Virtual Articulation and Kinematic Abstraction in Robotics

by

Marsette Arthur Vona, III

B.A., Dartmouth College (1999)
S.M., Massachusetts Institute of Technology (2001)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

Author .... [illegible]
Department of Electrical Engineering and Computer Science
September 1, 2009

Certified by ....
Daniela Rus
Professor
Thesis Supervisor

Accepted by .... [illegible]
Terry P. Orlando
Chairman, Department Committee on Graduate Theses

Virtual Articulation and Kinematic Abstraction in Robotics

Marsette Arthur Vona, III

Submitted to the Department of Electrical Engineering and Computer Science
on September 1, 2009, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

This thesis presents the theory, implementation, novel applications, and experimental validation of a general-purpose framework for applying virtual modifications to an articulated robot, or *virtual articulations*. These can homogenize various aspects of a robot and its task environment into a single unified model which is both qualitatively high-level and quantitatively functional.

This is the first framework designed specifically for the mixed real/virtual case. It supports arbitrary topology spatial kinematics, a broad catalog of joints, on-line structure changes, interactive kinostatic simulation, and novel *kinematic abstractions*, where complex subsystems are simplified with virtual replacements in both space and time. Decomposition algorithms, including a novel method of hierarchical subdivision, enable scaling to large closed-chain mechanisms with 100s of joints.

Novel applications are presented in two areas of current interest: operating high-DoF kinematic manipulation and inspection tasks, and analyzing reliable kinostatic locomotion strategies based on compliance and proprioception. In both areas virtual articulations homogeneously model the robot and its task environment, and abstractions structure complex models. For high-DoF operations the operator attaches virtual joints as a novel interface metaphor to define task motion and to constrain coordinated motion (by virtually closing kinematic chains); virtual links can represent task frames or serve as intermediate connections for virtual joints. For compliant locomotion, virtual articulations model relevant compliances and uncertainties, and temporal abstractions model contact state evolution.

Results are presented for experiments with two separate robotic systems in each area. For high-DoF operations, NASA/JPL's 36 DoF ATHLETE performs previously challenging coordinated manipulation/inspection moves, and a novel large-scale (100s of joints) simulated modular robot is conveniently operated using spatial abstractions. For compliant locomotion, two experiments are analyzed that each achieve high reliability in uncertain tasks using only compliance and proprioception: a novel vertical structure climbing robot that is 99.8% reliable in over 1000 motions, and a mini-humanoid that steps up an uncertain height with 90% reliability in 80 trials. In both cases virtual articulation models capture the essence of compliant/proprioceptive strategies at a higher level than basic physics, and enable quantitative analyses of the limits of tolerable uncertainty that compare well to experiment.

Thesis Supervisor: Daniela Rus
Title: Professor

Acknowledgments

The ideas presented in this thesis tie together several long-standing threads in my own personal intellectual development. The oldest of these, an interest in the geometric mechanisms of robots for their practical and aesthetic values, reaches back to my childhood. Then as now, my parents have enthusiastically and unfailingly supported each and every one of my projects, and deserve the first thanks. My mother often tells the story of the first "machine" I ever built that "really worked": an indoor apparatus for melting snow. My father unexpectedly passed away just a few months before the final submission of this dissertation. I still remember those early science projects you helped me with, Dad: making a battery from a lemon and strips of metal from the hardware store; stringing a working telegraph—complete with actual Morse keys—across my first-grade classroom; the white-hot "light bulb" filament that worked so well only with your special dental ligature wire. Thanks Dad. You will always be the first "Dr. Vona."

In terms of long-term support, my adviser Daniela Rus comes in an unusually close second-place to my parents. I have had the uncommon opportunity to work with Daniela both for my undergraduate thesis, which I completed at Dartmouth college in 1999, as well as for this dissertation at MIT. At Dartmouth Daniela introduced me to the field of self-reconfiguring robots, a still-productive line of research that includes the large-scale modular tower example in Section 5.4. Since then, Daniela has been instrumental in most of the major transitions in my career. Rather than rigidly telling me what to do, she has always enabled and encouraged me to follow the ideas that are the most productive and interesting.

This thesis entailed significant theoretical development, practical physical implementation of several robotic systems, and experiments using both those systems and also the ATHLETE robot at NASA's Jet Propulsion Laboratory. Many people have helped in each of these three areas.

I thank the other members of my Ph.D. committee, Rodney Brooks and Erik Demaine, for inspiration and guidance in the conceptual and theoretical directions of the work. In this realm I also am grateful for key insights gained in discussions with Keith Kotay, with whom I worked both at Dartmouth and at MIT; Iuliu Vasilescu and Eduardo Torres-Jara, also recent graduates of MIT; Russ Tedrake, associate professor of EECS at MIT; and other members of past and current research groups including in particular Nikolaus Correll, Mac Schwager, Peter Osagie (who designed and implemented the path planner for the Shady climbing robot presented in Section J.3.2), and Robert Fitch. Special thanks also go to Sanjay Sarma, professor of Mechanical Engineering at MIT, with whom I studied during my first year back at MIT in 2003-4 (after working at JPL for two years between my M.S. and Ph.D.); and to Ulrich Kortenkamp, now a professor at the University of Education Karlsruhe, who introduced me to the important property of *continuity* in interactive geometry software.

As a (primarily) computer science student, I have had much to learn about the electromechanical design and fabrication that was needed for the interface device and the climbing robot hardware I developed for this thesis (Appendices I and J). Keith and Iuliu have helped me innumerable times here as well, as have other students in our group including Carrick Detweiler (who designed the motor control boards used on the climbing robot) and Marek Doniec. I learned about mechanical systems, sensing, control, and actuation during my M.S. work with David Trumper, a professor of Mechanical Engineering at MIT, and from my co-students at that time, including Marten Byl, Katie Byl, and Michael Liebman. I also thank Mark Yim, now professor of Mechanical Engineering at the University of Pennsylvania, and Satoshi Murata, a professor at Tokyo Institute of Technology, both founders in the area of self-reconfiguring robot hardware. Over the years I have picked up valuable knowledge from each of them, both in person and through their writings. For example, the singular-linkage grip mechanism in the climbing robot was motivated after a particular discussion with Professor Murata.

Some of the key experimental work in this thesis (Section 5.3) was carried out using the excellent ATHLETE hardware at JPL, which was developed there by Brian Wilcox, Todd Litwin, Jeff Biesiadecki, Jaret Matthews, Matt Heverly, and others. Jeff Norris, a long-time collaborator, and now the supervisor of the Planning Software Systems group at JPL, was instrumental both in developing and supporting the concept and details of these experiments, as well as in arranging the logistics. David Mittman was the main on-the-ground collaborator during my visit to JPL, and deserves a major portion of the credit for the success of our experiments with ATHLETE. The graphical simulation model of the robot pictured in this thesis is provided courtesy of the RSVP Team at NASA/JPL.

Early experiments with the stair-stepping mini-humanoid in Section 6.7 were developed by Albert Wang and Jeremy Lai, then both undergraduates at MIT. They developed the CAD model of the robot pictured in this thesis, based originally on a public-domain model from Pedro Teodoro at the Technical University of Lisbon.

Melissa King has been an inspiring companion for almost exactly as many years as I have been at work on this thesis. It's hard to imagine how I could have done it without her support and understanding. Other friends that have stuck with me through this long process include Ben Guaraldi, the Homer extended family, Joe Edelman, and Brent Knopf. Cleopatra King, though she doesn't talk and has no thumbs, has suggested to me some of the amazing potential for low-impedance locomotion mechanisms.

Work with ATHLETE was funded in part by a grant from the JPL/Caltech Directors Research and Development Fund Strategic University Research Partnership program. Other aspects of this thesis were funded by the National Science Foundation under the Emerging Frontiers in Research and Innovation program.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Normally the kinematic topology of an articulated robot—the structure of its links and joints—is considered invariant[1]. Change is typically localized to motion in the joints. This thesis explores the possibilities enabled by virtually relaxing that assumption, i.e., by allowing virtual links and joints—*virtual articulations*—to be interconnected with a model of the robot as it operates.

Some specific virtual articulations useful in particular situations have already been reported in the literature [19, 8, 76, 127, 115, 1]. I consider the general case, and develop a full-featured framework for kinostatic modeling and simulation of arbitrary-topology three dimensional *mixed real/virtual* articulated systems. This framework covers both open- and closed-chain kinematics, includes a rich set of mutation primitives for topological dynamics, and provides real-time motion computation algorithms that scale to large structures of up to hundreds of joints. I present the theory in detail, and then show novel applications, using a full software implementation of the framework, in two domains of current interest: human interfaces to operate coordinated motion in robots with large numbers of joints, and qualitative and quantitative models that capture the essence of reliable locomotion strategies based on compliance and proprioception. Experiments are presented with two separate robotic systems in each domain; three of these four systems are themselves novel contributions, and the fourth is a new robot under development at NASA's Jet Propulsion Laboratory.

[1]Robots specifically designed for reconfiguration are one exception.

1.1 Virtual Articulations

To see how virtual articulations can help operate a robot with many joints, consider an example with this latter robot. With 36 revolute joints in six identical limbs, *ATHLETE* [167] is highly flexible and can in theory be applied in a broad range of tasks. Often, an intended motion is easily visualized, but expressing it could be arbitrarily tedious in prior interfaces (e.g. [103]). Virtual articulations can help. Figure 1-1 shows a basic example: the operator would like to tilt the field of view of a camera that is rigidly embedded in the robot's frame, effectively rotating the body about a virtual axis in space, with the legs moving as necessary. Using the *mixed real/virtual interface*, my implementation of the framework, the intended rotation axis can be graphically defined by attaching a virtual revolute joint between the robot body and the ground. Rotating the joint drives the intended motion, and the system automatically computes compatible motions for all the legs.

Much of the power of virtual articulations comes from their capacity to *homogenize* various aspects of a robot and its task environment into a single unified model, leading to the following thesis statement:

By homogeneously combining a robot with its task, virtual articulations enable models that are both qualitatively high-level and quantitatively functional.

The task constraint in the above example—to rotate the camera about a fixed axis—is homogeneously modeled as a revolute joint, interleaved directly with the joints and links of the robot itself.

Homogenization is also clearly seen in a second example, this time from the domain of compliant mechanisms. It has long been known that clever use of compliance can confer reliability even under significant uncertainty—e.g., the remote compliance center (RCC) wrist is a seminal compliant mechanism for reliable peg-in-hole assembly [46, 32]. But there has been little work towards building a higher-level understanding of the relation between compliance and uncertainty tolerance [122, 93]. As illustrated in Figure 1-2, virtual articulations can help by modeling uncertainties, compliances, and contacts in a single homogeneous model including the robot itself.

