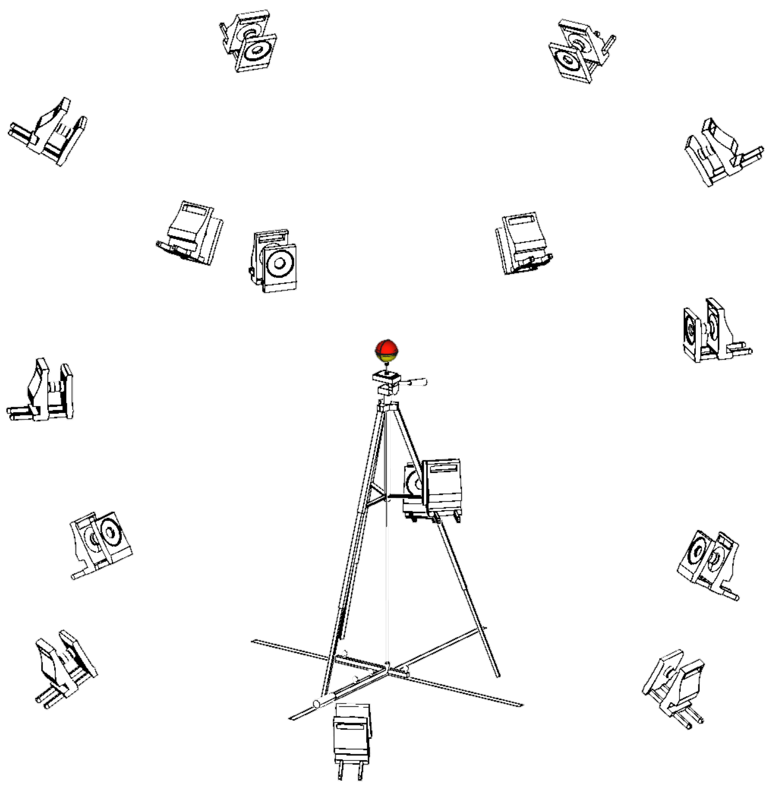


Figure 21: Stand Placement in Registration Array

the positioning of the stand in the registration array. We activate all three camera rings in the registration array. With the additional cameras and the WB mounted to the stand, we are able to apply markers to any point on the shell without manipulating the WB. We find that marker configurations and tracking models developed using this approach lowers tracking error. These adjustments enable us to significantly reduce localization errors compared to our initial approach that did not use the stand-derived marker placement.

The VICON reference frame orientation is nominally inherited by the tracking model when that model is registered. We find that if the WB lies in an arbitrary orientation during registration, the transform from registration frame to capture frame is difficult to determine. By mounting and aligning the WB to the stand, we are able to identify this transformation. We align the stand to the central axis by centering

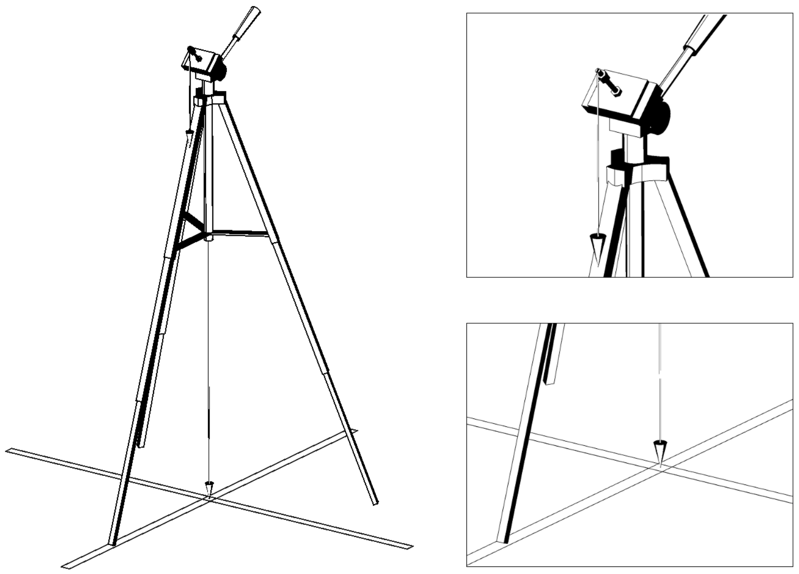


Figure 22: Illustration of the Stand Alignment Procedure

the stand to the central axis origin using a plumb bob and then we rotate the stand until one leg is aligned with an axis. We align the tripod head to the axis-aligned leg by panning the head until the head is plumb with the leg. Once the pan degree-of-freedom is aligned, we level the head with tilt and leg length adjustments. Figure 22 illustrates alignment of the stand with the central axis.

We align the WB to the stand using features that we added to the tripod and to the WB. To mount the WB, we attach a clamp with a notch to the tripod head. We align the notch to the axis-aligned stand leg by tilting the tripod head, suspending a plumb bob through the notch, and then panning the tripod head until the notch is plumb with the center of the leg. We level the tripod head and confirm the head is level in both axes. If the head is not level, we adjust the height of the legs until the head is level. We mount the WB to the stand mounting plate using the hardware and procedure detailed in Figure 23. We align the WB to the clamp using a pen mark A drawn in an arc across the free hemisphere from the center C of the WB port to the shell equator. Mark A is drawn across the equator onto the fixed hemisphere, and we label the mark on the fixed hemisphere A'. We use A to align the shell to the stand

and we use both A and A' together to realign the shell hemispheres to each other whenever we open the WB for servicing. When mounting the WB, we rotate the WB until A aligns with the notch. Figure 24 illustrates the registration frame origin and the alignment of A to the stand. We choose to mark A to align with the joint angle where the motor assembly closes the signal circuit so that we can assign that joint position as the zero position.

5.3.2 Enclosing the Weazelball During Motion Tracking

We designed a square, inset enclosure (similar in appearance to a table) illustrated in Figure 25 to limit the area needed for camera coverage and to ensure the WB remains as observable as possible during motion capture. The enclosure is approximately 4ft on a side and 2in "deep". The 4ft surface is large enough for the WB to reach maximum velocity and exhibit characteristic behaviors, and the enclosure is also small enough that the cameras' fields-of-view overlap (as necessary for motion capture). The enclosure depth is greater than the WB radius, *i.e.* 41mm, to prevent the WB from leaving the enclosure, but the enclosure depth is also shallow enough so that camera placement is not limited to top down perspectives. Due to our marking method, we found that a number of cameras must overlap fields-of-view from many directions. We also found that several cameras must be positioned low enough to track markers close to the enclosure surface, which motivated us to add the mid-height camera ring.

During construction, we cut small notches on top of and at the center of each rail that frames the enclosure. These *center notches* serve two purposes: they enable the enclosure to be aligned with the central axis and they enable the center of the enclosure to be set as the capture frame origin.

We set up the enclosure using the following steps. First, we placed a small, commercially produced table over the central axis (henceforth denoted the "base").

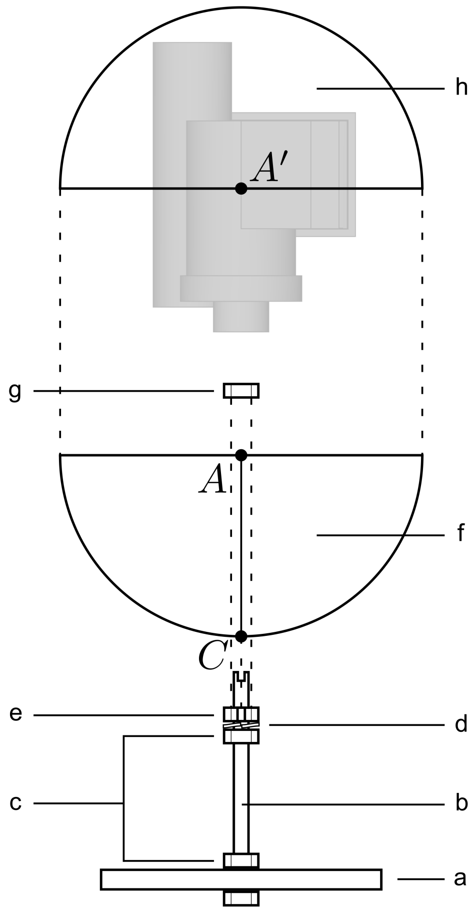
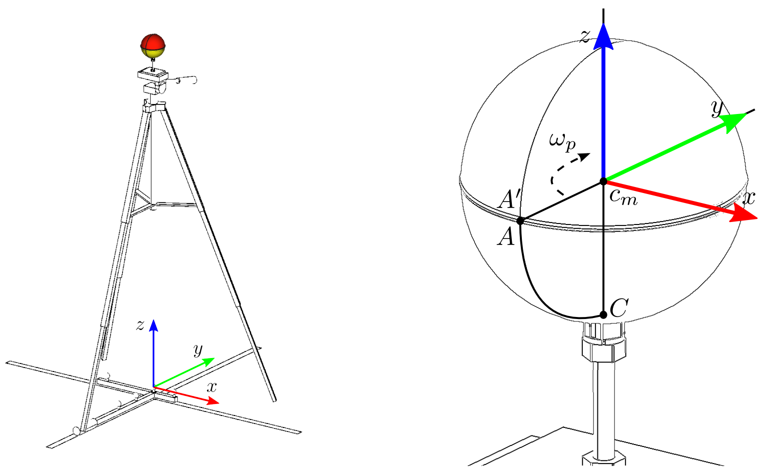


Figure 23: Front View of the Weazelball Mounting Process

We mount the Wb to the tripod such that A and A' are colocated and $\overline{AC}$ aligns to the aligned tripod leg which aligns $\overline{AC}$ to the registration frame. We modified a standard tripod mounting plate (a) to accept a 2-1/2in. bolt (b) and we cut a 2mm notch into the threaded end of the bolt. We thread two retaining nuts (c), a locking washer (d), and a third nut with a centered notch (e) onto the bolt. We align (e) with the notch in the bolt and align the notch in the bolt such that the notch is perpendicular to the front of the plate. The notch in the bolt accepts the plumb line for aligning the head of the stand with the aligned leg. We then tighten retention nuts (c) to prevent rotation of (b) and (e). Once we align the stand to the registration array, we remove the mounting plate from the stand, mount the Wb to the mounting plate, and return the mounting plate to the stand. To mount the Wb, we unthread the shell, place the free hemisphere (f) onto bolt (b) by inserting the bolt into the port such that (f) rests on (e), clamp the free hemisphere to the bolt with nut (g), remove the plastic cover on the motor assembly that contains the plastic nipple used to actuate the switch, thread the fixed hemisphere (h) onto (f) aligning A' with A, and align $\overline{AC}$ with the notch in the face of (e)



(a)   (b)

Figure 24: Illustration of the Tracking Model Reference Frame

(a) We assign the registration frame origin to the central axis and, once the stand is aligned, we mount the Wb to the stand. (b) The model frame origin position is manually assigned but the orientation is inherited from the registration frame origin. By aligning mark A to the stand, we are able to transform a fixed point on the shell to any pose recorded by motion capture

We then placed the enclosure on top of the base. Next, we aligned the enclosure to the central axis. Finally, we leveled the enclosure by inserting shims between base and enclosure until the enclosure is level at the center and in all four quadrants.

We aligned the enclosure to the central axis by suspending plumb bobs from the center notches and then adjusting the enclosure's pose until it is plumb with two axes of the central axis. We assigned the capture frame origin to the enclosure center by suspending mason's line between center notches and then aligning the wand with the intersection of the line. Figure 26 illustrates alignment of the enclosure with the central axis and positioning of the wand.

5.3.3 Visualizing Weazelball State Data in Gazebo

We used Gazebo to visually review streamed Vicon state for two purposes: (1) to reduce error in data collection, and (2) to synchronize between Vicon data and HD video data. We used the Solidworks COLLADA Wb models as a visual model.