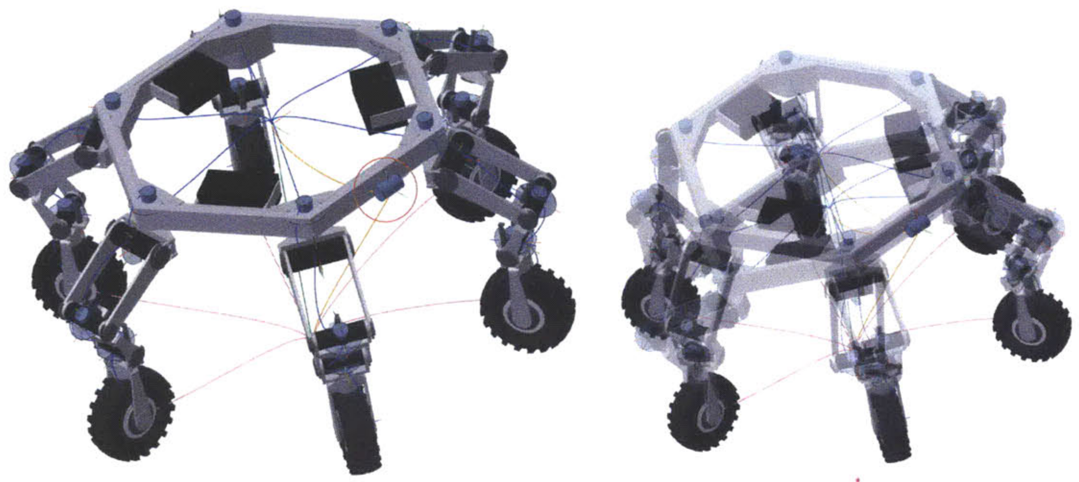


Figure 1-1: Operating ATHLETE in the *mixed real/virtual interface*. Virtual articulations can be added by an operator as a means of constraining motion and defining task-relevant DoF. In this example, a virtual revolute joint (circled) is added to facilitate tilting the field of view of one of ATHLETE's cameras, which are rigidly mounted to the robot (the camera itself is not visible in the model—it is actually embedded in the frame). The camera roll can be explicitly commanded by directly rotating the virtual joint; kinematically feasible whole-robot motions are automatically computed on-line. Figure 5-7 shows a corresponding experiment on the actual hardware. ATHLETE graphical model courtesy the RSVP team, NASA/JPL/Caltech.

Such models can be significantly higher-level than afforded by basic physics, but can still capture the essence of a compliant mechanism. And, when properly calibrated, they can also make quantified predictions, such as the limits of tolerable uncertainty, with reasonable fidelity.

1.2 Kinematic Abstractions

While virtual articulations are useful directly, I go a step further. Computational abstraction is well known for managing complexity in algorithmic systems, but so far there has been no corresponding formalism in kinematics. By applying virtual articulations in a specific hierarchical framework, I show that a novel technique of *structure abstraction* can be used to effectively hide the complexity of a kinematic sub-

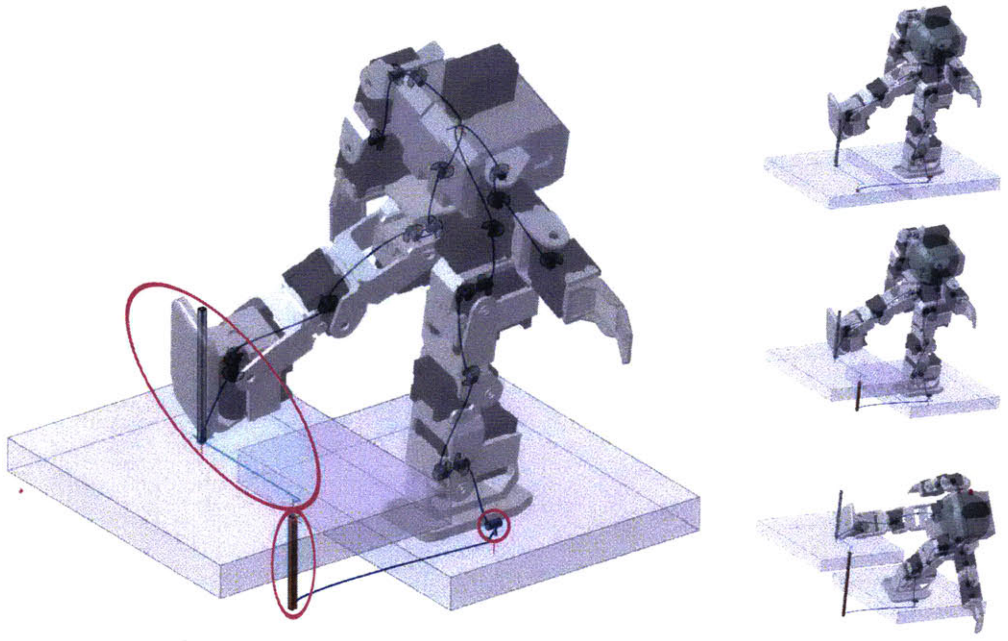


Figure 1-2: Modeling compliant step climbing in the mixed real/virtual interface. Virtual articulations can form a homogeneous model including uncertainties—here, the unknown step height, modeled as a virtual prismatic joint; compliances (in this case, all the robot joints are compliant); and contacts. Such models are useful to aid qualitative understanding and communication, and can also make quantitative predictions. Here, the minimum and maximum permissible step heights are predicted based on simulated first heel contact and fall, respectively. Figure 6-9 shows a corresponding hardware experiment.

system in a way that corresponds to traditional abstraction in computation. Figure 1-3 shows a simple example where a series chain of four revolute joints is abstracted as a piston-like assembly, hiding the detailed motion of the middle link.

In some cases, complexity arises not from the structure of the robot itself, but rather from the evolution of contact interactions between robot and environment. I thus also introduce *sequence abstraction* to form higher-level virtual models that abstract the potentially complex details of a broad class of such interactions. Whereas structure abstractions operate in the spatial domain, sequence abstractions hide complexity in the time domain. The virtual articulations at each foot of the humanoid in Figure 1-2 are each sequence abstractions: the left (support) foot normally rests on the lower platform, but tilts up when the robot begins to fall. The right foot is either

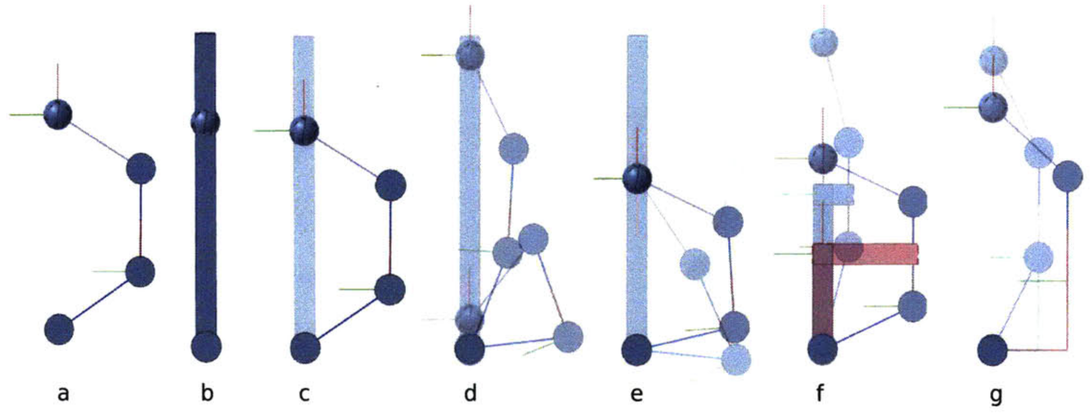


Figure 1-3: Structure abstraction and kinematic constraint.
Virtual articulations can be used both for *structure abstraction* and to constrain intended motion. The actual robot mechanism in (a) is an open chain of four revolute joints. The operator is mainly interested in the piston-like behavior of the endpoints. In the mixed real/virtual interface, this can be directly modeled as a revolute-prismatic-revolute construction (b) which virtually replaces the model of the actual mechanism (c,d), in the same way that an abstraction for an algorithm can be considered to stand in for its implementation. Here, the detailed motion of the middle link is hidden below the abstraction; left alone, this link is free to "flop around" (e); the operator chooses to constrain it parallel to the piston axis by adding an assembly of two virtual prismatic joints (f) below the abstraction barrier. The same constraint can also be defined using a shorthand *Cartesian-2* joint (g).

dangling or contacts the upper platform at the heel corner, depending on the step height, which is unknown to the robot. In both cases multiple contact configurations are modeled with a single persistent virtual assembly.

1.3 Organization of the Thesis

The remaining sections of this chapter give a brief summary of the implementation of the mixed real/virtual interface and highlights of the contributions of the thesis.

Chapter 2 will situate these contributions relative to prior works in virtual articulations and related concepts. This high-level review is complimented later by more focused discussions for each of the two application areas in Section 5.5 and 6.8, and also for the novel systems I introduce in Appendices J, K, and L.

Chapters 3 and 4 describe the framework for mixed/real virtual models in detail. The first focuses on the structure of the model and related topological mutation algorithms; the second adds algorithms to compute motions during interactive use.

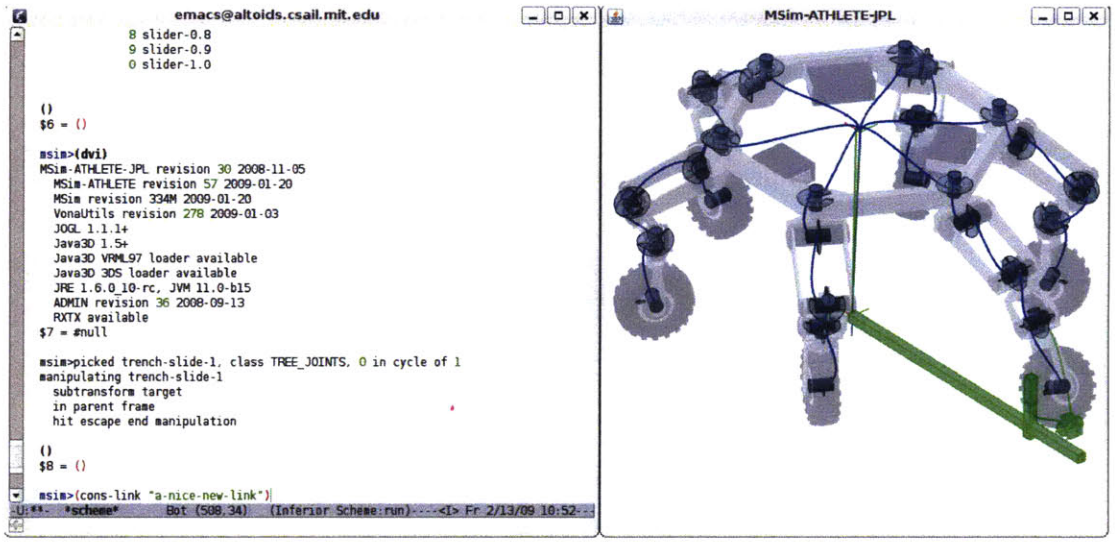
Chapter 5 develops the concept of using virtual articulations and structure abstraction as an operations interface for high-DoF robots, and presents applications both for ATHLETE and for a novel modular robot I introduce called *Multishady.* The latter is in simulation only, but shows the scalability of the motion computation algorithms to topologically large closed-chain systems with hundreds of joints.

Chapter 6 presents the second class of applications, to compliant/proprioceptive locomotion. The idea of sequence abstraction is developed here, as is a general procedure for quantitatively studying the relationship between the compliances in a system and the amount of uncertainty tolerance they afford. Two particular applications are presented: *Shady*, a novel scratch-built compliant/proprioceptive structure climbing robot; and *Steppy*, an off-the-shelf mini-humanoid for which I developed a unique compliant/proprioceptive control strategy for climbing a step of uncertain height.

Chapter 7 summarizes the advances and limitations of the ideas presented in the thesis, and describes potential future work. A number of appendices follow covering various details both from the theory and from the applications.

1.4 System Implementation

A major part of the work behind this thesis was a full software implementation of the framework, called the *mixed real/virtual interface.* This implementation includes essentially all data structures and algorithms presented in Chapters 3 and 4 (any omissions are noted in-context). Figure 1-4 shows a typical screenshot. The core implementation is fully generic, and not tied to any particular robot or task. The application examples in chapters 5 and 6 were all built directly within it—in most cases that involved writing some scripts so that specific setups could be repeatedly re-constructed during the experiments. Interfacing with the ATHLETE systems at JPL also required some special-purpose data import/export code. Figure 1-5 shows a breakdown of code by function, both in the core system and for the applications.



```
              8 slider-0.8
              9 slider-0.9
              0 slider-1.0


()
$6 = ()

msim>(dvi)
MSim-ATHLETE-JPL revision 30 2008-11-05
  MSim-ATHLETE revision 57 2009-01-20
  MSim revision 334M 2009-01-20
  VonaUtils revision 278 2009-01-03
  JOGL 1.1.1+
  Java3D 1.5+
  Java3D VRML97 loader available
  Java3D 3DS loader available
  JRE 1.6.0_10-rc, JVM 11.0-b15
  ADMIN revision 36 2008-09-13
  RXTX available
$7 = #null

msim>picked trench-slide-1, class TREE_JOINTS, 0 in cycle of 1
manipulating trench-slide-1
  subtransform target
  in parent frame
  hit escape end manipulation

()
$8 = ()

msim>(cons-link "a-nice-new-link")
-U:**-  *scheme*      Bot (508,34)   (Inferior Scheme:run)----<I> Fr 2/13/09 10:52--
```

Figure 1-4: Screenshot of the mixed real/virtual interface.
At left, a Scheme read-eval-print loop serves as a textual command interpreter for detailed interaction. At right, a graphics window, in this case showing ATHLETE with added virtual articulations. General click-and-drag interaction is implemented for interacting with both links and joints. Virtual articulations can also be added/removed/reconfigured directly in the graphical view.

Though the implementation was crucial for validating the theory and in implementing the applications experiments, I do not focus on the details of the software in the thesis. Its architecture is a fairly direct implementation corresponding to the theoretical descriptions of the following aspects of the framework:

- the hierarchical linkage graph data structure and topological mutation algorithms (Sections 3.2.4 and 3.6)
- the $(\boldsymbol{t}, \boldsymbol{\theta})$ 3D rigid transform representation, with dynamic reparametrization and exp/log maps (Section 3.3.3 and Appendices B and C)
- the prioritized damped least squares solver (Algorithm 4.1, and Appendix F) for local assembly and differential control motions, including my design for six particular priority levels (Section 4.7)

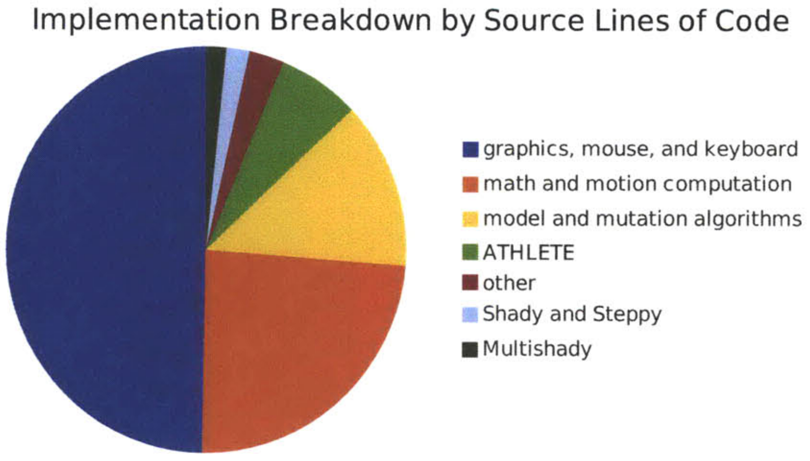


Figure 1-5: Breakdown of sourcecode in the implementation.
Breakdown by major function of the 43k source lines of code in the implementation. About 85% of the code is in Java, with the rest in Scheme. These metrics were generated using David A. Wheeler's "SLOCCount".

- the Jacobian and residual computation algorithms (Section 4.9 and Appendices G and H), though the implementation is optimized to exploit the typical sparsity of the various binary projection matrices, which are just a convenient way to express index bookkeeping, and are not represented directly in the code
- the simulation loop (Algorithm 4.12), effectively the "main loop" of the system.

A significant portion of the code deals with the many details of fast but attractive on-line rendering of potentially large models. Processing mouse gestures also involves some work: the mouse is a 2D device but in many cases it is used to manipulate spatial objects with up to 6-DoF poses.

1.5 Contributions

At the highest level, there are three main contributions in this thesis:

1. the algorithms and data structures of the general purpose framework in Chapters 3 and 4 for modeling and simulating mixed real/virtual articulated systems, which supports novel kinematic abstractions in both space and time, and which scales to topologically large systems while retaining interactive performance

2. the operations environment in Chapter 5 for high-DoF robots based on virtual articulations and structure abstractions, enabling rapid graphical specification of a broad class of coordinated motions which could be easy to visualize but difficult to specify in prior interfaces

3. the technique in Chapter 6 of using virtual articulations and sequence abstractions to form homogeneous models of compliant/proprioceptive locomotion, which can be qualitatively more concise than pure physical models but still capture the essence of the system and make quantitative predictions.

Individual contributions within these are summarized in the following subsections.

1.5.1 Contributions of the Framework

Though other related modeling frameworks have previously been reported [38, 89, 52, 33, 43, 130, 171, 18], mine addresses some particular challenges that become interesting in the mixed real/virtual case:

- *Genericity*—First, while real robots are severely limited by practical engineering considerations, with virtual articulations a larger variety of joint types and topologies become both possible and interesting. I introduce a particular parametrization for 3D joint mobility based on translation and orientation vectors and exponential and logarithmic maps thereof (Section 3.3.3). I prove the completeness (Theorem 1) of a natural partition of this parameter space, which I then use as the basis for a catalog of useful virtual joints, including all lower pairs except helical. Second, my framework supports arbitrary mixes of open and closed kinematic chains, allows both Cartesian and reduced coordinate modeling (Section 3.2.3), and smoothly handles both under- and over-constraint (Section 5.1).

- *Mutability*—To add, remove, and reconfigure virtual articulations on-line, I present algorithms implementing a comprehensive set of structure mutation primitives (Section 3.6). Such a complete and detailed set of algorithms for topological dynamics in a kinematic model is unusual: previous modeling frameworks typically dealt with structures that did not change topology on-line; the few frameworks that do deal with structure-varying (reconfigurable) robots [108, 130, 91] are typically limited to attach/detach operations, and do not cover more general topological evolution, including locking/unlocking joints, changing joint mobility type, etc.

- *Hierarchy*—While control authority over the actual joints of a robot is limited by the available actuators and the laws of physics, we are free to assign more arbitrary semantics to virtual articulations. One way my framework exploits this freedom is by defining a specific semantics for hierarchically encapsulated *sub-linkages* (Section 3.5). This can help organize topologically large models, and it also forms the basis for my technique of *structure abstraction* (Section 3.5.3).

- *Scalability*—As virtual articulations are added to a model, the total model size can increase quickly. Also, some key applications for virtual articulations and kinematic abstractions are driven by the challenges of actual robots which start out with relatively large numbers of joints. The motion computation algorithms in my framework thus scale to 100s of joints (Section 5.4) on commodity hardware while maintaining interactive response times in the range of 10s of milliseconds. Two key decomposition algorithms support this: *coupling decomposition* (Section 4.10) and *hierarchical decomposition* (Section 4.10.1).

In addition

- I take the unusual step of extending the core kinematic framework to also handle kinostatic models that involve gravitational and elastic potential energy. These are both necessary and sufficient for a broad class of interesting compliant mechanisms, but are simpler than the more common physical dynamics frameworks (e.g. [143]) that also include kinetic energy.

- I break the requisite motion computations into two particular problems: *local assembly* and *differential control*, and I introduce a novel set of six particular priority levels to solve these on-line under a task-priority framework based on iterative local linearization, Jacobian pseudoinverse, and nullspace projection.

1.5.2 Contributions of the Operations Interface Applications

To the best of my knowledge, the research presented in Chapter 5 is the first example in the literature of using virtual links and joints in a constructive graphical interface to operate high-DoF robots. Often an intended coordinated motion is easy for an operator to visualize, but expressing that intention could be frustrating and tedious in more traditional operations interfaces (Section 5.3 describes such a traditional interface for ATHLETE). Virtual articulations can graphically parametrize a wide variety of kinematic tasks and can also constrain coordinated whole-robot motion.

The experiments which demonstrate the viability of this approach involve two specific robots: ATHLETE and *Multishady.* While I was not involved in the development of the ATHLETE robot itself, Multishady (Section 5.4 and Appendix K), used in simulation to demonstrate the scaling potential of my framework up to 100s of joints, is a novel concept for a bi-partite truss-like self-reconfiguring robot. This scaling is enabled by my novel technique of *structure abstraction* (Section 3.5.3), which (1) helps the operator subdivide the problem in the same way that traditional algorithmic abstraction aids a software designer; and (2) enables the system to perform *hierarchical decomposition* (Section 4.10.1) to keep the costs of motion computation low, maintaining interactive response time.

1.5.3 Contributions of the Compliant Motion Applications

While the use of virtual joints to model *instantaneous* contacts and uncertainties has previously been proposed [19], I show that in some interesting cases, *sequence abstractions* (Section 6.3) can be valid high-level approximations on much longer timescales, even through various contact configurations. Though physics is technically sufficient it can be overly detailed. Virtual articulation models with sequence abstractions can be qualitatively useful to help "see the forest over the trees." And they can also make quantitative predictions—I introduce a general strategy for using them to predict the limits of tolerable uncertainty, and show that these can compare favorably to corresponding real-world tests on the actual hardware.

These ideas are again experimentally validated, and here, both of the systems in the experiments, *Shady* (Section 6.6 and Appendix J) and *Steppy* (Section 6.7 and Appendix L), are substantially novel. Shady is a scratch-built structure climbing robot that achieves experimentally demonstrated reliability using a combination of mechanical compliance and a proprioceptive control strategy. While the Steppy robot itself is an off-the-shelf mini humanoid, I developed and analyzed a novel compliant/proprioceptive strategy for reliably climbing a step of significantly uncertain height.

Chapter 2

Related Work

Virtual articulations, and to a lesser extent, kinematic abstractions, have been considered previously in the literature. This chapter reviews these earlier works and highlights the main differentiators of this thesis. Overall, my approach is both significantly more broad than prior reports of virtual articulations, which are usually quite specific or embedded within the context of a particular robot or task; and also substantially deeper in that I develop, implement, and experimentally validate a concrete set of algorithms for adding/removing/modifying virtual articulations and for interacting with the resulting mixed real/virtual model.