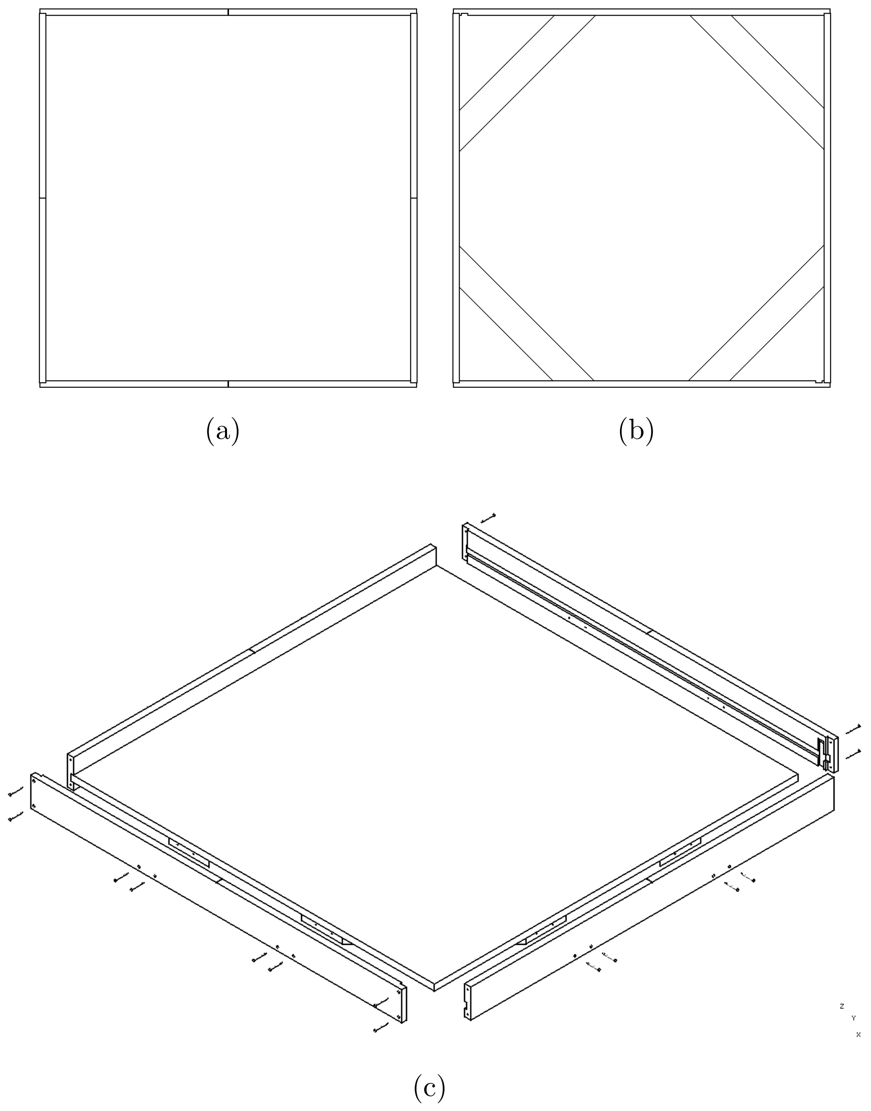


Figure 25: Enclosure Model

Enclosure model: (a) top view with visible center notches for alignment; (b) bottom view with angle braces used to square and reinforce the enclosure corners; (c) exploded view shows enclosure construction

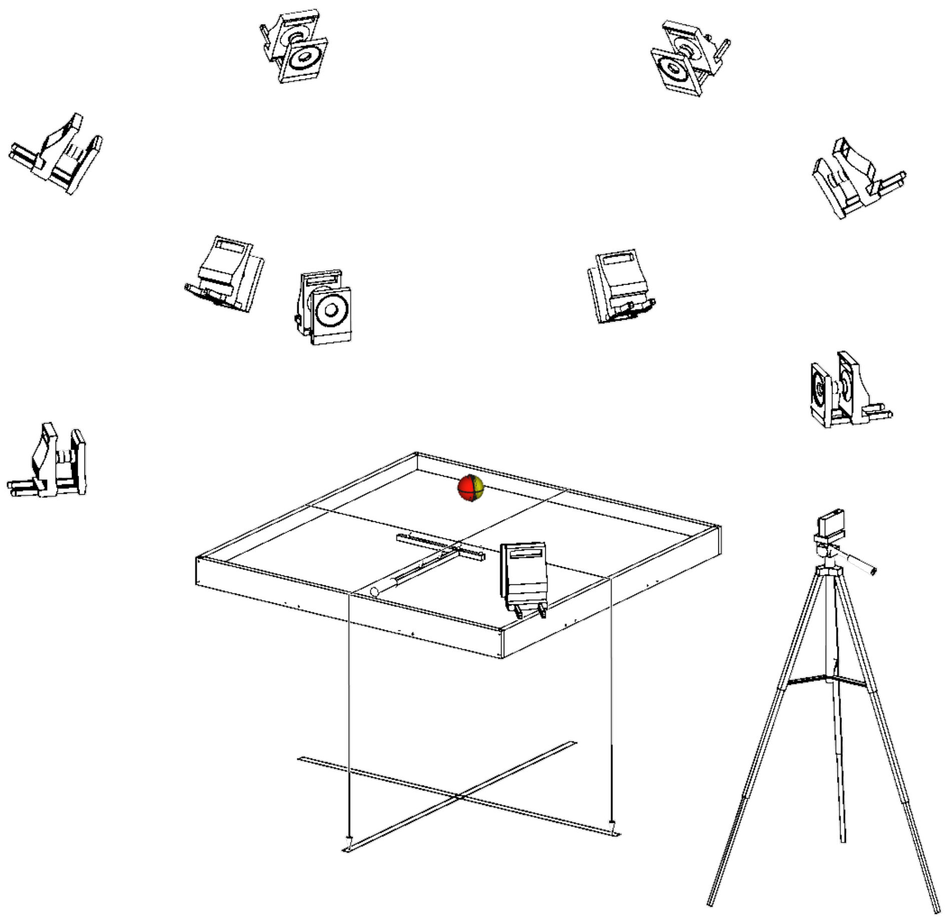


Figure 26: Illustration of the Motion Capture Configuration

We activate the high and mid camera rings in the motion capture array. We deactivate the low camera ring because the cameras are occluded by the enclosure. We aligned the enclosure by using plumb lines suspended from the center notches. We centered the wand to the enclosure using mason's line suspended between center notches and then assigned the array origin. We set up a HD video camera with a clear view of the enclosure to record signal events from the motor assembly

We used a Gazebo plugin to read Vicon state and to update the Wb model toward replaying captured data. We have found that replay visualization allows us to assess the quality of Vicon camera and Wb marker configurations. Through such replay, we found that the configurations of the array and markers have significant impact on the amount of error introduced into the data generated by the motion tracking system.

5.4 Capturing Weazelball State

We found that our motion tracking method requires careful planning and testing of both the Vicon array and marker arrangement to capture accurate data. Through an initial data collection experiment, we noted that casual arrangements of either cameras or tracking markers result in obvious, unpredictable, and difficult to correct errors in capture data. These errors seem to result from having too few markers visible to too few cameras. To minimize such errors, we focus a large number of cameras into the experimental space while varying the camera heights to increase marker detectability, and we use a systematic approach to model registration. We also found that by testing Wb motion capture and by visualizing the test data, we were able to tune camera and marker placement to reduce errors in the collected session data. We executed motion tracking using the Vicon Tracker application which is designed for tracking rigid bodies. For each sample taken by Vicon, Tracker maps the pattern of visible markers to a tracking model registered with Tracker, computes the pose of the model, and streams data for recording by another process.

5.4.1 Registering the Weazelball Shell with Vicon

Our registration process uses the following steps. We align the wand to the central axis and then assign the array origin frame to the wand[3]. We replace the wand with the stand and align the stand to the central axis. We mount and align the Wb to the

[3]Our registration frame was rotated 180° around the z-axis with respect to our capture frame.

stand. We apply a marker to the shell and register the marker to the current tracking model through Tracker, which the software then validates. Tracker may report that the new marker results in a symmetric tracking model. If a symmetric model is detected, we remove the marker, delete the current tracking model, and register all markers previously added (we must delete and re-register the tracking model because a previous marker arrangement may now be detected as symmetric, thereby forcing us to remove markers until Vicon accepts a valid marker configuration). Once a valid marker configuration has been established, we repeat the marker addition process. We were able to add a total of eight markers to the Wb using the process, and the resulting tracking model was tracked unambiguously for nearly all samples captured.

5.4.2 Capturing Weazelball Shell State Using Vicon

Our motion capture process involves switching from the registration array to the capture array, aligning the enclosure to the central axis, assigning the capture array origin, aiming the HD video camera, and streaming data from the capture array. Before gathering session data, we gather and visualize test data. If the test data exhibits obvious visual artifacts, we repeat model registration to improve the tracking model. Before the Wb is brought into the enclosure, we cover the Wb with an opaque bag. Vicon streams data whenever a tracking model is detected, and using the bag reduces occurrences of premature detection.

We executed session collection using the following procedure: we place the Wb into the bag, we start recording Vicon, we start recording HD video, we power on the Wb, we place the bag into the enclosure, we release the Wb from the bag (and remove the bag from the enclosure), and we record data for the allotted time. Once the session time has elapsed, we stop recording both Vicon and video. We carried out a total of ten collection sessions to develop initial sample data. Each session lasted approximately two minutes and was recorded using a 100Hz Vicon array and a 30Hz HD video camera.

5.4.3 Video Recording Hidden Weazelball Actuator State

We wish to capture as many signal events from the LED circuit as possible. To ensure that the earliest signal events are visible to the HD camera, we release the WB at a pose determined relative to the camera's pose. We place the HD camera on the side of the enclosure and center the enclosure's center in the camera frame. When we introduce the WB into the enclosure, we orient the WB so that the free hemisphere faces the HD camera. We found that the WB is very likely to continue moving across the enclosure with the LED directly visible to the HD camera (at least until the WB collides with a wall) if the WB is given a small initial forward velocity when released. The WB release pose is illustrated in Figure 27.

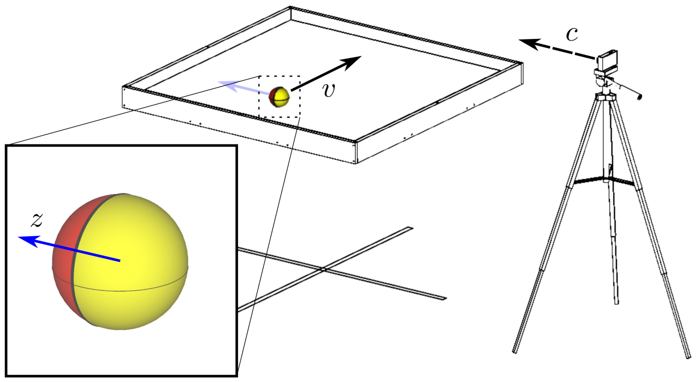


Figure 27: Releasing the WEAZELBALL

When we release the WB, we consider the positioning of the enclosure with respect to the camera. The WB will likely roll in the general direction of v if we release the WB with some initial forward velocity in the direction of v as long as the WB model frame's z axis is roughly aligned with the camera view vector c

Each video frame contains a WB reference pose and may contain evidence of an LED signal event. We used the open source software tool `melt` to review each video frame. `melt` supports frame-stepping video replay in our HD camera's native codec, thus permitting us to analyze HD video frame-by-frame. Using such frame-by-frame analysis, we detected and classified LED signal events, and we cross-referenced video poses with visualized VICON poses. Through these processes, we were able to synchronize data and to validate VICON measurements.

5.5 Estimating Weazelball State

After collecting session data, we process raw VICON and video data to identify marker detection error, to synchronize the data streams, to classify signal events, and to estimate actuator state from video data.

5.5.1 Synchronizing Vicon Data and Video Data

Because our process generates both VICON data and video data, it is necessary to synchronize the two independent data sets to produce a single, unified data set. We synchronize by reviewing video data until we identify a collision between the WB and at least one of the enclosure rails that results in the WB coming to a stop. We then step the motion capture visualization to the same collision and compare the video reference pose with the visualized pose to synchronize the data. We compare the visualized pose with the video reference pose and if they are the same we use the current VICON time and the current video frame number to compute a virtual time basis shared by both data sets. Once the video and visualization are synchronized at this collision, we record the frame number of the video and the time of the capture visualization. We then backtrack to the earliest VICON measurement where we can identify no errors and use that state as the start of virtual time. We compute the virtual time of all VICON and video frames using the start of virtual time and the sampling rate of the VICON system and the frame rate of the video camera. For each motion capture session, we record the following synchronization parameters: the video frame number of the first collision that could be synchronized with VICON data, the VICON time of the same collision in the VICON samples, the total number of frames in the video, the frame rate of the video, the sampling rate of VICON, the start of virtual time in the VICON data, and the video frame that matches the start of virtual time. Figure 28 shows a synchronized frame from both visualization and video replay.

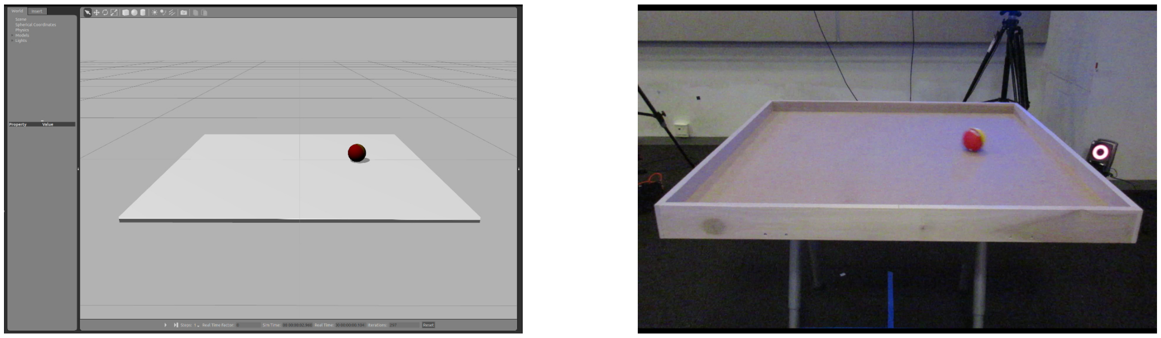


Figure 28: Synchronized WEAZELBALL Data

Two frames from a synchronized visualization show WB state at a given time during motion capture.