The few existing prior works which mention ideas similar to my kinematic abstractions are mainly conceptual—they do not typically present associated algorithms or experiments. My spatial abstractions are made concrete via a novel hierarchical decomposition algorithm and tested in simulated experiments, and my temporal abstractions are demonstrated to be both qualitatively useful and quantitatively predictive in hardware experiments with several specific robots.

2.1 Overview

The research context for this thesis as a whole splits into seven parts:

1. **previous uses of virtual articulations** reviewed next in Section 2.2

2. **earlier ideas related to kinematic abstraction** reviewed in Section 2.3

3. **other similar systems** including other kinematic modeling frameworks (Section 2.4), geometric constraint solvers (Section 2.5.2), and physical dynamics simulation (Section 2.5.1)

4. **existing methods for operating high-DoF robots** specific related work in this application area is deferred to Section 5.5

5. **other approaches to high-level modeling for compliant mechanisms** similarly deferred to Section 6.8

6. **earlier robots** related to the novel robotic systems I have developed: Shady (context in Section J.1), MultiShady (Section K.3), and Steppy (Section L.1)

7. **foundational works used as components** such as the prioritized damped least squares method I adopt from Baerlocher and Boulic [11] and Grassia's dynamic reparametrization for orientation vectors [55]. These and other foundational works are identified in the context of the appropriate chapters.

2.2 Virtual Articulations in Specific Applications

A literature search for prior uses of virtual articulations finds a handful of particular applications: virtual joints (virtual links are somewhat less common) are proposed for a specific robot or a specific task, or in some cases for a specific class of robots or tasks. I categorize these prior works in terms of the benefit conferred by the virtual articulations: resolving redundant motion, speeding inverse kinematics solution, maintaining Jacobian rank, modeling link compliances, and modeling geometric uncertainties.

Redundancy Resolution A few authors have previously explored the idea of constraining the kinematic motion of redundant robots by adding chain-closing virtual joints. Ivlev and Gräser [75, 76] focused mainly on defining virtual chain closures which help restrict the motion of redundant revolute serial-chains, for example to avoid known obstacles in the environment. They treated only smaller examples which could be solved analytically on paper. Bruyninckx [19] also showed one example where a virtual closed chain minimizes the otherwise unconstrained twist motion about the axis of a round tool on a revolute-jointed industrial manipulator. Bruyninckx's work further touches on a few other aspects of virtual articulation and even temporal kinematic abstraction, as detailed below and in Section 6.8.

Faster Inverse Kinematics for Long Chains While even one extra degree of mobility with respect to a task creates a kinematic redundancy, *hyper-redundant* robots [29] are loosely defined to have larger numbers (10s or more) of extra DoF. Often-studied systems in this field are long serial chains, typically of revolute joints. Inverse kinematic computation can become expensive in these cases due to the large number of joints, and some authors have previously observed that replacing individual sub-chains with virtual assemblies can serve to reduce the computational complexity. Ashrafiuon and Sanka [8] applied the idea to generate inverse kinematic control algorithms for specific 9- and 10-DoF spatial revolute serial chains. Williams and Mahew [168] similarly built an IK controller for a hybrid structure consisting of a serial chain of parallel mechanisms (this structure bears some resemblance to the tower experiment in Section 5.4). These ideas are related to my approach of structure abstraction (Section 3.5.3), but whereas my method is developed and implemented (via the hierarchical decomposition algorithm, Section 4.10.1) for the general case, these prior reports are specialized treatments applied by hand for specific robots. The virtual joints are fixed off-line and only to help control algorithmic complexity of inverse kinematic solution; they are invisible to the operator. My method of abstraction can be useful both to the operator, to structure the design of the motion, and to the system, by decoupling the inverse kinematics computation into smaller sub-problems.

Maintaining Jacobian Rank at Singular Configurations Oetomo, Ang, and Lim [115] propose inserting virtual joints to virtually relieve the instantaneous loss of mobility which can occur at a singular configuration. Constructions like this are also possible in my system, though I have not explored this particular application.

Modeling Link Compliances In [1], Abele, Rothenbucher, and Weigold also insert virtual joints, in this case to model compliances which in flexible links. This could also be done in my system, though in the examples in Chapter 6 I have instead focused on compliances that are co-located with actual joints, which are of much greater magnitude than link flexibility in the robots which I consider.

Modeling Geometric Uncertainties Bruyninckx [19] uses virtual joints to model some kinds of geometric uncertainty in compliant motion tasks, but focuses on smaller perturbation-like uncertainties vs. gross uncertainties like the step height in my humanoid stair-stepping experiment (Section 6.7, also see Section 6.8).

2.3 Prior Work in Kinematic Abstraction

Searching for prior reports of kinematic abstraction yields only a few works related to my spatial structure abstractions and temporal sequence abstractions.

Abstraction in Space Davis [36] explored various forms of spatial geometric abstraction at a conceptual level, including an interesting but very cursory (one paragraph) mention of the idea of a "kinematic device as black box." Zanganeh and Angeles [176] present a more detailed and formal development of the idea of separating topologically large kinematic structures into sub-mechanisms, but their approach does not specifically separate interface from implementation—it just demarcates sub-mechanisms. My method of structure abstraction subsumes this capability ("simultaneous" sub-linkages, Section 3.5.1), but also allows separate interface mechanisms which stand-in for an underlying implementation (Section 3.5.3).

Abstraction in Time It appears that few other authors have considered kinematic abstractions in the temporal domain, as I do in my method of sequence abstraction 6.3, though the idea of using kinematic models for contact was originally popularized by Salisbury [99]. Bruyninckx [19] models contact using virtual joints, but mainly addresses instantaneous contact configurations. Whereas such models would need to be reconfigured as contacts are made and broken (or possibly even after any motion), I show in Chapter 6 that in some cases sequence abstractions can persist through changes in contact configuration and contact motions.

2.4 Prior Work in Kinematic Modeling

Chapters 3 and 4 develop a new kinostatic modeling framework that has particular features to help support mixed real/virtual models. This section reviews prior frameworks, all of which have some drawbacks for the purposes of this work. Overall, my approach is more generic, mutable, scalable, and hierarchical—Section 3.1 expands on these issues and the ways in which I address them.

In the 1950s, Denavit and Hartenberg [38] introduced a kinematic modeling framework which is still widely used. It supports revolute, prismatic, and helical joints (common uses are restricted to the first two only, cf. [145]) and both open and closed chains, but it falls short for more general hybrid open/closed (i.e. mixed serial/parallel) structures [81], and it cannot represent a true spherical joint without gimbal lock. The overall scope of the DH and related approaches is significantly narrower than my framework—on-line topological mutation is not addressed, there is no included support for any kind of topological decomposition, no statics parameters are included (mass/stiffness/gravity), and motion computation algorithms are not considered.

Some more modern and comprehensive frameworks include Diaz-Calderon, Nesnas et al's CLARAty Mechanism Model [43]; the OROCOS Kinematics and Dynamics Library of Bruyninckx et al [18]; the kinematics components in Pryor, Taylor, Kapoor, and Tesar's OSCAR [130]; and The Mathworks' SimMechanics [171]. The first three of these are mainly for open-chain mechanisms only, with some small exceptions. CLARAty Mechanism Model supports four- and six-bar planar closed chains, but not more general hybrid topologies. In [130] OSCAR is demonstrated on a closed-chain modular robot, but it appears this involved additional code to handle chain closures and module connect/disconnect operations. Other than this one example, the first three frameworks neither seem to support on-line topological structure changes.

SimMechanics is more general, and appears to overcome many of the drawbacks of the other frameworks. However it does not include topological decomposition algorithms or true kinematic abstractions that separate interface from implementation.

Flückiger [52] reported a novel "virtual reality" interface for kinematic operation of articulated robots, including serial, parallel, and hybrid topologies. Topological modifications apparently required his system to be re-started, and he did not report any use of true virtual articulations—the real robot links and joints were modeled alone, and then manipulated in a virtual graphical environment using various kinds of 3D user interface hardware.

2.5 Other Similar Systems

Many of the above environments also offer physical (Newtonian) dynamics modeling and simulation, in addition to pure kinematics. This motivates a question: Why not just use a real-time physics simulation package, such as the Open Dynamics Engine (ODE) [143]? In fact, there is also a symmetric question at the other end of the kinematics vs. physics modeling spectrum: Why not just use an interactive geometric constraint solver, such as in a modern computer aided design (CAD) program like SolidWorks (a product of Dassault Systèmes SolidWorks Corporation)? Both approaches seem to have some feasibility, but each would lack particular desirable features as called out in the next two sub-sections.

2.5.1 Physical Dynamics Simulation

Overall, physics based approaches generally require more parameter tuning (e.g. inertia tensors, stiffness matrices, friction and damping coefficients) than simpler kinematic or even kinostatic models. This would potentially add additional burden on the user as virtual elements are added, removed, and reconfigured. And, while algorithms for efficiently simulating the physical dynamics of reduced coordinate models (cf. Section 3.2.3) and closed kinematic chains [50] are known in the literature, currently available real-time physics engines (including ODE) are typically based on Cartesian coordinate models (called "maximal coordinate methods" in [84]) and need to be carefully tuned to efficiently and stably simulate kinematic chains, especially in the presence of chain-closing joints [15].

Of course, if full physics simulation is performed, then one could also consider adding virtual elements that are not strictly kinematic; for example, virtual springs and dampers. This has been explored by Pratt et al [128, 127] as *virtual model control.* In their reports, the virtual elements (typically springs and dampers; virtual joints and links are not as prominent) are manually designed, tuned, and implemented within custom control code on a case-by-case basis for each task of each mechanism.

Sticking to a kinematic model, with minimal (and optional) extensions for quasistatics, enables topologically large-scale constructions with many closed chains, real-time motion computation on commodity hardware even for systems with 100s of joints, on-line structure changes[1], and kinematic abstractions in space and time.

2.5.2 Geometric Constraint Solving

Geometric constraint solvers [89, 17, 67], which automatically solve for a spatial configuration of a collection of geometric objects satisfying a given set of constraints, have long been studied—even Sutherland's first report of a computer drawing program in [152] included constraints. Current implementations in mechanical CAD systems, such as D-Cubed's Dimensional Constraint Manager [117, 66], support 3D "mating" relationships: concentric surfaces of revolution, incident faces and edges, etc.

In this design domain, the canonical problem [89] is typically to find a well-constrained rigid configuration. However, some systems allow under-constrained assemblies (over-constraint is usually still forbidden) which can model articulated robots, using mates as proxies for some kinds of kinematic joints. For example, a com-

[1]Lampariello, Abiko, and Hirzinger [91] and Nakamura and Yamane [108] have considered structure changes in the context of physical dynamics simulation.

bination of a concentric and a perpendicular incident-face constraint could correspond to a revolute joint. The designer can then graphically interact with the model and the system maintains all specified constraints. Open- and closed-chain mechanisms can be constructed. However, the origin of the constraint solving problem, focused on the well-constrained rigid case, has led to less-than-desirable methods for handling under-constrained systems. Commonly, hidden constraints are inferred (e.g. [78]) to convert the under-constrained problem into a well-constrained one. These inferences are based on heuristics, and can be surprising.