We found that differences between the recording rates of the VICON system and the HD camera used in the experiments makes perfect synchronization infeasible. We also found that the video camera sensor type can produce artifacts in the video data such as the afterimage of a signal appearing in the subsequent frame or the smearing of moving objects in a single frame (due to scan latency). Though our data appears cogent by visual inspection, future capture efforts should anticipate and compensate for these issues.

5.5.2 Qualitative Assessment of Weazelball Behavior and Actuator Data

We processed each video stream frame-by-frame to catalog each identifiable signal event recorded by the HD camera. We define an *identifiable signal* as either a direct observation of the lit LED or as an indirect observation of LED light reflected from the enclosure. We found a number of signal events were not identifiable due to occlusion of the LED to the HD video camera and the LED being oriented such that LED light is not noticably reflected by the enclosure; however, given an approximate frequency of the motor assembly we infer that a signal should have been observed. We denote such missing signals as a "gap". Gaps may consist of the loss of one or more signals. We chose not to attempt to infer the number of signals lost in gaps because we discovered that motor assembly angular velocity varies with WB forward velocity, which would require more sophisticated modeling than our initial, constant-angular-velocity assumption can provide. We also found that signals may span two video frames with one of the two signals appearing weak; such cases suggest that the signal duration is approximately equal to the frame rate of the camera plus some small fraction of the frame rate. Table 2 summarizes the signals captured and classified for all sessions.

Table 2: HD video LED signal detection

Session	Identifiable Signals	Indirect Signals	Gaps	Two-Frame Signals	Longest Observation (s)
1	249	73	31	39	8.9
2	242	64	32	29	7.567
3	215	37	38	22	4.433
4	232	46	29	18	7.8
5	225	50	34	20	10.33
6	207	35	27	22	8.367
7	265	68	41	46	6.533
8	271	101	37	54	9.567
9	249	58	33	31	6.2
10	265	82	37	39	9.567

Table 2 summarizes our classification of the signal data gathered during session collection. *Identifiable signals* are the number of discrete signals identified on video. *Indirect signals* are those identified through specular reflection. *Gaps* quantifies the number of times that at least one signal should have been observed but no signal was detected. *Two-frame signals* refers to the times that a single signal spanned consecutive video frames. *Longest observation* is the longest period of time where signals are continuously observed

5.5.3 Identifying Error in Weazelball State Data

Through visualization, we identified two anomalous behaviors in the raw Vicon data as error indicators: (1) from one sample to another, the visualized Wb spontaneously jumps from the enclosure surface and significantly changes orientation without any external force applied and (2) the visualized Wb oscillates between interpenetrating and floating above the enclosure surface. We refer to (1) as *snap changes* and attribute these events to marker occlusion. We are able to significantly reduce the frequency of snap changes by optimizing the position and number of markers and cameras. We were unable to eliminate all snap changes through hardware optimization, but we found that if these orientation errors occur with low incidence, they are correctable through interpolation. We attribute (2) to a combination of errors: error between the tracking model center and the physical center of the Wb (arising from manual center assignment) and error from using dome markers. We are able to mitigate the error in the center position by estimating the center of the tracking model. Figure 29 illustrates error in the data gathered from Session 1. We focused our analysis on the vertical component z of the Wb's geometric center, which should be equal to the Wb radius (ideally $z = 41$mm) and on change in quaternion orientation $\Delta\theta_i = ||\theta_i - \theta_{i-1}||$ between the current frame and the subsequent frame. Figures 29(a) and 29(c) show that there are infrequent spikes in both z and $\Delta\theta_i$ in corresponding samples, there is error between measured mean z and radius, and there is noise in all measurements.

5.5.4 Correcting Weazelball Orientation Error in Vicon Data

We attribute spontaneous snap change error events to significant (multiple marker) occlusion. We found that snap orientation change incidence is subject to camera and marker placement. We found that snap changes are infrequent for well-positioned camera arrays, are infrequent for a sufficient number and arrangement of markers,

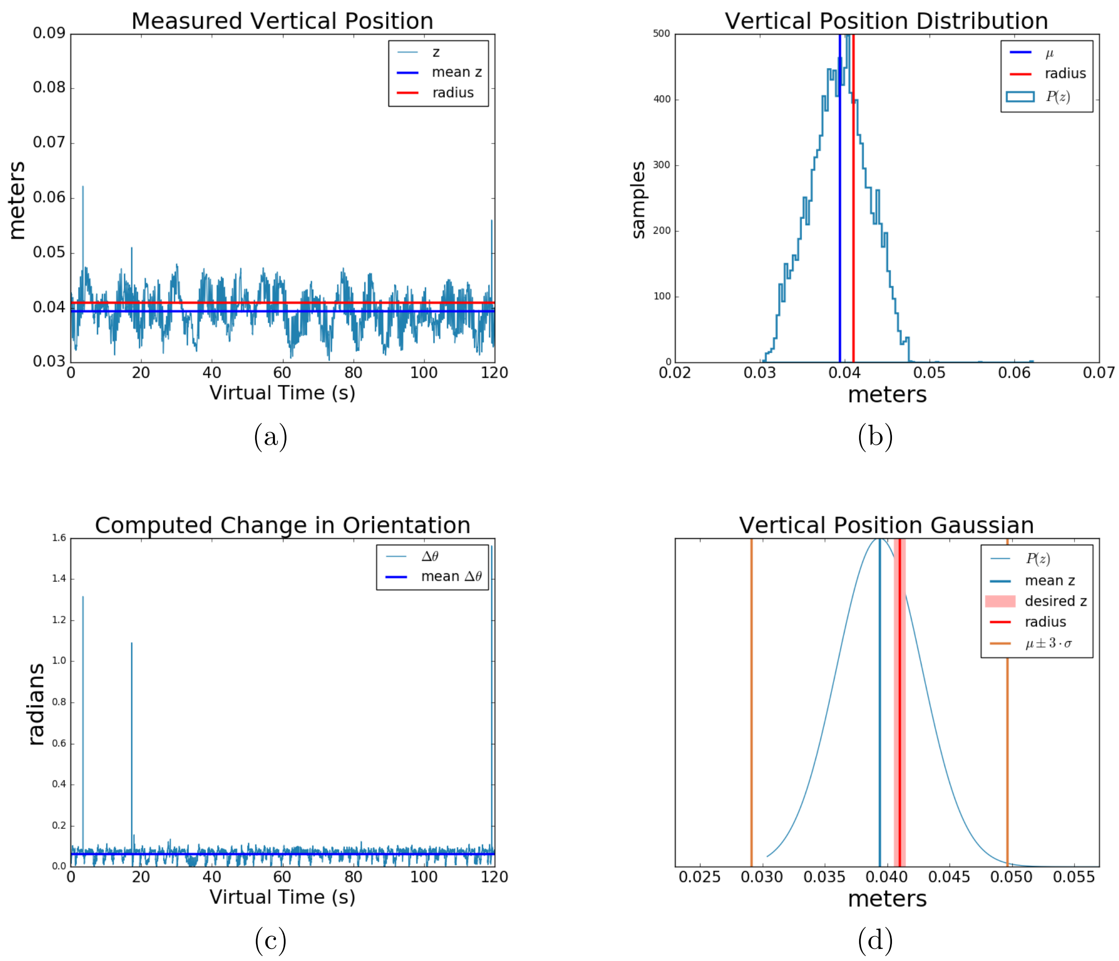


Figure 29: Error Analysis of Weazelball Motion Capture Data

These plots reflect data gathered during our first session and are representative of data gathered across all sessions. (a) Shows the vertical z component of each Vicon sample. Because the Wb is spherical and never leaves the table's surface, z should equal the Wb radius; however, the error between mean and radius is pronounced and the signal exhibits both infrequent spikes and substantial noise. The sign of the error between mean and radius implies that the situated Wb interpentetrates into the table surface which is physically impossible, and the spikes suggest that the Wb occasionally jumps from the surface which is contradicted by video data. (b) A histogram of the vertical z component suggests the samples have a Gaussian distribution. (c) A plot of the orientation change between neighboring samples shows noise in orientation measurements and shares spikes in the same samples with the vertical measurements. Spikes in orientation change suggests that the Wb makes snap changes in orientation which is contradicted by video data. (d) The Gaussian defined by measured vertical position does not fit the ideal. We expect some error in measurements due to sensor noise, but highly accurate measurements should result in samples tightly clustered around the radius. Characteristics of the vertical position Gaussian, *i.e.* the wide bell, error between mean and radius, and the presence of a number of outliers, suggest error in the raw signal data is a product of several errors in the measurement system and occur primarily when the WB enters a corner. We also found that it is feasible to use interpolation to correct snap change events if their incidence is low. We use large $\Delta\theta_i$ to identify snap change events.

Our process to identify snap change events in a given session is as follows. We first compute $\Delta\theta_i$ between each VICON sample and the subsequent sample. We derive the mean μ and standard deviation σ from the distribution of $\Delta\theta_i$. We scan the set of $\Delta\theta$ in order of samples and, if $\Delta\theta_i$ is greater than $\mu + 3\sigma$, we record the index of that sample as an outlying $\Delta\theta_i$. Each snap change event consists of at least two outlying $\Delta\theta$'s where the first in the sequence indicates VICON has lost track of enough markers to fit the correct pose to the WB and the second in the sequence indicates that enough markers became visible for VICON to resume correct tracking. Typically each snap change event consists of two outlying transitions; however, we did identify one instance where VICON recorded three transitions in the snap change event, *i.e.* a sequence of two rapidly changing orientations but incorrect poses followed by a third transition back to correct localization. Once we have identified samples containing outliers, we step visualization to the sample and evaluate whether the WB has undergone a change in orientation that exceeds typical physical performance. If the change in orientation is unrealistic, we record that sample as the beginning of the snap change event, and then we step through the subsequent samples until we identify the sample where the WB returns to an orientation consistent with physical behavior. We then interpolate between the start outlying sample up to the end outlying sample. Table 3 summarizes the number of snap change events that we identified across all of the session data that we collected.

5.5.5 Correcting Weazelball Position Error in Vicon Data

We attribute, at least partially, the oscillation anomaly and the offset between mean and WB radius demonstrated in Figure 29(a) to manual center assignment in the tracking model. We found that manual assignment results in an unknown offset

Table 3: Frequency of VICON orientation snap change error events

Session	Samples	Mean $\Delta\theta$	σ	3σ	Error Events	Mean Error $\Delta\theta$	Samples Affected
1	12416	6.38	3.57	10.72	6	127.93	4
2	12464	6.38	2.86	8.58	6	68.10	3
3	12227	6.39	4.94	14.82	14	131.48	32[1]
4	12377	6.32	4.35	13.03	8	148.01	5
5	12490	6.26	3.83	11.49	10[2]	101.56	7
6	12311	6.29	4.29	12.88	8	147.56	8
7	12324	6.94	5.59	16.78	18	131.54	13
8	12590	6.60	4.43	13.29	8	146.05	8
9	12433	6.72	5.99	17.97	18	147.59	18
10	12469	6.72	4.68	14.04	10	144.43	6

Table 3 summarizes the number of (erroneous) orientation snap changes that we identified in our session data. $\Delta\theta_i$ is the computed change in orientation between two VICON samples and *mean* $\Delta\theta$ is the average change in orientation over all samples for a given session. A generally decreasing mean $\Delta\theta_i$ between Sessions 1-6 and Sessions 7-10 reflects decreasing battery charge, and the sudden increase in mean $\Delta\theta$ between Sessions 6 and 7 reflects a battery change between these two sessions. For each session, σ is the standard deviation in the distribution of $\Delta\theta$ and 3σ represents the $\Delta\theta$ distribution's 99% confidence interval. We assume that orientation changes that substantially exceed 3σ are orientation snap changes. For each session, *error events* quantifies the number of orientation snap changes, *mean error* $\Delta\theta$ quantifies error event average $\Delta\theta$, and *samples affected* quantifies the number of samples that require correction via interpolation
1 — The bulk of the samples affected in Session 3 occur when the WB comes to a halt in a corner in such a way that most markers are occluded. 22 samples occur where WB orientation is effectively unchanging
2 — One sample in Session 5 exceeded the 3σ criteria but only by a small margin, *i.e.* $\Delta\theta_i = 19.543264$, leaving an odd number of events which is explained by another anomaly at samples 33, 34, and 35 where the WB first flipped one way, then another, and then back to correct orientation

between the tracking model and the subject, which requires estimation of center offset and compensation in Vicon data.