In an informal test, I constructed a model of ATHLETE in a recent version of SolidWorks and attempted a bi-manual operations example similar to the one shown in Figure 5-8. The result was very hard to control, apparently due to hidden constraints which apparently locked certain joints in order to fully constrain the model. It was also prone to non-continuous motion (jumping from one connected component of the configuration space to another).

Using geometric constraints as a tool for defining and constraining motion has also been proposed in the field of articulated figure animation (e.g. Phillips, Zhao, and Badler [123]; Welman [163]; and Baerlocher [9]). In these works typically only a few simple constraints are available, for example, point-on-plane and point-on-line. The constraints are not themselves considered to be kinematic joints, and can only be added between geometric objects situated within links of the original articulated model. In my framework the operator is free to construct arbitrary additions including both virtual links and a relatively large catalog of virtual joints.

2.6 Additional Categories of Related Work

This chapter has covered the first three of the seven categories of related work listed in Section 2.1: previous uses of virtual articulations; earlier reports of kinematic abstraction; and other similar systems, including kinematic modeling frameworks, physics-based approaches, and geometric constraint solvers. The remaining categories are covered later, in the specific sections called out in that list.

Chapter 3

Model Structure and Topological Interaction Algorithms

This chapter presents the topological and structural aspects of my framework for modeling articulated robots combined with virtual elements. These aspects enable (1) building models of articulated robots and (2) dynamically adding and removing virtual articulations and kinematic abstractions. This is the first half of a full description of the framework; the other half, the *kinetics* of how models move, is the topic of Chapter 4. Also, the model developed in this chapter is purely kinematic—it does not incorporate any representation of physical energy. Chapter 4 will add extensions for elastic and gravitational potential energy, enabling the representation of a broad class of kinostatic systems.

These chapters constitute the main theoretical foundation of this thesis. All of this theory has been implemented and used in a complete software system to produce the results in Chapters 5 and 6. Figure 3-1 shows the architecture of this implementation, the *mixed real/virtual interface*, with the aspects covered in this chapter highlighted. Together, this chapter and Chapter 4 present and analyze all the important data structures and algorithms in the figure.

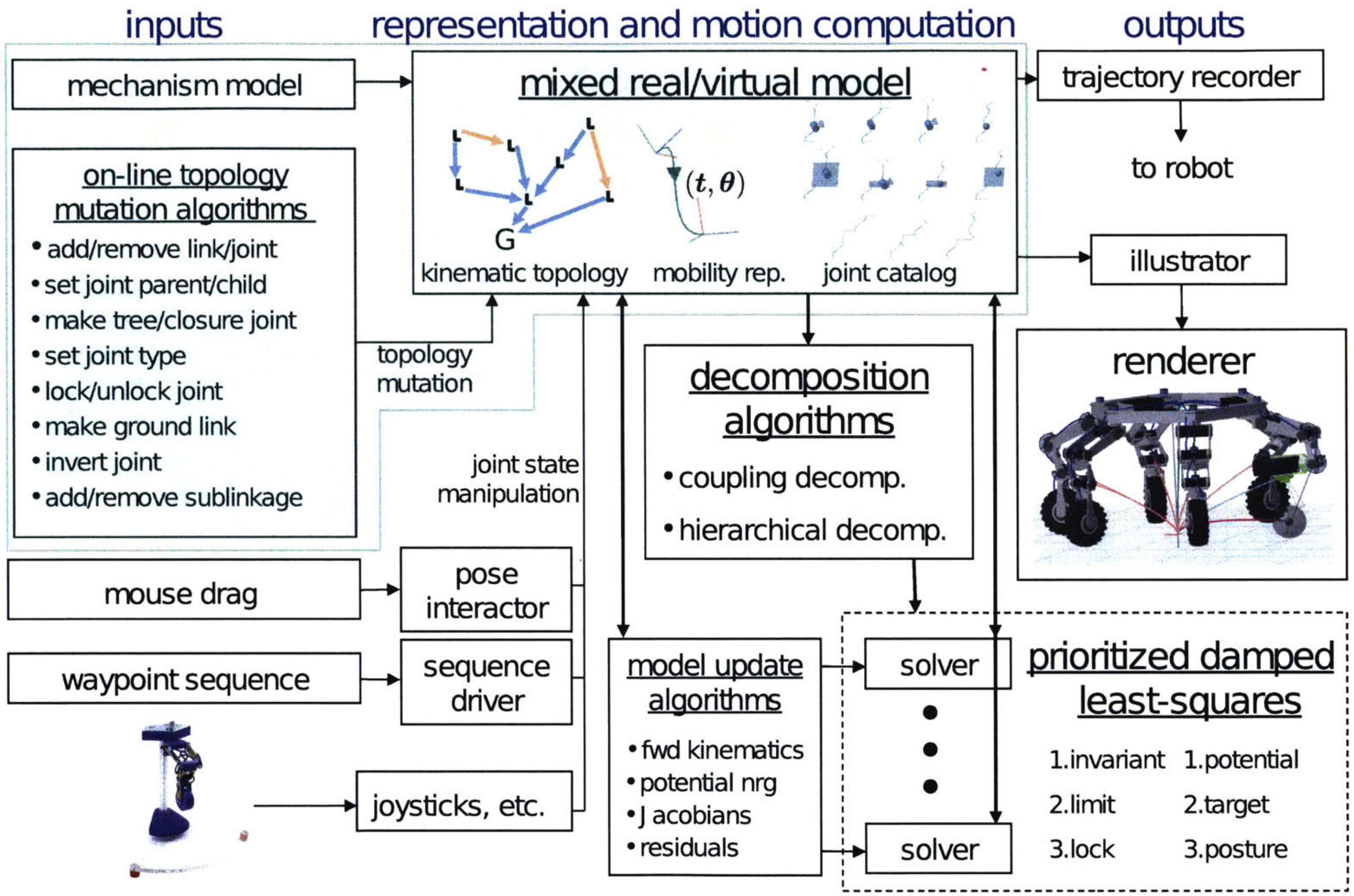


Figure 3-1: Structural and topological aspects of the implementation.

The next section summarizes my approaches to the unique challenges of mixed real/virtual modeling. The framework is then presented over the remaining four sections, starting with Section 3.2 which describes the highest level of the representation, the *kinematic graph* with vertices corresponding to the model's links[1] and edges corresponding to the joints.

I leverage a particular exponential map *mobility representation* to support a useful variety of joint types; this is explained next in Section 3.3. The development culminates in a new completeness proof for a partitioning of the space of 3D rigid body motion, upon which I build a useful *joint catalog.* .

A key innovation I introduce in this thesis is the capability to *hierarchically subdivide* large models. Section 3.5 extends the framework to support this, and introduces the technique of *structure abstraction* which can hide complexity in the operation of very high-DoF robots by enabling complex kinematic sub-systems to be virtually replaced with simpler abstractions.

Finally, Section 3.6 presents a complete set of *structure mutation primitives* that can be invoked to (1) build up a model of any articulated robot, and (2) dynamically add and remove virtual articulations and kinematic abstractions. These are important because they constitute a concrete set of tools available to build models and to add and remove virtual articulations. The presentation of such a set of operations is uncommon since this level of on-line structural mutation is not typically necessary for traditional robot modeling without dynamically added virtual elements.

3.1 Challenges of Mixed Real/Virtual Modeling

The main idea of virtual articulation and kinematic abstraction is to *homogeneously* model a variety of phenomena using the same joint and link metaphors as are used for the robot itself. Thus my framework is related to prior kinematic and kinostatic robot modeling techniques, e.g. [38, 89, 52, 33, 43, 130, 171, 18]. However, the need to conveniently support virtual elements calls to focus some aspects which are not

[1]I use the term "link" to refer to the rigid components (solid bodies) in the model.

necessarily important when modeling robots alone:

- *Genericity*: For practical reasons—load bearing, mechanical tolerance, manufacturability, mechanical complexity, etc.—actual articulated robots are often special cases, and prior frameworks are often tailored to these cases. Common restrictions include limitation to only revolute joints, and/or only simple chain or tree topologies. The practical concerns that underlie these restrictions do not necessarily apply to virtual articulations, so my framework handles an unusually general class of articulated systems, with a broad set of joint types and no topological restrictions.

- *Mutability*: In practice most articulated robots have a fixed kinematic structure, so traditional frameworks usually do not include significant support for on-line structure changes, though a few can model mechanical attach/detach operations [108, 130, 91]. In contrast, many uses of virtual articulations and kinematic abstractions rely directly on the ability to evolve the model's kinematic topology, and because these changes are virtual they can easily be more intricate than just attach/detach. I thus take the unusual step of presenting a rich and complete set of operations for evolving the structure of the model. Certain design choices I make in the representation, such as *root joints* (Section 3.2.3), specifically aid these operations.

- *Scalability*: For reasons of both practicality and cost, common articulated robots have relatively few links and joints. For example, articulated manipulators for manufacturing and factory automation rarely have more than 6 joints. Humanoids and hyper-redundant robots can have 10s of joints, but usually not more, and are still relatively rare. Thus, scaling to hundreds of articulations or more is not a particular focus of prior frameworks (Pryor et al's work in [130] is an exception here also, as is Redon and Lin's work in [132]). Again, virtual articulations do not necessarily have the same kinds of penalties associated with them, so it can be meaningful and reasonable to add them in relatively large numbers (e.g. 100s), and my framework supports this.

- *Hierarchy*: A key type of kinematic abstraction I explore is the notion of virtually replacing a complex substructure with something simpler. Thus, my representation can partition the whole model into sub-models, and can make virtual substitutions thereto. To the best of my knowledge, no prior frameworks directly support this. To ease the description, I initially skip the hierarchical features, then add them in Section 3.5.

Finally, there are a few modeling aspects on which I do *not* focus; these could all be added as extensions in future work:

- *kinetic energy*: This chapter first presents a purely kinematic framework, with no modeling of physical energy. Chapter 4 will add extensions to also model gravitational and elastic potential energy, which are needed for the proprioceptive modeling applications in Chapter 6. I have not yet pursued further extension to also model physical kinetic energy, for several reasons. One is that general-purpose physics simulation systems (e.g. [143]) are now commonly available and well studied. While many can model kinematic joints, they are often primarily focused on rigid-body models with impacts rather than continuous kinematic constraint, and their performance especially while handling large numbers of closed kinematic chains can be poor [15]. In part because kinematic chain closing is essential in using virtual articulations to constrain motion, I have focused on my own simulation algorithms (Chapter 4), which are designed expressly to handle closed chains. Another reason I have not yet considered kinetic energy is simply that it has not been required for any of the applications I have studied.

- *collision detection*: Many prior frameworks, especially in physics simulation, include detailed models of the geometry of the bodies which form the articulated mechanism, and compute collisions among them. This is a well-studied problem. Also, while collision is always a consideration for models of real robots, virtual elements can legitimately have no associated geometry, and thus no possibility of collision. For these reasons I choose not to focus on collision, with the exception of the high-level models of contact state evolution developed in Section 6.3.

- *distinguishing virtual elements*: It is not hard to distinguish real from virtual model elements, e.g. by setting a flag. One use for such a distinction could be to prioritize actual over virtual constraint in inconsistently overconstrained cases, but I neither focus on this. Again, the main idea is to homogeneously handle both the real and virtual parts of the model.

3.2 Linkages and The Kinematic Graph

An articulated system in my representation is called a *linkage*[2]. In this section the high-level topological structure of a linkage will first be introduced informally, followed by a concise formal definition on page 53.

The binary connectivity of joints and the arbitrary connectivity of links naturally suggest representing the topology of a linkage as a graph, as in Figure 3-2, where the links are vertices and the joints are edges. Such graphs have been used at least since the 1960s [44], but the graph itself does not define the full structure of the system.

One large gap, specification of the range of motion for each joint, is covered later in Section 3.3.

Another gap is that any restrictions on allowable graph topologies must be made explicit. For example, should cycles be allowed? What about disconnected graphs? My answers to these particular questions are yes and no, respectively. Cycles are crucial because it is precisely the mobility restriction capability of closed kinematic chains that makes virtual articulations a useful tool for designing constrained motion (recall Figure 1-3). And, though on first consideration it might seem that disconnected graphs would be the more general case, the homogenizing power of virtual articulations is easily leveraged to ensure that the kinematic topology always remains connected, with special un-constrained virtual joints automatically added and removed as necessary to maintain connectivity. I describe my design for these *root*

[2]Some authors use this term more specifically, for example to refer only to articulated systems whose mobility space is strictly 1-dimensional [73] or to systems with only rotating joints [37].

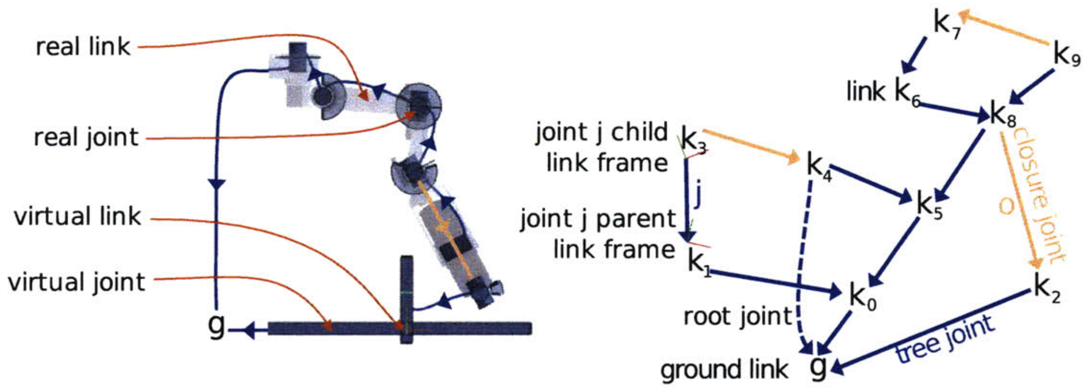


Figure 3-2: Linkage graph examples.
At left, a mixed real/virtual model with superimposed linkage graph. At right, a second linkage graph. *Links*—rigid parts of the model—form the vertices, and *joints*—moving inter-link attachments—the edges of the graph. Such graphs have long been used in kinematics and robotics; I make some particular design choices, including the ever-presence of a spanning tree (dark edges) with the remaining *closure* joints (light edges) closing kinematic chains. Edge direction gives the forward sense of a corresponding *joint transform* that takes coordinates in a joint's child *link frame* to the link frame of its parent. Special un-constrained virtual *root joints* (one shown dashed, others hidden) connect each link directly to the *ground link* at the root of the spanning tree.

joints below in Section 3.2.3; one main benefit of maintaining connectivity is that *joint transforms*, associated to each edge in the graph, are always sufficient to compute the pose of any link with respect to any other.

3.2.1 Spanning Tree Topology

When the kinematic graph is cyclic there may be multiple paths between a given pair of links; a main design feature of my linkage representation is that the relative pose of any two links can always be uniquely defined by the path between them consisting only of joints in an identified spanning tree. This tree also induces a natural directivity on the edges it contains: in my design such edges point towards the root of the tree, and I also use (the same) edge direction to give the forward sense of the associated joint transform. These properties will be used frequently, so some notation and terminology is helpful (also see Figure 3-2).

Definition 1 Each joint (graph edge) j is directed so that it connects a pair of links (p_j, c_j) called the *parent and child link* of j. The edge direction is from c_j to p_j.

Definition 2 Each link k (graph vertex) has an associated coordinate frame F_k called the *link frame* of k. The link frame of the child (resp. parent) link of a joint j is called the joint's *child* (resp. *parent*) *frame* F_{c_j} (resp. F_{p_j}).

Definition 3 For each joint j there is an associated rigid-body *joint transform* X_j that defines the pose of the child frame with respect to the parent frame. X_j is one aspect of the mobility representation O_j for j, which will be detailed in Section 3.3 (cf. Eq. 3.9).

Definition 4 Exactly one of the links is identified as the *ground* link g.

Definition 5 Every link $k \neq g$ has exactly one of its outgoing joints (i.e. a joint whose child is k) identified as its *parent joint* p_k, and g has no parent joint. To complete the terminology, all of the incoming joints of a link k (i.e. joints whose parent is k) are called *child joints* of k.

Definition 6 The set T of all parent joints of all links must form a *directed spanning tree* with root g; i.e., all edges in T point towards g. Thus, a transform[3] $X_{k_0 \leftarrow k_1}$ giving the pose of any link frame F_{k_1} relative to any other link frame F_{k_0} is defined by composing the joint transforms along the path $g \leftarrow k_1$ followed by the inverse of the composition of the transforms on the path $g \leftarrow k_0$ (transforms compose right to left, and the products are taken in path order):

$$X_{k_0 \leftarrow k_1} = \left(\prod_{\imath \in g \leftarrow k_0} X_\imath\right)^{-1} \left(\prod_{\imath \in g \leftarrow k_1} X_\imath\right) = \left(\prod_{\imath \in k_0 \leftarrow g} X_\imath^{-1}\right) \left(\prod_{\imath \in g \leftarrow k_1} X_\imath\right). \qquad (3.1)$$

Definition 7 If J is the set of all joints in the linkage, then the remaining joints $C = J \setminus T$ are *chain closures.* The membership of a joint in T is its *disposition.*

[3] $X_{k_0 \leftarrow k_1}$ takes coordinates in F_{k_1} to coordinates in F_{k_0}.

3.2.2 Names and Paths

Besides ensuring that the relative pose of any pair of links is never ambiguous, another benefit of the spanning tree is that it enables a straightforward way to avoid ambiguity when identifying entities in the user interface.

Let each link k and each joint j have a textual name n_k resp. n_j. One approach to avoid name collision would simply be to require that all names are globally unique, but this can become tedious for models that contain repeated substructures. Instead we can leverage the tree path of the entity to reduce the possibility of collision.

Definition 8 The existence of the spanning tree means that there is always a *path* $n_g.\cdots.n_i$ consisting of the names from the ground link g to any link or joint i.

Paths are always unique as long as no two child joints of any link have the same name—a less stringent requirement than global uniqueness.

3.2.3 Root Joints

Always requiring a spanning tree may seem a strong constraint on allowable model topology. What if the system to be modeled really is disconnected? Or, what if the modeler does not wish to distinguish some joint in a kinematic cycle as a closure? These issues are both solved in a relatively straightforward way by leveraging the homogenizing power of virtual articulations.

Definition 9 In my framework there is a special unconstrained (i.e. allowing any 3D rigid motion) virtual *root joint* connecting each link directly to the ground link. Root joints are automatically created whenever a link is added, and deleted whenever the associated link is removed. They may either be tree or closure joints; in the former case the attached link is a free-floating *root link.* Figure 3-2 shows an example .

Though root joints are always present for every link, in most figures (and in the interactive software) they are typically hidden from view.

Root joints aid in managing the life cycle of a link by providing a canonical joint to which any link can return if it loses its parent joint. The connectivity requirement

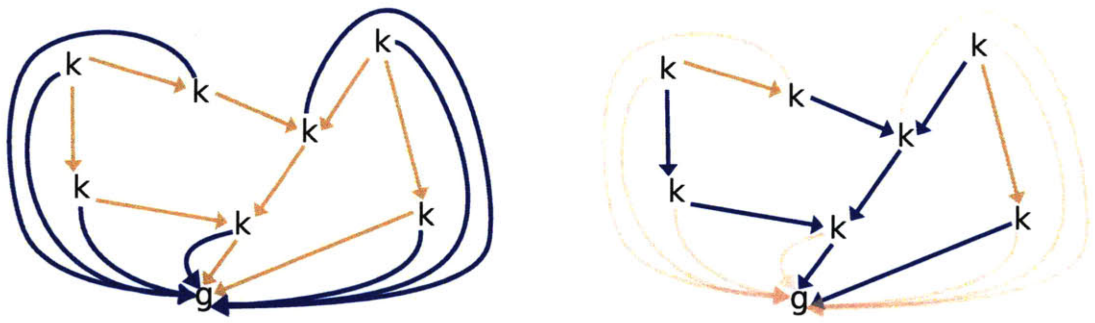


Figure 3-3: Cartesian vs. reduced coordinate modeling.
Left: using root joints to emulate Cartesian coordinate modeling. Each root joint has tree disposition (dark arrows) and sets the pose of the connected link directly with respect to the ground link. All other joints are closures (light arrows). Right: the same linkage topology in a reduced coordinate model. Root joints are still present connecting each link to the ground link even in the reduced coordinate model; however, they are all unconstrained 6DoF closure joints, and to reduce clutter such unconstrained root closures are normally not rendered.

is thus easily maintained, and in this sense root joints play a role as data structure sentinels in the mutation operations given in Section 3.6.

Root joints can also be used to build *Cartesian coordinate* models instead of the *reduced coordinate* models implied by deeper kinematic spanning trees. As shown in Figure 3-3, in a fully Cartesian coordinate model all state is encapsulated in the root joints, which are all tree joints and thus set the poses of the attached links; in a fully reduced coordinate model all state is contained in the spanning tree joints, and the tree may have depth greater than one [171][4]. It is thus possible to model a kinematic cycle in my framework without making any particular joint "special"—instead *all* joints in the cycle are closures.

More generally, root joints enable the user to choose either Cartesian or reduced coordinates for different parts of the model, while always maintaining the required kinematic spanning tree. Each technique has advantages and disadvantages, see e.g. [171] for details. Reduced coordinate models are generally appropriate when the robot itself is tree structured, as are many articulated robots in practice.

[4]The wording "absolute" vs "relative" is used in this reference instead of "Cartesian" vs "reduced"; in the field of physical dynamics the corresponding terminology is "maximal" vs "generalized."

Root joints also serve some roles in homogenizing non-topological interaction. Click-and-drag manipulation of a link is easily reduced to manipulation of the pose of its associated root joint, so the interaction code only needs to handle the case of manipulating joints, even when the operator is actually dragging a link. Also, it is sometimes desirable to *lock* the spatial pose of a link, and this can again be accomplished by locking its root joint. The algorithms that implement manipulation and locking are described in Chapter 4.