Vicon captures a pose by fitting the tracking model to the physical robot. The tracking model is defined in the model frame, so the pose recorded by Vicon represents a transform from the model frame to the capture frame. If the center of the physical robot is not colocated with the model frame center, the position recorded by Vicon will contain an offset error that is oriented with respect to the model orientation in the capture frame. We establish a clear rotational map between physical robot and tracking model by aligning the Wb using the stand; however, manual center assignment establishes an arbitrary point as the "center" of the model frame, and this center is the translational basis for transformations between model frame and capture frame. There is an unknown offset between the arbitrary model frame center and the physical robot center which must be computed.

We use $\boldsymbol{c}_m$ to represent the manually assigned center, $\boldsymbol{c}_0$ to represent the true physical center of the Wb, and $\boldsymbol{u}$ to represent the offset between the two centers, all defined in the model frame. We define $\boldsymbol{u}$ by the formula:

$$\boldsymbol{u} = \boldsymbol{c}_0 - \boldsymbol{c}_m \tag{4}$$

We use V to represent the set of all Vicon samples, $\boldsymbol{p}$ to represent a pose of the Wb in the capture frame where $\boldsymbol{p} \in V$, $\boldsymbol{x}$ to represent the translational transform of $\boldsymbol{p}$, and $\mathbf{R}$ to represent the rotational transform defined by $\boldsymbol{p}$. We also use $\hat{\boldsymbol{\imath}}$, $\hat{\boldsymbol{\jmath}}$, and $\hat{\boldsymbol{k}}$ to represent unit vectors directed along the x-, y-, and z-axes of the global coordinate frame.

We compute $\boldsymbol{u}$ to correct displacement error in all Vicon samples using two methods (for cross-checking): (1) compute the closest point to all marker points recorded in model registration, (2) minimize the Wb height error across all Vicon samples. We found that these two approaches, described in detail below, produced nearly identical solutions.

Iterative center estimation

We found that VICON tracking of dome markers is less accurate than tracking of traditional spherical markers. Due to error in dome marker tracking, the positions of dome markers assigned to the tracking model are estimates, which means that marker positions recorded in the VICON skeleton file (**vsk**) data contain error. The **vsk** defines model space and the relative positions in model space of all marker centroids registered to the model. Regardless, we used **vsk** data to estimate $\boldsymbol{u}$.

By definition, any point on the surface of a sphere is equidistant to the center of the sphere. The WB shell appears spherical, so markers on the shell should be equidistant to the center of the WB. The manually assigned center $\boldsymbol{c}_m$ defines the position of the model frame origin, so the marker positions stored in the **vsk** are relative to $\boldsymbol{c}_m$. We verified that $\boldsymbol{c}_m$ is not equidistant from all marker positions, which supports our hypothesis that a manually designated center contributes error to captured poses.

We estimated $\boldsymbol{c}_0$ as $\boldsymbol{c}_m$ and then repeatedly perturbed the estimated center until it was approximately equidistant from all marker positions. Due to the inaccuracy of marker measurements and the number of markers, we believe that an analytic solution is infeasible and an iterative solution is practical.

Our iterative process consists of (1) assigning $\boldsymbol{u} = \boldsymbol{c}_m$ and storing both the distances from $\boldsymbol{u}$ to all markers and the sum of all distances; (2) perturbing $\boldsymbol{u}$; and (3) storing the updated distance information retaining the new $\boldsymbol{u}$ (only if both the sum of distances decreases and the distance to every marker decreases). Using this approach and after 10^7 iterations, we computed:

$$\boldsymbol{u} \approx [-1.067, 3.499, 2.313] \times 10^{-3} m \tag{5}$$

Height error minimization

In order to validate the offset we computed using the iterative center estimate method, we computed the offset using a second approach as well. We know that the WB is a rigid ball of known radius, and we confirmed through review of video of all sessions that the WB never leaves contact with the enclosure surface. Based on this information, we formulated a least squares optimization approach to minimize the error in the capture frame's vertical dimension (z-axis) for all VICON samples.

Because the WB maintains contact with the enclosure surface and the WB is a rigid sphere, for all WB poses in V, the actual vertical position z of the geometric center of the WB must equal the WB radius r. We assume that the offset u between true physical center and manually assigned center in the model frame accounts for the error between z and r. Given perfect pose measurements, for all $\boldsymbol{p} \in V$ the following holds true:

$$(\boldsymbol{x} + \mathbf{R}\boldsymbol{u})^{\mathsf{T}} \hat{\boldsymbol{k}} = r \tag{6}$$

We find $\boldsymbol{u}$ by minimizing the error, ϵ, across all samples in V:

$$\epsilon = \frac{1}{2} \sum_{i=1}^{n} \left((\boldsymbol{x} + \mathbf{R}\boldsymbol{u})^{\mathsf{T}} \hat{\boldsymbol{k}} - r \right)^2 \tag{7}$$

Using this approach with a data set consisting of all samples from all sessions, we computed:

$$\boldsymbol{u} \approx [-1.267, 3.365, 1.907] \times 10^{-3} m \tag{8}$$

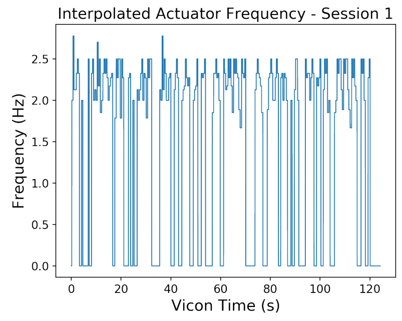
We believe that the minimized height error method produces a more accurate estimate than the iterative center estimate method. The minimization method derived a solution from more than 120,000 samples while the iterative method derived a solution from a total of 8 samples. We are satisfied that the two methods produce the same answer to one significant digit given the noise in the VICON data and the difference between the number of samples used to arrive at each result, but we believe the minimization method's solution is more accurate because it is derived from several orders of magnitude more samples. We therefore use the $\boldsymbol{u}$ computed by the minimization method to correct all samples.

5.5.6 Estimating Weazelball Actuator State

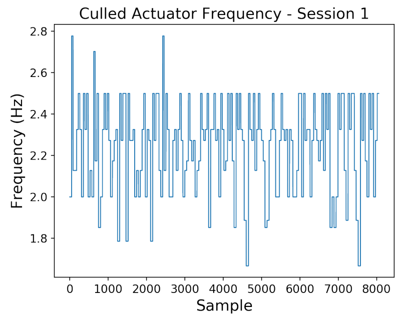
The Wb motor is a low cost, unregulated DC motor that operates at variable velocity depending on the state of the Wb. Interactions between motor, battery, shell, gearing, and environment result in variable motor performance and motor assembly rotational velocity which complicates development of a detailed motor model. When the shell collides with the edges of the enclosure, the inertia of the motor assembly may contribute an impulse that causes the motor assembly to increase rotational velocity for a short duration, and when the shell comes to a stop, the motor assembly rotational velocity will decrease due to friction between the shell and surface plane. When the Wb has sufficient forward momentum and is rolling freely across the center of the enclosure, the motor assembly rotational velocity will increase until the Wb reaches maximum forward velocity.

For modeling, we assume that the motor operates at constant velocity; therefore, the motor assembly has constant rotational velocity and the actuator has constant rotational frequency. We provide the actuator rotational frequency as a parameter to control the simulated actuator. While the real actuator has variable frequency, we can assume that once the actuator has reached maximum frequency, the motor assembly rotational velocity remains constant. Constant rotational velocity allows us to interpolate actuator state – position and velocity – for Vicon samples where the actuator is operating at maximum frequency.

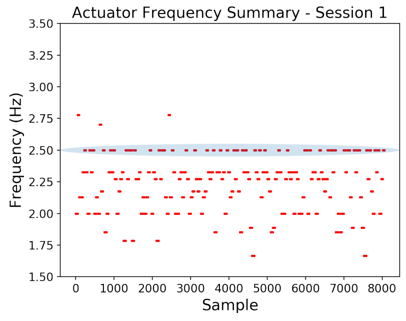
We estimated the actuator rotational frequency from video analysis as 2.333Hz; however, we carried out a more in depth assessment of the signal data and determined that 2.5Hz is a more reliable estimate for sessions 1 through 6, refer to Figure 30.



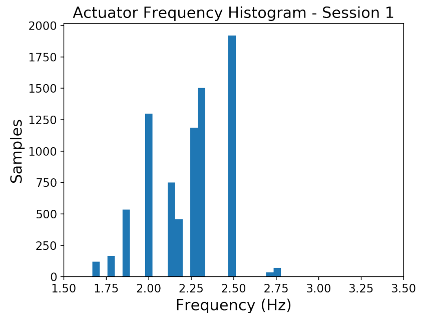
(a)



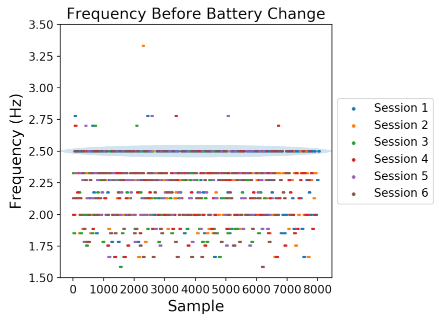
(b)



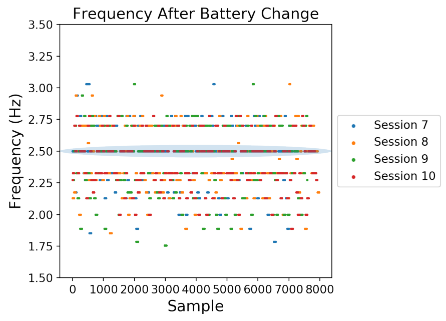
(c)



(d)



(e)



(f)

Figure 30: Interpolated Weazelball Actuator Frequency

(a) Actuator frequency was derived from video data; however, many samples are unobservable due to gaps in observation. Non-zero values are based on observations and zero values are the result of non-observations. (b) We culled samples where the actuator frequency is zero from further consideration. (c) The samples are not necessarily adjacent, so a scatter plot more appropriately represents actuator freqency for observed samples. An ellipse in this scatter plot and subsequent scatter plots highlights the 2.5Hz actuator frequency. (d) A histogram of the session 1 samples shows a small number of outliers greater than the maximum frequency which can be attributed to impulses generated by collisions with the enclosure. (e) For the sessions before the battery change, the maximum frequency is consistent. (f) After the battery change, actuator frequency is inconsistent with the actuator frequency preceding the battery change.

Because the signal circuit cannot always be observed when signals are generated due to gaps, there are a significant number of VICON samples where actuator frequency is indeterminate. Figure 30(a) shows interpolated actuator frequency from the first VICON session between visible signals where zero values indicate indeterminate frequency. We excluded from further investigation any samples with indeterminate frequency and illustrate the remaining samples in Figure 30(b). If we exclude outliers beyond the apparent maximum actuator frequency, we measure the maximum frequency to be 2.5Hz, see Figures 30(c) and (d). The measured maximum actuator frequency is consistent in all sessions preceding the battery change, see Figure 30(e); however, following the battery change, the actuator frequency had higher variability and the maximum frequency is not consistent with the maximum frequency exhibited in sessions preceding the battery change, see Figure 30(f). Because the voltage supplied to the motor is unregulated, the battery change alters motor performance until the battery has discharged somewhat.

5.6 Validating Weazelball Simulation Through Support Vector Regression

.

Figure 31 plots the trajectory of a situated WB in the xy-plane and shows that a situated WB tends to roll forward when unobstructed and may undergo wild direction changes upon collision with an obstacle. We simulated the WB with realistic inertias, realistic coefficients of friction, and a realistic motor velocity. Figures 32 and 33 illustrate that small variations in a model can significantly alter behavior, implying that the model must be carefully constructed to obtain accuracy. Our goal in modeling and simulating the WB has been to investigate divergences between simulated and real behaviors in state-of-the-art robotics simulators when using reasonably accurate system parameters.

Validation of a WB simulation poses two problems: (1) the state space is multi-

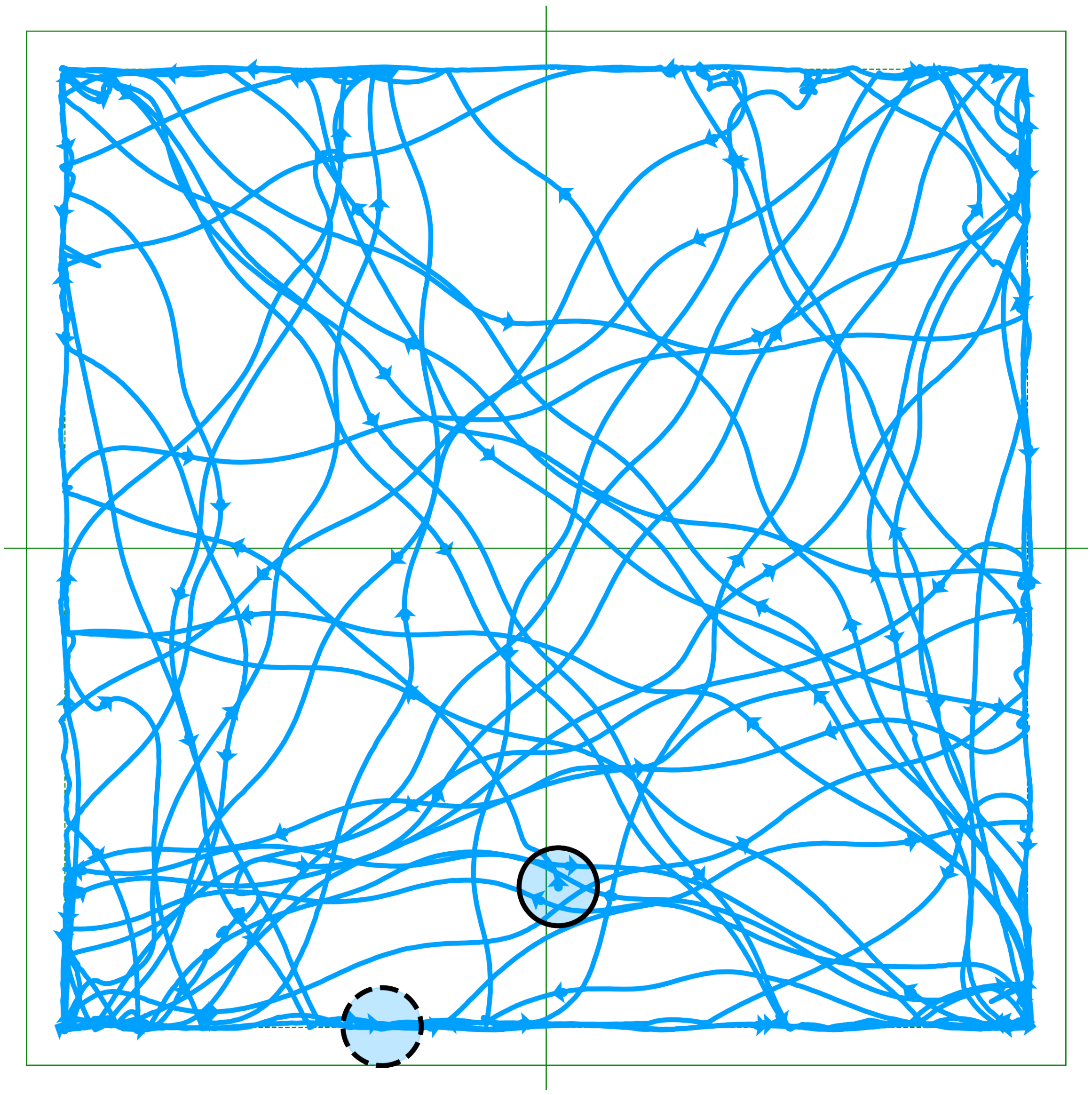


Figure 31: Trajectory from a Situated WEAZELBALL

120s of data collected from a real WEAZELBALL shows the robot tends to oscillate around a relatively straight trajectory when rolling unobstructed and may change directions wildly after colliding with an obstacle. Arrows indicate direction at 1s time intervals, a solid circle indicates starting position, and a dashed circle indicates ending position.

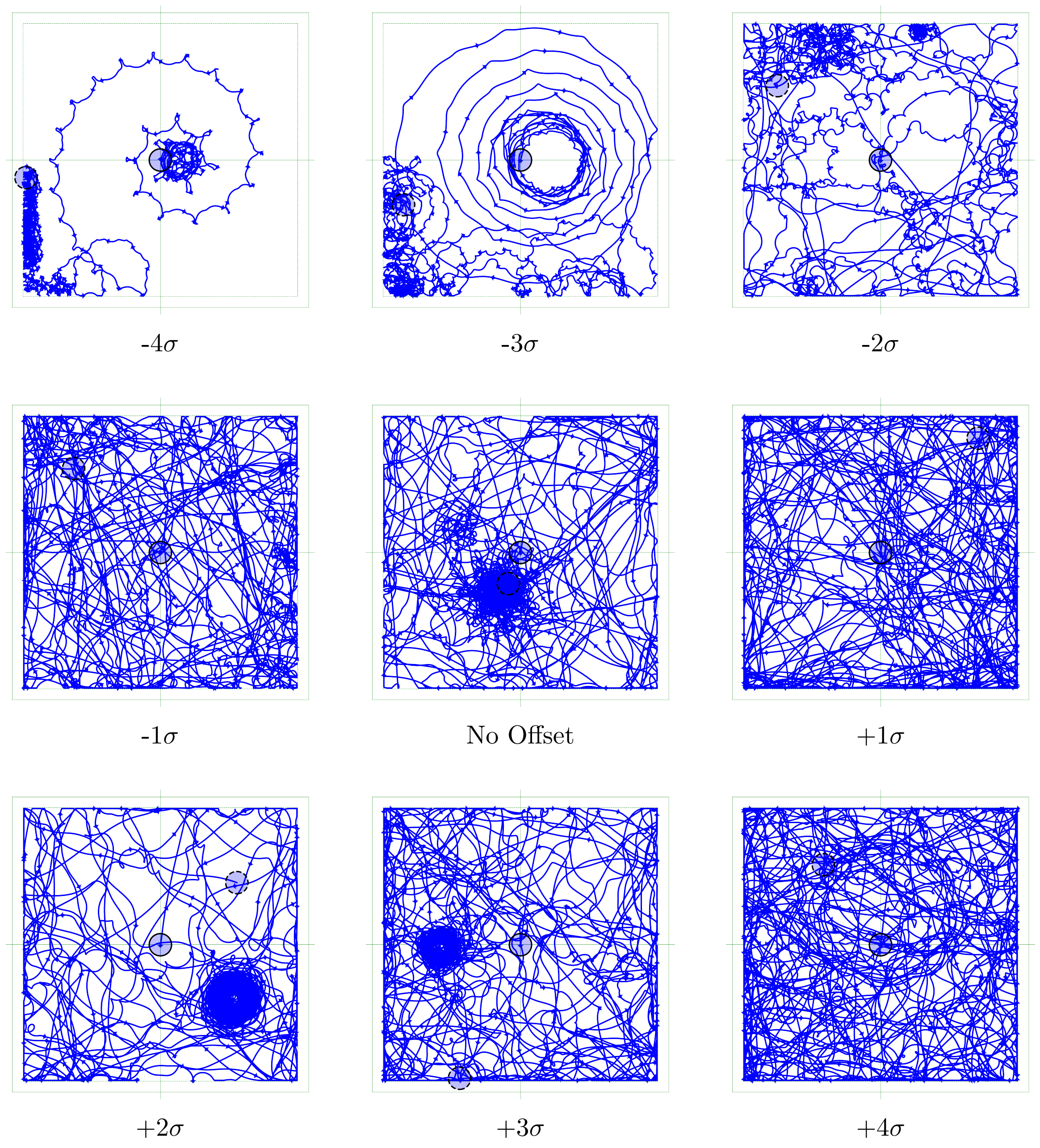


Figure 32: Offset Varied WEAZELBALL Trajectories Simulated by ODE

These plots depict simulation of the WB using the ODE dynamic engine in GAZEBO. Each plot represents the WB trajectory produced when the bob center-of-mass is displaced by a factor of a small amount σ as part of our validation process and reflects 300s of simulation time at an integration step size of 10μs. The bob center-of-mass was displaced both inward and outward with respect to the bob's modeled center-of-mass and the WB central axis. For each simulation, all parameters are held constant except for the positioning of the bob center-of-mass.

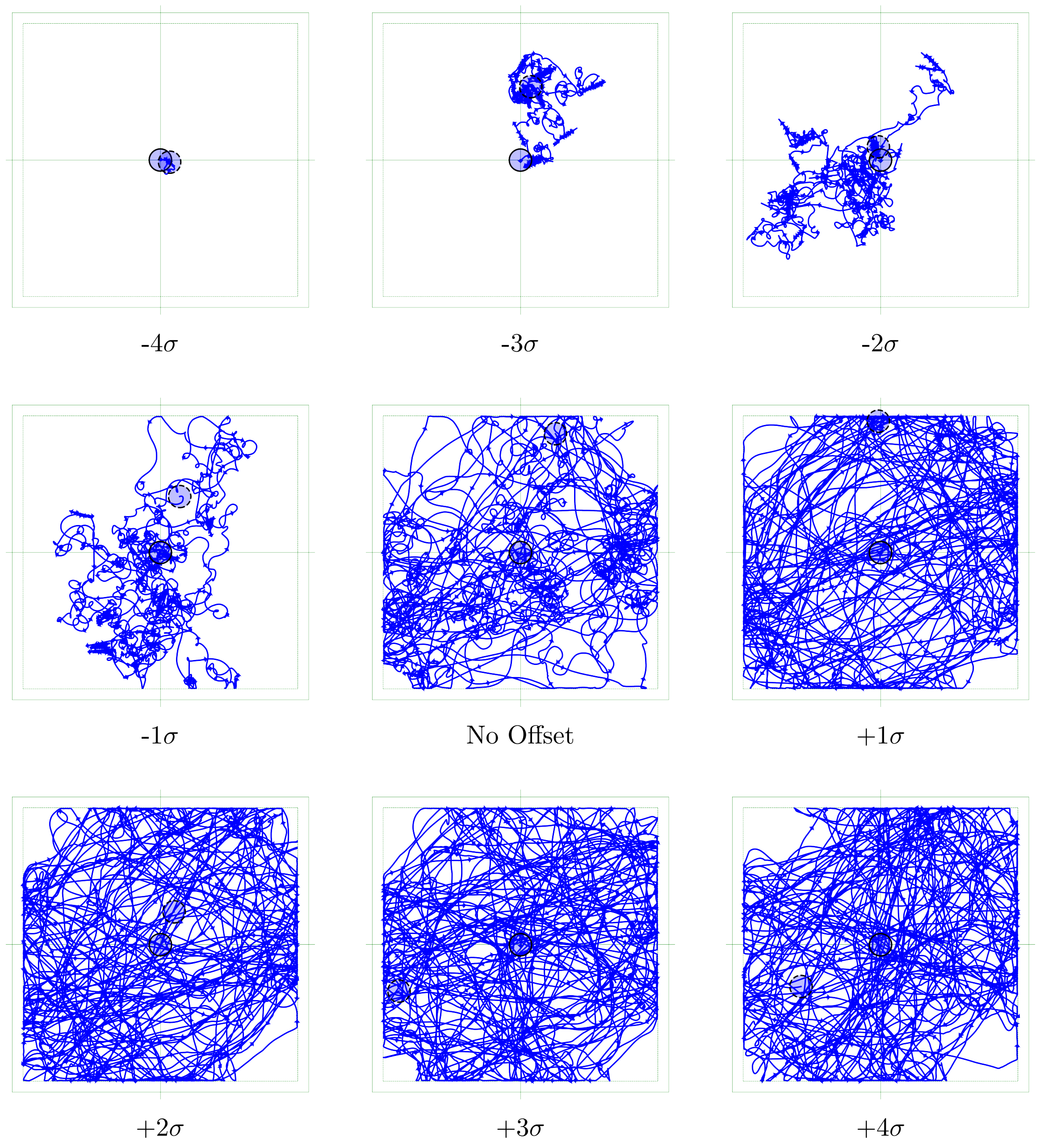


Figure 33: Offset Varied WEAZELBALL Trajectories Simulated by DART

These plots depict simulation of the WB using the DART dynamic engine in GAZEBO. Each plot represents the WB trajectory produced when the bob center-of-mass is displaced by a factor of a small amount σ as part of our validation process and reflects 300s of simulation time at an integration step size of 10μs. The bob center-of-mass was displaced both inward and outward with respect to the bob's modeled center-of-mass and the WB central axis. For each simulation, all parameters are held constant except for the positioning of the bob center-of-mass.

dimensional with mixed units (meters and radians), thereby precluding simple metrics for comparison; (2) upon impacts with its environment, the WB evinces chaotic behavior, as predicted by Ivanov [36]. The nonholonomicity of the WB, its primitive control system, and the intermittent chaotic behavior cause predictions to diverge from telemetry data given sufficient (usually small) time. The remainder of this section describes our approach to address these problems. The general idea is to (1) generate telemetry data for the WB *under multiple values of one parameter of the mechanism's model*, (2) train a regression model on the mapping from pairs of state data (from virtual sample i to virtual sample $i + 1$), and (3) test the regression mechanism on the data collected from VICON.