3.2.4 Formal Linkage Definition

The main topological aspects of (and constraints on) a linkage have now been covered, and much terminology has been introduced that will be used throughout the rest of the thesis. Definition 10, below, collects all of this information into a concise and complete reference. In particular, many of the symbols used here are also used in the algorithm pseudocode in Section 3.6 and in Chapter 4.

This definition is essentially a summary of what has already been stated, though there is additional structure within O_j, detailed below in Section 3.3. Also, three new components, p_L, d_L, and Υ_j, are included as placeholders for hierarchical linkages and potential energy modeling. These features will be explained in later sections; all are optional and are disabled by setting the placeholder to $\emptyset$.

Definition 10 A *linkage* is a structured graph represented as a 5-tuple

$$L = (K_L, J_L, g_L, p_L, d_L) \tag{3.2}$$

with K_L, J_L the sets of links and joints

$g_L \in K_L$ the ground link,

p_L the parent (enclosing) linkage, if any, else $\emptyset$ (Section 3.5)

d_L the linkage disposition, if any, else $\emptyset$ (Section 3.5) with each joint $j \in J_L$ a 5-tuple

$$j = (p_j, c_j, O_j, \Upsilon_j, n_j) \tag{3.3}$$

with $p_j, c_j \in K_L$ parent and child links,

mobility representation O_j (Section 3.3),

potential energy parameters Υ_j, if any, else $\emptyset$ (Section 4.6),

n_j a name string s.t. $p_{j0} = p_{j1 \neq j0} \Rightarrow n_{j0} \neq n_{j1}$ (sibling joint names unique),

and each link $k \in K_L$ a triple

$$k = (p_k, r_k, n_k) \tag{3.4}$$

with $p_{k \neq g_L}, r_{k \neq g_L} \in J_L$, $p_{g_L} = r_{g_L} = \emptyset$ parent and root joints (none for ground link),

$p_{r_k} = g_L$, $c_{r_k} = k$ (root joint connects non-ground link to ground),

$j = p_k \Rightarrow k = c_j$ (parent-child referential consistency),

n_k a name string,

and the edges of a directed spanning tree of L rooted at g are

$$T = \bigcup_{k \in K} \{p_k\}. \tag{3.5}$$

3.3 Joint Mobility

This section will fill in the details of O_j, the *mobility* of joint j, a pivotal aspect of the framework. O_j identifies both the current joint transform X_j, which sets the 3D pose of the joint's child frame with respect to its parent frame, and the *joint space* $\mathcal{J}_j$: the space of all relative poses that j will permit between its child and parent frames. Nominally, $X_j \in \mathcal{J}_j$, though it is also possible for a joint to be *broken*, as described below.

At the lowest level, both X_j and $\mathcal{J}_j$ will depend on particular parametrization of the space of 3D rigid-body motions (*displacements*) via a translation and orientation vector pair $(\boldsymbol{t}, \boldsymbol{\theta})$. Section 3.3.3 introduces this parametrization; it will not be required before then.

The elements of the representation for $\mathcal{J}_j$ will first be developed informally, and then collected formally with X_j to construct O_j in Definition 16.

3.3.1 Components of the Joint Space Representation

One contribution in this thesis is the definition in Section 3.3.4 of a *catalog* of joint *types* Y s.t.

$$Y \subseteq SE(3), \tag{3.6}$$

with $SE(3)$ the Special Euclidean group of all 3D rigid body displacements

which define *canonical mobility spaces* for different kinds of joints. For example, there will be an entry in the catalog for revolute joints, another entry for prismatic joints, etc. To keep the catalog small (and finite)—in particular, to avoid requiring a separate type for e.g. each pose of the rotational axis of a revolute joint, or separate types for joints that differ only in their motion limits—I add three levels of indirection: *mobility limits*, an *inversion flag*, and *positioning transforms.*

The limits I_j of a joint j will be defined in Section 3.3.5 to trim the canonical mobility space Y_j of j to produce the actual mobility space.

Definition 11 The *actual mobility space*, or just the *mobility space*, $\mathcal{M}_j$ of a joint j is a restriction of the canonical mobility space—identified by the joint type Y_j—to the joint limits I_j:

$$\mathcal{M}_j = Y_j|_{I_j}. \tag{3.7}$$

And, just as $X_j \in \mathcal{J}_j$ for an unbroken joint, the mobility space nominally contains the mobility transform M_j, the factor of X_j which varies with the joint's pose.

Definition 12 The *mobility transform* M_j of j is the part of the pose of the joint (technically, M_j is a factor of X_j, as in Eq. 3.9) which varies when the joint moves. j is *unbroken* iff $M_j \in \mathcal{M}_j$, otherwise j is *broken.*

For example, the mobility transform for a revolute joint would normally give its current axial rotation.

To see why any joint would ever need to be broken, refer to Figure 3-2 (this is a classic observation in kinematics). Without the labelled closure joint o, the pose $X_{k_2 \leftarrow k_8}$ of link frame F_{k_8} with respect to F_{k_2} is unambiguously determined by the connecting chains of tree joints ($g \leftarrow k_8$, $g \leftarrow k_2$), and Eq. 3.1. But with the closure joint in the picture, it must also hold that $X_o = X_{k_2 \leftarrow k_8}$.

There are a few reasonable ways to handle this situation; my design choice is that Eq. 3.1 always takes priority, and thus the closure joint transform X_o is always defined by the tree joints that support links k_8 and k_2. Even if all of those tree joints are unbroken, it is entirely possible that the resulting X_o will break the closure joint o. Thus, in my representation the closure joints—and only the closure joints—may be broken.

The mobility transform normally takes coordinates from child to parent frame. However, it is useful to support the *inversion* of a joint, where this relationship is reversed, and for that purpose I include an additional flag.

Definition 13 The mobility *inversion flag*

$$\phi_j \in \{+1, -1\}. \tag{3.8}$$

determines whether M_j or its inverse defines the pose of the joint; cf. Eq. 3.9.

Such a flag is not the only way to implement inversion, but it does simplify some operations. In particular, joint inversion via flipping ϕ_j does not require any change to $\mathcal{M}_j$: whether or not j is inverted, it is always the case that $M_j \in \mathcal{M}_j$ for an unbroken joint. The full details of joint inversion are given in Algorithm 3.5 on page 90.

The variable mobility transform M_j combines with two constant transforms to produce the joint transform X_j, and the set of all X_j produced from $M_j \in \mathcal{M}_j$ forms the joint space $\mathcal{J}_j$. The formal definitions are as follows:

Definition 14 The mobility space *positioning transforms* C_j and P_j are arbitrary rigid body transforms that give constant offsets for the child and parent frames of j, F_{c_j} and F_{p_j}, respectively, relative to the *child and parent mobility frames* F_{cm_j} and F_{pm_j} of j.

The *joint transform* X_j for joint j (Definition 3), is the serial composition of the positioning transforms, interposed by the mobility transform, as shown in Figure 3-4:

$$X_j = P_j M_j^{\phi_j} C_j. \tag{3.9}$$

Specifically, C_j takes coordinates in F_{c_j} to F_{cm_j}, $M_j^{\phi_j}$ takes coordinates in F_{cm_j} to F_{pm_j}, and P_j takes coordinates in F_{pm_j} to F_{p_j}.

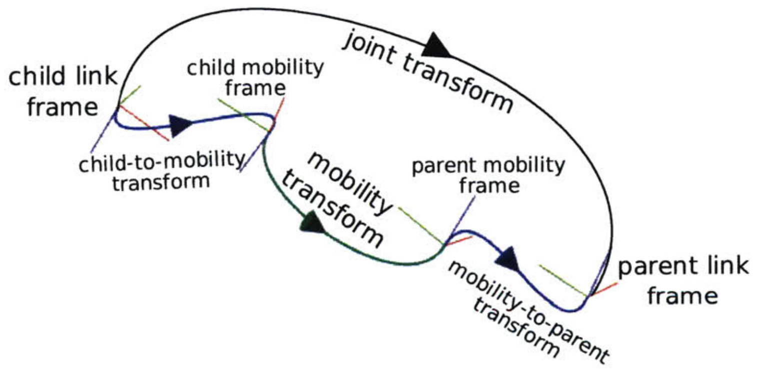


Figure 3-4: The mobility space positioning transforms.
This figure shows the detailed breakdown of sub-transforms and coordinate frames associated with a single joint. The overall joint transform is the product of a chain of three sub-transforms (Eq. 3.9). The outer two are *positioning transforms* which pose the inner *mobility transform* in space. The mobility transform represents actual joint motion, for example, the rotation of a revolute joint.

Definition 15 The *joint space* $\mathcal{J}_j$ of a joint j is a subspace of $SE(3)$ induced by the mobility space $\mathcal{M}_j$ and the positioning transforms:

$$\mathcal{J}_j = \{X_j = P_j M_j^{\phi_j} C_j \mid M_j \in \mathcal{M}_j\} \subseteq SE(3). \tag{3.10}$$

Note that $X_j \in \mathcal{J}_j \iff M_j \in \mathcal{M}_j$.

3.3.2 Formal Mobility Definition

Definition 16 Formally, the *mobility representation* O_j of joint j is a 6-tuple

$$O_j = (M_j, Y_j, I_j, \phi_j, P_j, C_j) \tag{3.11}$$

with mobility transform $M_j \in Y_j|_{I_j}$ if j is a tree joint, else $M_j = \emptyset$ (see below)

Y_j the joint type and I_j the joint limits,

$\phi_j \in \{+1, -1\}$ the mobility inversion flag,

P_j, C_j the positioning transforms.

The reason the mobility transform for a closure joint j is not explicitly represented ($M_j = \emptyset$ if j is a closure joint) is that it is always implied by the mobility transforms of *supporting* tree joints.

Definition 17 The *support* S_j of j is the minimal sequence of joints from the parent link p_j to the child link c_j. S_j is partitioned into into the *downchain* $S\downarrow_j$ of joints from p_j to the *least common ancestor* link[5] $\textsc{lca}(p_j, c_j)$ and the *upchain* $S\uparrow_j$ from $\textsc{lca}(p_j, c_j)$ to c_j:

$$S_j = (S\downarrow_j, S\uparrow_j). \tag{3.12}$$

The mobility transform for a closure joint is computed from the mobility transforms

[5]I.e. the topologically nearest link in L that is an ancestor of both p_j and c_j.

of the supporting tree joints based on Eq. 3.1 and 3.9:

$$\text{for a closure joint } j \text{ with support } S_j = (S{\downarrow}_j, S{\uparrow}_j)$$

$$M_j = \left[C_j \left(\prod_{i \in S\uparrow_j} X_i^{-1} \right) \left(\prod_{i \in S\downarrow_j} X_i \right) P_j \right]^{-\phi_j} \tag{3.13}$$

with the products taken in chain order.