5.6.1 Simulating the Weazelball in Gazebo

We simulated the WB using GAZEBO 4.0. GAZEBO is robotics focused and allows simulation using a variety of dynamics engines with minimal or no changes to models and controllers. We hoped that GAZEBO would provide a dynamics-engine-agnostic interface; however, the interfaces and integration of all dynamics engines are not equally supported. We will discuss this issue in further detail below.

Simulation of the WB requires a model of the motor. We used a derivative controller ("D"-control) with an arbitrarily selected gain and the observed actuator frequency (2.5Hz, as described in Section 5.5) as the control input.

Simulating the WB also requires selecting an integration time step. This parameter has a large impact on simulation accuracy and running time. As the time step decreases, initial value problem solution accuracy increases which is accompanied by an increase in real computation time. For this reason, we prefer to take the largest step size possible and the smallest step size needed. To find an optimal step size, we ran a number of simulations where we reduced the fixed step size in each simulation until the behavior did not change appreciably (i.e., the behavior "converged").

Figure 34 shows the first 10s of simulation time for ODE and DART at various integration time steps with an actuator frequency of 2.333Hz. ODE appears to exhibit some convergence for a short period of time at smaller integration time steps; however, as the number of collisions increases, the state diverges as expected [36] as even small differences in position and orientation at the time of collision result in diverging states [10]. DART did not appear to converge regardless of step size.

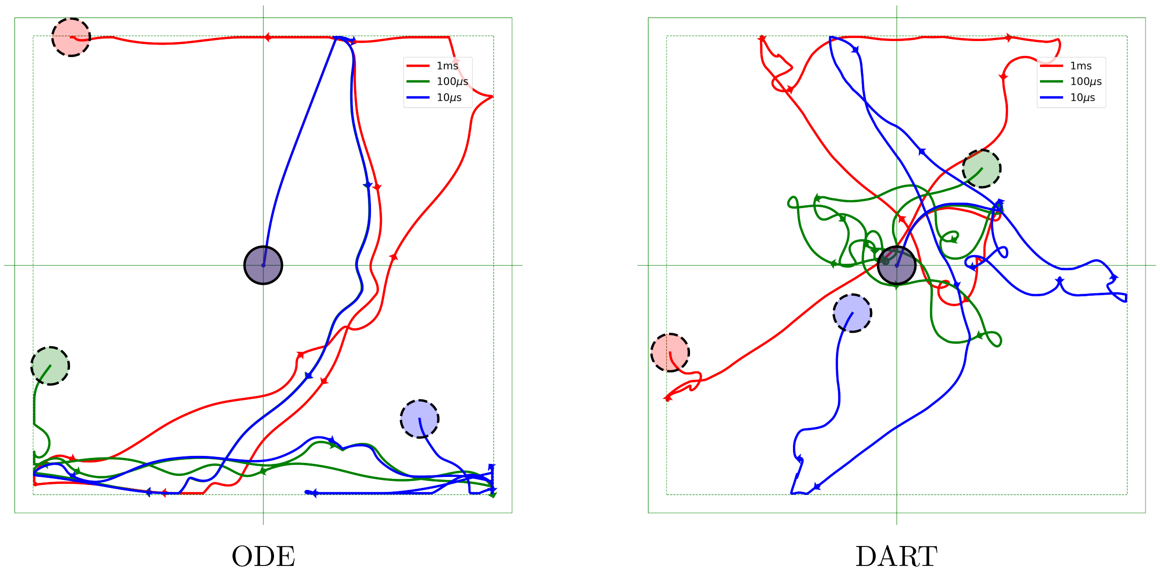


Figure 34: Simulated WEAZELBALL Trajectories with Varied Integration Step

These trajectories were generated over 10s of simulation time using the labeled simulator. The integration time step was varied for each simulation while all other parameters were held constant. All simulations begin with the WB positioned with the same state. Red lines map trajectories for 1ms time step, green lines map trajectories for 100μs time step, and blue lines map trajectories for 10μs time step. ODE appears to demonstrate convergence until several collisions have occurred while DART is unpredictable regardless of collisions.

5.6.2 Generating Training Data from Weazelball Simulation

We used the signed distance of the bob center-of-mass (CoM) as the mechanism parameter described at the beginning of this section. We varied this distance by adjusting the offset positively (i.e., the bob CoM was moved toward its "inner" joint) and negatively (i.e., the bob CoM was moved away from its joint). We represent the constant offset displacement as σ and use multiples of σ across the simulation experiments. After the model parameter was altered, we simulated the Wb and recorded the resulting telemetry data.

We carried out this process for two simulators that demonstrated plausible results: ODE and DART (other simulators supported by Gazebo were either too slow or exhibited gross artifacts). Once the regression model was trained using the data set from simulation, the model was made to predict the bob CoM offsets on the Vicon data set. These CoM offset predictions generated by the regression model thus provide a quantitative measure of how accurate a simulator is at generating “realistic” data.

Because we did not observe absolute convergence as step size is reduced for either simulator, we simulated the Wb using the smallest step size for which we could still simulate minutes of virtual time, 10μs. We carried out nine simulations per simulator, corresponding to four positive CoM offsets, four negative offsets, and one zero offset (a total of 18 simulations for the two simulators) for 5 minutes of simulation time. The four offsets in each direction were equally spaced, and the last offset was set such that the CoM of the bob would still lie well within the spherical shell. It is clear from Figures 32 and 33 that the spacings permitted significant variation in behavior.

We downsampled the simulation data to 100Hz to match the Vicon sampling rate before training. Because our simulation motor model assumes constant velocity, we generated all simulation training data using a 2.5Hz actuator frequency and we filtered the Vicon data by removing any samples not operating at 2.5Hz before predicting the offset. The 2.5Hz frequency corresponds to the Vicon samples where the Wb had reached maximum velocity: we observed the Wb actuator velocity occasionally increase or decrease after impacts with the walls of its enclosure. Since we assume that the actuator velocity is constant, we discarded Vicon samples that corresponded to telemetry data where the actuator did not operate at maximum velocity.

We built three simulation data sets from 30, 120, and 300 seconds of the simulated data illustrated in Figures 32 and 33, so that we could discern when added data produced no further prediction error decrease: both running the simulations and training the regression model require considerable computational time. Each training sample consists of a feature vector of two states (a total of thirty values), where each state comprises fifteen state values — shell position (3 values), shell orientation quaternion (4 values), shell linear velocity (3 values), shell angular velocity (3 values), actuator position, and actuator velocity — and a single output: the offset parameter corresponding to that feature. As already noted, simulation is carried out with an integration step size of 10μs but the training data is drawn from a subset of the simulation data that is generated by downsampling the simulation data to a frequency of 100Hz. Each training sample is therefore a concatenation of the i state and the $i + 1$ state from the downsampled 100Hz subset. The test set was taken from the Vicon data, collected and processed as described in the preceding sections of this chapter; state velocity values were approximated using a first-order finite difference between samples.

5.6.3 Training the Regression Model

We used the radial-basis function (RBF) based support vector regression (SVR) [13] library in `scikit-learn` to train the regression model. We chose RBF kernels over linear kernels because preliminary tests using the latter on small data sets resulted in significantly longer training time and greater error than with the RBF kernels. These trends (i.e., longer training times and larger errors) continued as more data was added, so we used the RBF kernels for the remainder of our experiments.

The `scikit-learn` RBF kernel implementation uses three primary parameters, `C`, `gamma`, and `epsilon`. `gamma` determines the amount of influence individual training samples exert on the classifier and `C` is used to specify the requisite accuracy for classification. The authors of the library conducted a search for the "best parameters" of `gamma` and `C`[4] and found that 0.1 and 1, respectively, generally were the best settings of these parameters. When searching for parameters via cross-validation, we initially provided a larger range of values but found that the scikit team's "best parameters" invariably yielded the lowest training error. The `epsilon` parameter constrains the region where no penalty is associated in the training loss function. `epsilon` had the most significant impact on training error, and we found that an `epsilon` value of 10^{-6} minimized training error regardless of the number of samples in the training set.

Each simulation data set was randomly shuffled and split to produce training sets and validation sets. We held back 10% of each training data set for cross-validation; this training/cross-validation split yielded our lowest training set error.

The trained regression model should predict a zero offset for all test set data, so the predicted offset produces a metric of the inaccuracy of the simulation. We measure regression error, ε, for a sample by computing the error between the predicted offset and the actual offset for the sample:

$$\varepsilon = |\sigma_{predicted} - \sigma_{actual}| \tag{9}$$

5.6.4 Analysis of Regression

Figures 35, 36, and 37 plot the results of the regression analysis for each simulation data set and Table 4 summarizes the mean error for all regression models. Error is expressed in the plots and accompanying table as a percentage relative to one CoM offset, *i.e.* 3.312mm; however, restating the error in terms of an absolute measurement provides perspective on the magnitude of the error. For 30s of ODE training data, the 4.43% training error is equivalent to 0.1467mm and the 7.26% testing error is equivalent to 0.24mm; for 30s of DART training data, the 4.46% training error is equivalent to 0.1477mm and the 17.28% testing error is equivalent to 0.5723mm.

One can draw interesting conclusions from both Figures 35, 36, and 37 and Table 4.

[4]`http://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html`

From the latter, we can glean that adding beyond 30s of data does not impact the results significantly. In fact, increasing the data by a factor of 10 yields only a 10% performance improvement in training error; error on prediction performance actually increases with more data for the two constructed regression models.

Are these results sensible from a qualitative perspective? In other words, does ODE actually appear to simulate the Wb better than DART? The answer is “yes”, at least looking at Figures 32 and 33, where DART trajectories (at zero CoM offset) show significant changes in direction in all simulations whereas ODE trajectories (again, at CoM offset) become hard to distinguish from real trajectories. The validation approach does appear to give a sense of which simulation is generating the more realistic data.

Nevertheless, we cannot conclude that ODE is a more accurate dynamics simulator than DART. Gazebo has a long development history with ODE while the interface between Gazebo 4.0 and DART was relatively new. As an example of the better support that ODE enjoyed, we note that the friction coefficients specified in the Gazebo models were not assigned to the DART bodies by Gazebo; DART computations relied instead on a default internal value. We modified the DART source to use the same value specified in the Gazebo models. While we carried out a close inspection of the friction parameters, DART might have been poorly supported in other ways.

5.6.5 Conclusions and Future Weazelball Work

We presented our approach for collecting data from and validating multi-rigid body simulations of a wild, robotic system. We provide a repository of our work[5]. Our repository includes all raw Vicon and video data collected; the Solidworks models and inertial estimates; Gazebo controllers developed to validate, synchronize, and interpolate; all signal and synchronization data; and all training data, implemen-

[5]`https://www.github.com/PositronicsLab/wild-robot`

Table 4: Regression error on training and testing data sets

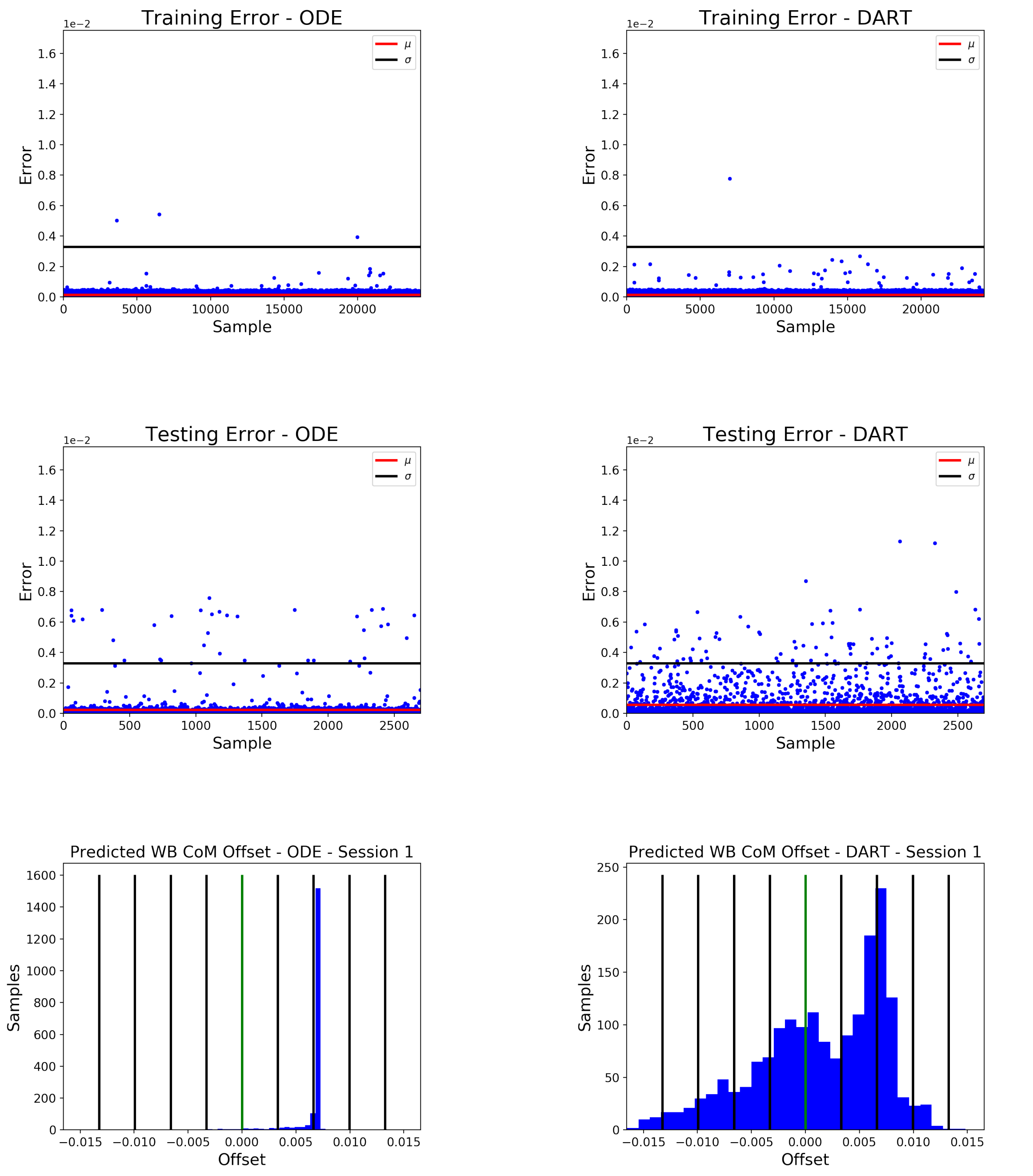
		ODE		DART	
t	Samples	$\varepsilon_{\text{train}}$	$\varepsilon_{\text{test}}$	$\varepsilon_{\text{train}}$	$\varepsilon_{\text{test}}$
$30s$	25000	4.43%	7.26%	4.46%	17.28%
$120s$	100000	4.08%	7.67%	4.20%	22.43%
$300s$	250000	4.04%	7.75%	4.07%	27.22%

Table 4 summarizes the error generated by our regression models for both ODE and DART training sets relative to one offset of the bob CoM, *i.e.* 3.312mm. t represents the duration in seconds of simulation data used for training, Samples represents the number of samples contained in a training set, and ε_{train} is the training error and ε_{test} is the testing error of the regression model for the given simulator.

tations, and support scripts necessary to recreate our validation process.

Our validation approach demonstrates one methodology that might be used to compare simulations with *in situ* data: see whether regression models trained by simulating a system from various initial conditions and a particular parameterization can predict the parameterization of the true system on real telemetry data. This approach is particularly useful when simulations are deterministic and do not attempt to introduce any noise, which should permit a sufficiently complex regression model to "learn" the simulation. This approach accommodates the high dimensional state spaces common in robotics, allows measurement of a simulator's sensitivity to the variation of a specific parameter, and can help establish a fair process for comparing the accuracy of simulations.

We report a caveat with our approach that lies with the model parameter selection: if the model parameter being varied (bob center-of-mass offset in our approach) is susceptible to ambiguity—a pair of states can be generated under multiple parameters—then such samples can adversely impact the model training. We also expect that one can identify that scenario, though, as training error will plateau regardless of the number of samples, the training time, or the regression model complexity. We believe this phenomenon is responsible for the performance plateaus observed in Table 4.



ODE DART

Figure 35: Regression Analysis for 30s of WEAZELBALL Simulation

These plots report the error and predicted offset for each regression model when trained by 30s of simulation data. The regression analysis for ODE appears in the left column and DART appears in the right column. In the training and testing error scatter plots, the black line represents one offset σ of the bob CoM and the red line represents the mean error μ. The histograms are centered on a green vertical line that represents the modeled and expected bob CoM and neighboring vertical black lines mark each inward and outward offset.

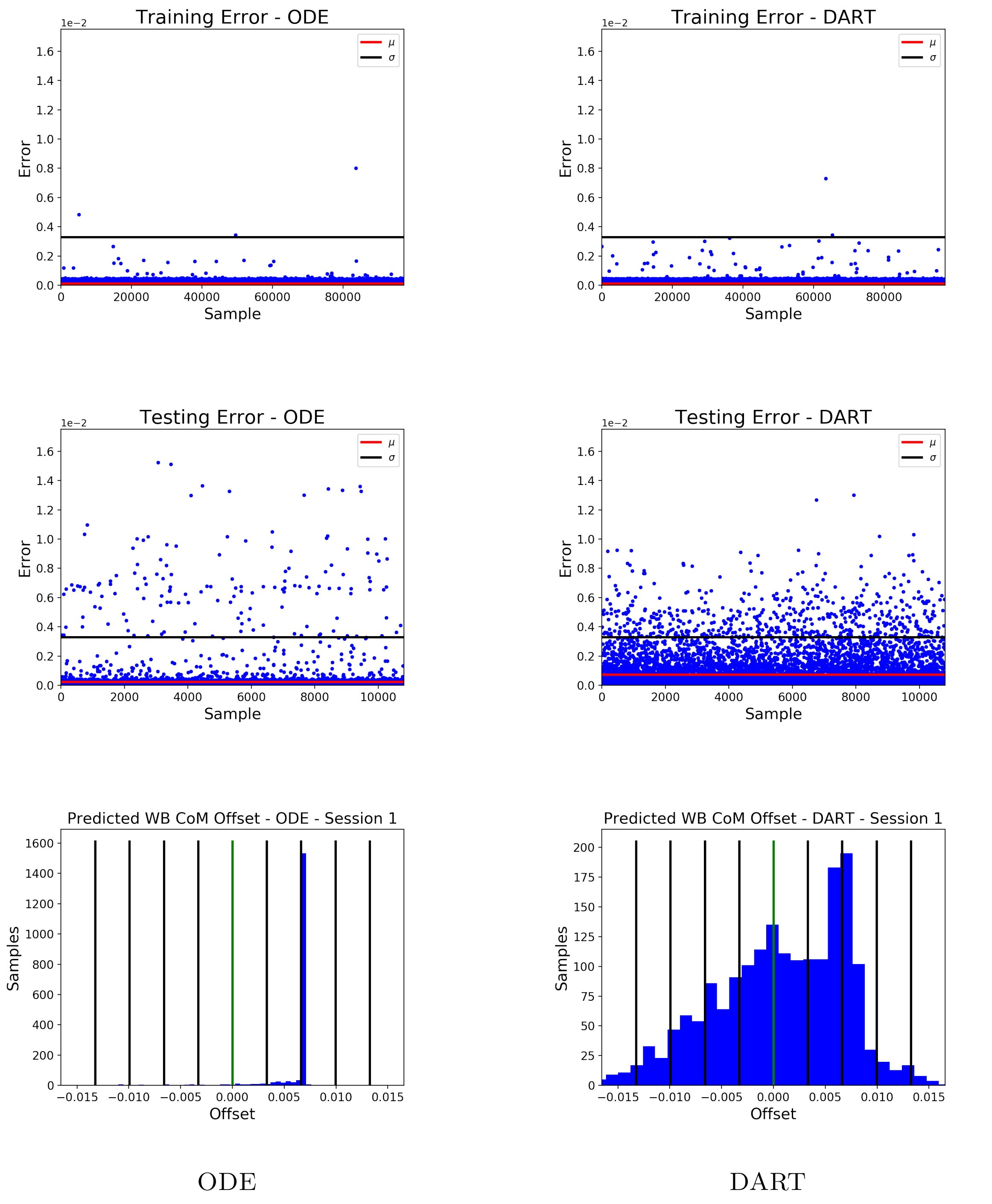


Figure 36: Regression Analysis for 120s of Weazelball Simulation

These plots report the error and predicted offset for each regression model when trained by 120s of simulation data. The regression analysis for ODE appears in the left column and DART appears in the right column. In the training and testing error scatter plots, the black line represents one offset σ of the bob CoM and the red line represents the mean error μ. The histograms are centered on a green vertical line that represents the modeled and expected bob CoM and neighboring vertical black lines mark each inward and outward offset.

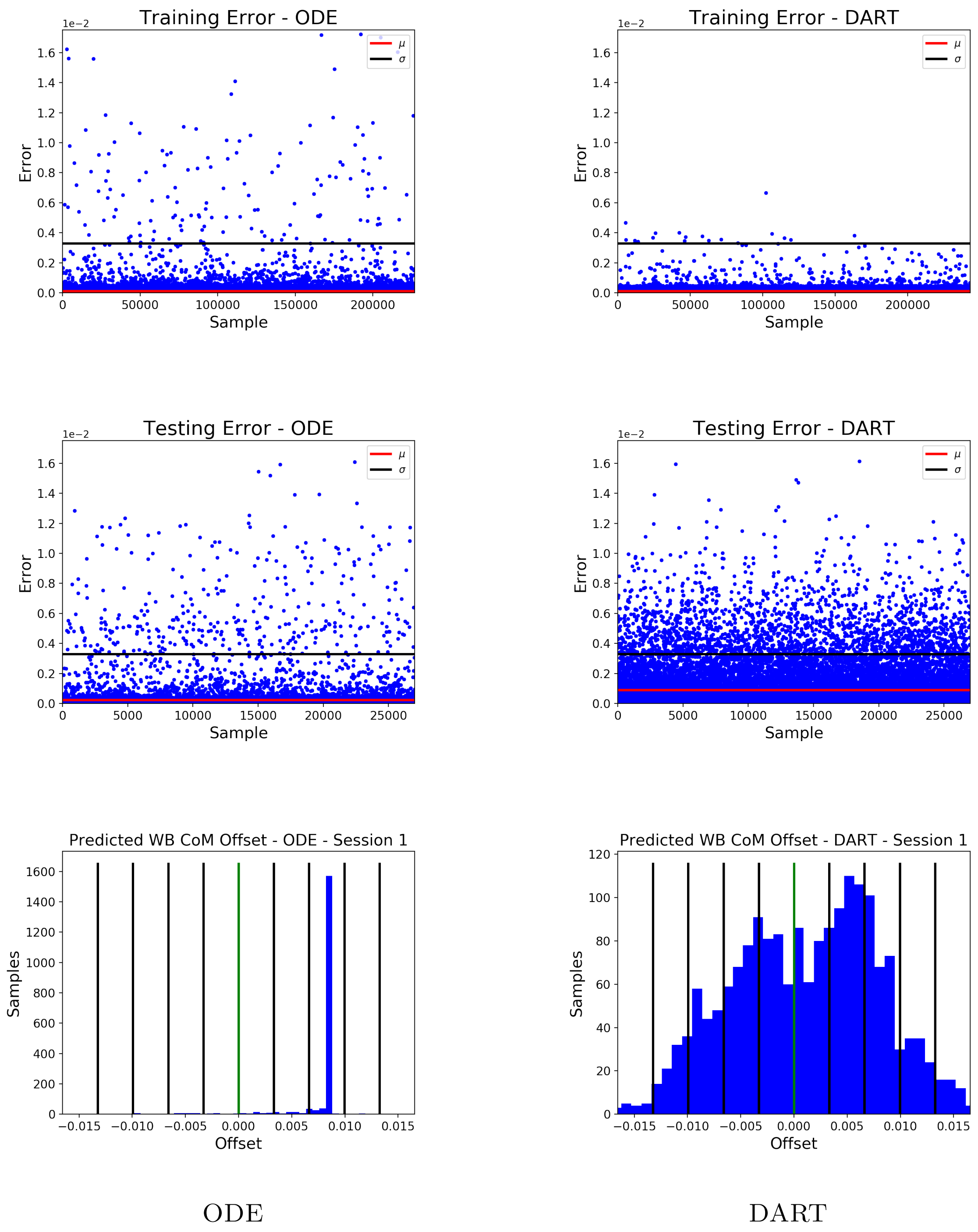


Figure 37: Regression Analysis for 300s of Weazelball Simulation

These plots report the error and predicted offset for each regression model when trained by 300s of simulation data. The regression analysis for ODE appears in the left column and DART appears in the right column. In the training and testing error scatter plots, the black line represents one offset σ of the bob CoM and the red line represents the mean error μ. The histograms are centered on a green vertical line that represents the modeled and expected bob CoM and neighboring vertical black lines mark each inward and outward offset.

We will continue to explore the potential and limitations of our validation approach and to look at other possible ways to validate simulations of the WB against the data generated by this study. However, an important future contribution is continued advocacy and promotion of the use of this and other real data sets to validate robotics simulators. Testing and demonstrating the predictive capabilities of robotics simulators will require data from a large number of relevant experiments and we believe that WB data along with data from other real world contact and friction experiments will advance the state of art in robotics simulation.

References

References

[1] J. H. Anderson and M. S. Mollison. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 283–292, 2013.

[2] M. Anitescu and F. A. Potra. Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems. *Nonlinear Dynamics*, 14:231–247, 1997.

[3] T. Aswathanarayana, D. Niehaus, V. Subramonian, and C. Gill. Design and performance of configurable endsystem scheduling mechanisms. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 32–43, 2005.

[4] J. Baumgarte. Stabilization of constraints and integrals of motion in dynamical systems. *Comp. Math. Appl. Mech. Engr.*, 1:1–16, 1972.

[5] L. Bobadilla, F. Martinez, E. Gobst, K. Gossman, and S. M. Lavalle. Controlling wild mobile robots using virtual gates and discrete transitions. In *American Control Conference(ACC)*, pages 743–749, Montréal, Canada, June 2012.

[6] L. Bobadilla, O. Sanchez, J. Czarnowski, K. Gossman, and S. M. LaValle. Controlling wild bodies using linear temporal logic. In *Proc. Robotics Science and Systems(RSS) VII*, pages 17–24, Los Angeles, USA, June 2011.

[7] A. Boeing and T. Bräunl. Evaluation of real-time physics simulation systems. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, GRAPHITE '07, pages 281–288, New York, NY, USA, 2007. ACM.

[8] B. Brogliato, A. ten Dam, L. Paoli, F. Génot, and M. Abadie. Numerical simulation of finite dimensional multibody nonsmooth mechanical systems. *Applied Mechanics Reviews*, 55(2):107–150, 2002.

[9] R. A. Brooks and M. J. Mataric. *Real Robots, Real Learning Problems*, pages 193–213. Springer US, Boston, MA, 1993.

[10] A. Chatterjee and A. Ruina. A new algebraic rigid-body collision law based on impulse space considerations. *J. Applied Mechanics*, 65(64):939–951, 1998.

[11] R. W. Cottle, J.-S. Pang, and R. Stone. *The Linear Complementarity Problem.* Academic Press, Boston, 1992.

[12] S. P. Dirkse and M. Ferris. The PATH solver: A non-monotone stabilization scheme for mixed complementarity problems. *Optimization Methods and Software*, 5:123–156, 1995.

[13] H. Drucker, C. J. Burges, L. Kaufman, A. J. Smola, and V. Vapnik. Support vector regression machines. In *Advances in neural information processing systems*, pages 155–161, 1997.

[14] E. Drumwright. Moby. https:://github.com/edumwri/Moby.

[15] E. Drumwright. A fast and stable penalty method for rigid body simulation. *IEEE Trans. on Visualization and Computer Graphics*, 14(1):231–240, Jan/Feb 2008.

[16] E. Drumwright and D. Shell. Extensive analysis of linear complementarity problem (LCP) solver performance on randomly generated rigid body contact problems. In *Proc. IEEE/RSJ Intl. Conf. Intelligent Robots and Systems (IROS)*, Vilamoura, Algarve, Oct 2012.

[17] E. Drumwright and D. A. Shell. A robust and tractable contact model for dynamic robotic simulation. In *Proc. of ACM Symp. on Applied Computing (SAC)*, pages 1176–1180, 2009.

[18] T. Erez, Y. Tassa, and E. Todorov. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4397–4404, Seattle, Washington, May 2015.

[19] P. L. Fackler and M. J. Miranda. LEMKE. http://people.sc.fsu.edu/ burkardt/m_src/lemke/lemke.m, July 2011.

[20] N. Fazeli, E. Donlon, E. Drumwright, and A. Rodriguez. Empirical evaluation of common contact models for planar impact. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3418–3425, Singapore, May 2017.

[21] R. Featherstone. An empirical study of the joint space inertia matrix. *The Intl. J. of Robotics Research*, 23(9):859–871, September 2004.

[22] B. Ford and S. Susarla. Cpu inheritance scheduling. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[23] O. S. R. Foundation. Simulation Description Format. http://sdformat.org.

[24] O. S. R. Foundation. Unified Robot Description Format. http://wiki.ros.org/urdf.

[25] M. Frigerio, V. Barasuol, M. Focchi, D. G. Caldwell, and C. Semini. Validation of computer simulations of the hyq robot. In *Proc. Intl. Conf. Climbing Walking Robots (CLAWAR)*, 2017.

[26] D. E. Gierl, L. Bobadilla, O. Sanchez, and S. M. Lavalle. Stochastic modeling,

control, and verification of wild bodies. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 549–556, Hong Kong, China, May 2014.

[27] M. González, D. Dopico, U. Lugrís, and J. Cuadrado. A benchmarking system for mbs simulation software: Problem standardization and performance measurement. *Multibody System Dynamics*, 16(2):179–190, 2006.

[28] M. González, F. González, A. Luaces, and J. Cuadrado. A collaborative benchmarking framework for multibody system dynamics. *Engineering with Computers*, 26(1):1–9, 2009.

[29] K. Group. Collaborative Design Activity (COLLADA) Format. https://www.khronos.org/collada/.

[30] J. Güèmez, R. Valiente, C. Fiolhais, and M. Fiolhais. Experiments with the drinking bird. *American Journal of Physics*, 71:1257–1263, 12 2003.

[31] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex rigid bodies with stacking. *ACM Trans. on Graphics*, 22(3):871–878, 2003.

[32] S. Hart, R. Grupen, and D. Jensen. A relational representation for generalized knowledge in robotic tasks. Technical Report 04-101, Computer Science Dept, Univ. of Massachusetts Amherst, 2004.

[33] L. Hatton and A. Roberts. How accurate is scientific software? *IEEE Trans. Softw. Eng.*, 20(10):785–797, Oct. 1994.

[34] J. M. Hsu and S. C. Peters. Extending open dynamics engine for the DARPA virtual robotics challenge. In *Proc. Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2014.

[35] J. Hummel, R. Wolff, T. Stein, A. Gerndt, and T. Kuhlen. *Advances in Visual Computing: 8th International Symposium, ISVC 2012, Rethymnon, Crete,*

Greece, July 16-18, 2012, Revised Selected Papers, Part II, chapter An Evaluation of Open Source Physics Engines for Use in Virtual Reality Assembly Simulations, pages 346–357. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[36] A. P. Ivanov. On multiple impact. *J. Applied Mathematics and Mechanics*, 59(6):887–902, 1995.

[37] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

[38] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proc. of IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 2149–2154, Sendai, Japan, Sept 2004.

[39] R. Kolbert, N. C. Dafle, and A. Rodriguez. Experimental validation of contact dynamics for in-hand manipulation. In *2016 IEEE International Symposium on Experimental Robotics (ISER)*, Tokyo, Japan, October 2016.

[40] C. Lacoursière. Splitting methods for dry frictional contact problems in rigid multibody systems: Preliminary performance results. In M. Ollila, editor, *Proc. of SIGRAD*, pages 11–16, Nov 2003.

[41] C. Lacoursière. *Ghosts and Machines: Regularized Variational Methods for Interactive Simulations of Multibodies with Dry Frictional Contacts.* PhD thesis, Umeå University, 2007.

[42] A. D. Lewis and R. M. Murray. Variational principles for constrained systems: Theory and experiment. *The International Journal of Nonlinear Mechanics*, 30(6):793–815, 1995.

[43] Y. Lu, J. Williams, J. Trinkle, and C. Lacoursire. A framework for problem standardization and algorithm comparison in multibody system. In *10th Inter-*

national Conference on Multibody Systems, Nonlinear Dynamics, and Control, volume 6 of *IDETC/CIE 2014*, 2014.

[44] B. Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems.* PhD thesis, University of California, Berkeley, 1996.

[45] P. C. Mitiguy and A. K. Banerjee. Efficient simulation of motions involving Coulomb friction. *J. Guidance, Control, and Dynamics*, 22(1), Jan–Feb 1999.

[46] K. G. Murty. *Linear Complementarity, Linear and Nonlinear Programming.* Heldermann Verlag, Berlin, 1988.

[47] D. Negrut, R. Serban, A. Elmquist, and D. Hatch. Synchrono: An open-source framework for physics-based simulation of collaborating robots. In *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 101–107, May 2018.

[48] P. E. Nikravesh. *Computer-Aided Analysis of Mechanical Systems.* Prentice Hall, 1988.

[49] W. Oberkampf and T. Trucano. Verification and validation in computational fluid dynamics. *Progress in Aerospace Sciences*, 38(3):209–272, 2002.

[50] G. Parmer and R. West. HiRes: A system for predictable hierarchical resource management. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2011.

[51] F. Pfeiffer. Mechanische systeme mit unstetigen übergängen. *Ingenieur-Archiv*, 54(3):232–240, May 1984.

[52] F. Pfeiffer and C. Glocker. *Multibody Dynamics with Unilateral Contacts.* John Wiley and Sons, Inc., New York, New York, 1996.

[53] F. A. Potra, M. Anitescu, B. Gavrea, and J. Trinkle. A linearly implicit trapezoidal method for stiff multibody dynamics with contact, joints, and friction. *Intl. Journal for Numerical Methods in Engineering*, 66(7):1079–1124, 2006.

[54] A. Rocci, B. Ames, Z. Li, and K. Hauser. Stable simulation of underactuated compliant hands. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4938 – 4944, Stockholm, Sweden, May 2016.

[55] S. Schaal. The SL simulation and real-time control software package. Technical report, Univ. Southern California, 2009.

[56] S. Schlesinger, R. E. Crosbie, R. E. Gagné, G. S. Innis, C. Lalwani, J. Loch, R. J. Sylvester, R. D. Wright, N. Kheir, and D. Bartos. Terminology for model credibility. *SIMULATION*, 32(3):103–104, 1979.

[57] A. Seugling and M. Rölin. Evaluation of physics engines and implementation of a physics module in a 3d-authoring tool. Master's thesis, Umeå University, Sweden, 2006.

[58] A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of synchronization and consistency in beehive. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, SPAA '97, pages 211–220, New York, NY, USA, 1997. ACM.

[59] R. Smith. ODE: Open Dynamics Engine.

[60] R. Sondergaard, K. Chaney, and C. E. Brennen. Measurements of solid spheres bouncing off flat plates. *Journal of Applied Mechanics*, 57(3):694–699, 1990.

[61] P. Song, M. Yashima, and V. Kumar. Dynamic simulation for grasping and whole arm manipulation. In *Proc. IEEE Intl. Conf. Robot. Autom. (ICRA)*, San Francisco, CA, USA, Apr 2000.

[62] S. S. Srinivasa, D. Ferguson, C. J. Helfrich, D. Berenson, A. Collet, R. Diankov, G. Gallagher, G. Hollinger, J. Kuffner, and M. V. Weghe. HERB: a home exploring robotic butler. *Autonomous Robots*, 28(1):16, Nov 2009.

[63] D. Stewart and J. C. Trinkle. An implicit time-stepping scheme for rigid body dynamics with Coulomb friction. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA)*, San Francisco, CA, April 2000.

[64] D. E. Stewart and J. C. Trinkle. An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and Coulomb friction. *Intl. J. Numerical Methods in Engineering*, 39(15):2673–2691, 1996.

[65] D. Stoianovici and Y. Hurmuzlu. A critical study of the applicability of rigid-body collision theory. *Journal of Applied Mechanics*, 63(2):307–316, 1996.

[66] J. R. Taylor, E. Drumwright, and G. Parmer. Making time make sense in robotic simulation. In *Proceedings ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2014, Buffalo, New York, USA, August 17-20, 2014*, 2014.

[67] E. Todorov. A convex, smooth and invertible contact model for trajectory optimization. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, Shanghai, 2011.

[68] T. Vose, P. Umbanhowar, and K. M. Lynch. Friction-induced velocity fields for point parts sliding on a rigid oscillated plate. *Intl. J. of Robotics Research*, June 2009.

[69] H. A. Yanco, A. Norton, W. Ober, D. Shane, A. Skinner, and J. Vice. Analysis of human-robot interaction at the DARPA robotics challenge trials. *Journal of Field Robotics*, 32(3):420–444, May 2015.

[70] T. Ylikorpi and J. Suomela. Ball-shaped robots. In H. Zhang, editor, *Climbing & Walking Robots, Toward New Applications*, chapter 11, pages 546–567. Itech Education and Publishing, Vienna, Austria, October 2007.

[71] K.-T. Yu, M. Bauza, N. Fazeli, and A. Rodriguez. More than a million ways to be pushed. a high-fidelity experimental dataset of planar pushing. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 30–37, Oct 2016.

[72] L. Zhang, J. Betz, and J. C. Trinkle. Comparison of simulated and experimental grasping actions in the plane. In *First Joint International Conference on Multibody System Dynamics*, May 2010.