3.3.3 $(\boldsymbol{t}, \boldsymbol{\theta})$ Parametrization of $SE(3)$

The development so far has treated rigid body transforms as black-boxes. In this section the box is opened and I describe my representation for transforms, which is based on separate vectors $\boldsymbol{t}$ and $\boldsymbol{\theta}$ for translation and rotation:

$$X \in SE(3) = (\boldsymbol{t} \in \mathbb{R}^3, \boldsymbol{\theta} \in \bar{B}^{\pi}(3)) \tag{3.14}$$

$$\text{with } \bar{B}^r(d) \text{ the closed ball of radius } r \text{ centered at the origin in } \mathbb{R}^d$$

(I bend notation slightly, using the symbol for a transform X interchangeably with its parametrization $(\boldsymbol{t}, \boldsymbol{\theta})$). $\boldsymbol{t}$ is a standard translation vector, and $\boldsymbol{\theta}$ is an *orientation vector*. Orientation vectors are presented in detail in Appendix B; the main idea is that the direction of $\boldsymbol{\theta}$ defines the axis and its length the amount of rotation.

If F_c is the child frame and F_p the parent frame of a joint j with joint transform $X_j = (\boldsymbol{t}, \boldsymbol{\theta})$ then

- $\boldsymbol{t}$ gives the location of the origin of F_c in F_p, and

- $\boldsymbol{\theta}$ gives the rotation of F_c in F_p. Specifically, the axis of the extrinsic spatial rotation that makes F_c co-oriented with F_p contains the point $\boldsymbol{t}$ and is parallel to $\boldsymbol{\theta}$ (both $\boldsymbol{t}$ and $\boldsymbol{\theta}$ are taken to be in frame F_p), and the amount of (right-hand-rule) rotation about this axis is $\|\boldsymbol{\theta}\|$ radians.

Figure 3-5 illustrates the transformation between two arbitrary coordinate frames given by $\boldsymbol{t}$ and $\boldsymbol{\theta}$.

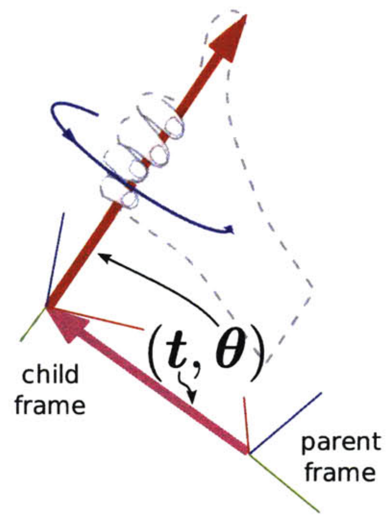


Figure 3-5: $(\boldsymbol{t}, \boldsymbol{\theta})$ transform representation.
$\boldsymbol{t}$ is a translation vector giving the location of the origin of a child frame in its parent frame. $\boldsymbol{\theta}$ is an orientation vector: the direction gives the axis and the length the amount (right hand rule, radians) of the spatial rotation of the child frame's axes in the parent frame.

One main reason I use this representation is that, because it contains only six parameters, there are no implicit constraints. Other popular representations (cf. [98]), including those based on unit quaternions and rotation matrices, contain more than 6 parameters—the total dimension of $SE(3)$—and account for this by requiring additional algebraic relationships to hold among them. This would put extra burden on the numeric SOLVE algorithm in Chapter 4.

The trade-off is that the global topology of the $(\boldsymbol{t}, \boldsymbol{\theta})$ parametrization space is actually not the same as the topology of $SE(3)$, which is non-Euclidean. I apply the *dynamic reparametrization* technique recently popularized by Grassia in [55] to handle this: whenever the state of a transform approaches a singularity in the parametrization space, a reparametrization is applied to a different set of parameters representing the same transform, but avoiding the singularity. The details are given in Appendix B.

The other major advantages of the $(\boldsymbol{t}, \boldsymbol{\theta})$ representation are (1) that it supports a compact implementation for the catalog of joint types, developed below Section 3.3.4;

and (2) that it admits a straightforward representation for the limits of common revolute and prismatic joints, Section 3.3.5.

Transform Computations

Three essential transform operations are composition (represented as multiplication of transforms), inversion, and the transformation of coordinate vectors (points). These are computed in the $(\boldsymbol{t}, \boldsymbol{\theta})$ parametrization as follows, building on the corresponding operations for orientation vectors and unit quaternions given in Appendices B and C, respectively:

$$(\boldsymbol{t}_v, \boldsymbol{\theta}_v)(\boldsymbol{t}_u, \boldsymbol{\theta}_u) = (\mathrm{vp}(\exp(\boldsymbol{\theta}_v)\mathring{\boldsymbol{t}}_u \exp(-\boldsymbol{\theta}_v)) + \boldsymbol{t}_v,\ \log(\exp(\boldsymbol{\theta}_v)\exp(\boldsymbol{\theta}_u))) \tag{3.15}$$

$$(\boldsymbol{t}, \boldsymbol{\theta})^{-1} = (-\,\mathrm{vp}(\exp(-\boldsymbol{\theta})\mathring{\boldsymbol{t}} \exp(\boldsymbol{\theta})),\ -\boldsymbol{\theta}) \tag{3.16}$$

$$(\boldsymbol{t}, \boldsymbol{\theta})\boldsymbol{v} = \mathrm{vp}(\exp(\boldsymbol{\theta})\mathring{\boldsymbol{v}} \exp(-\boldsymbol{\theta})) + \boldsymbol{t}. \tag{3.17}$$

The functions $\exp(\boldsymbol{\theta})$ and $\log(\mathring{\boldsymbol{q}})$ are the exponential and logarithmic maps defined in Eqs. B.9 and B.10; $\mathrm{vp}(\mathring{\boldsymbol{q}})$ and the operation $\mathring{\boldsymbol{t}}$ producing a quaternion from a vector $\boldsymbol{t} \in \mathbb{R}^3$ are defined in Eq. C.4. These equations may seem cumbersome relative to other representations, such as homogeneous transformation matrices, but it is not hard to optimize their implementation in practice with judicious caching of intermediate results ($\exp(\boldsymbol{\theta})$ in particular).

The residual computations in Chapter 4 use one more operation on transforms, the algebraic difference

$$X_1 - X_0 = (\boldsymbol{t}_1, \boldsymbol{\theta}_1) - (\boldsymbol{t}_0, \boldsymbol{\theta}_0) = (\boldsymbol{t}_1 - \boldsymbol{t}_0, \boldsymbol{\theta}_1 - \boldsymbol{\theta}_0) \tag{3.18}$$

($X_1 - X_0$ is not itself considered to be a transform). Though it is well-known [13] that there is no ideal distance metric on $SE(3)$, it is nevertheless true that

$$X_1 \to X_0 \iff \|X_1 - X_0\| \to 0, \tag{3.19}$$

as long as nearest orientation vector aliases are always selected (Section 4.9.1).

3.3.4 The Catalog of Joint Types

The domain $\mathcal{D}$ of the $(\boldsymbol{t}, \boldsymbol{\theta})$ representation is the product space $\mathbb{R}^3 \times \bar{B}^3(\pi)$, which can be considered embedded in $\mathbb{R}^6$:

$$\mathcal{D} = (\mathbb{R}^3 \times \bar{B}^3(\pi)) \subset \mathbb{R}^6, \tag{3.20}$$

though because the space of orientations is not Euclidean, the standard linear algebra of $\mathbb{R}^6$ does not also apply to $\mathcal{D}$.

Definition 18 The *type* Y_j of a joint j can now be precisely defined as a (semantically meaningful) subspace of the full $(\boldsymbol{t}, \boldsymbol{\theta})$ parameter space $\mathcal{D}$.

For example, the type of a revolute joint would be a 1-dimensional subspace of $\mathcal{D}$ in which the only variable is the length of the rotation vector.

This section will present my method for producing a useful *catalog* $\mathcal{Y}$ of joint types by intersecting *axis-aligned subspaces* of $\mathbb{R}^6$ with $\mathcal{D}$. This catalog is *expressive*, in the sense that all joint types used in a broad class of common articulated robots are included, including all the lower pairs (Appendix A) except helical, as well as a selection of several higher pairs; and it is *generative* in the sense of a grammar: an infinite set of novel constructions are possible. Expressiveness ensures virtual articulations are convenient in constraining high-DoF motion (Chapter 5) and in modeling contact, compliance, and uncertainty (Chapter 6). Generativity ensures that, even though the catalog is relatively small, it will suffice for a wide variety of robots and tasks.

Interestingly, a fairly powerful form of generativity is known even for linkages restricted to rotating (revolute in 2D, spherical in 3D) joints [37]. *Kempe's universality theorem* [80] states that even in this restricted context, it is possible to design a linkage which "signs your name,"[6] i.e. some link can be made to move along an arbitrary polynomial curve (2D) or surface (3D) [79]. This level of generativity is thus "free" because the joint catalog includes revolute and spherical joints.

[6]This particularly visual description is attributed to William Thurston [82].

The catalog will be developed in three parts. First the combinatorics of the entire family of axis-aligned linear subspaces of $\mathcal{D}$ will be analyzed. Then the results of this analysis will be collected in a formal definition of $\mathcal{Y}$. Finally, the actual implementation will be briefly introduced.

Combinatorics

Definition 19 Formally, the *axis-aligned (linear) subspace* $\mathcal{A}(\boldsymbol{a})$ of $\mathbb{R}^d$ identified by the binary *index* vector $\boldsymbol{a} \in \{0,1\}^d$ is the subspace

$$\mathcal{A}(\boldsymbol{a}) = \left\{\boldsymbol{p} = (p_0, \ldots, p_{d-1}) \in \mathbb{R}^d \mid \forall_{0 \leq i < d}\, (a_i = 0) \Rightarrow (p_i = 0)\right\} \qquad (3.21)$$
$$\text{with } \boldsymbol{a} = (a_0, \ldots, a_{d-1}) \in \{0,1\}^d$$

i.e. the i'th coordinate of $\boldsymbol{p}$ is zero if the i'th coordinate of $\boldsymbol{a}$ is zero, and is unconstrained otherwise.

For notational compactness, I use a d-bit binary integer a to refer to the corresponding binary vector $\boldsymbol{a} \in \{0,1\}^d$, and when I speak of $\mathcal{A}(a)$ as a subspace of $\mathcal{D}$ I formally mean

$$(\mathcal{A}(\boldsymbol{a}) \cap \mathcal{D}) \subseteq \mathbb{R}^d. \qquad (3.22)$$

In general there are are 2^d axis-aligned subspaces of $\mathbb{R}^d$, so for $\mathcal{D} \subset \mathbb{R}^6$ there is a family of $2^6 = 64$ subspaces to consider. Fortunately, not all are distinct for our purposes: Theorem 1, below, will establish that they are partitioned into only 20 equivalence classes. This reduction is made possible by the fact that the positioning transforms of a joint j, C_j and P_j in Eq. 3.9, are sufficient to *conjugate* the mobility transform M_j by an arbitrary rigid transform.

Definition 20 In this context, *conjugation* of a rigid transform M is the application of a rigid transform A, followed by M, followed by A^{-1}, with the aggregate effect of changing the coordinate frame in which M is applied.

Specifically, to conjugate M_j by an arbitrary transform A set $C'_j = AC_j$ and $P'_j = P_jA^{-1}$, where C_j and P_j were the original positioning transforms. Then

$$X_j = P_jM_jC_j = (P_jA^{-1})M_j(AC_j) = P'_jM_jC'_j. \tag{3.23}$$

Conjugation can be used to establish isomorphisms among the axis-aligned subspaces of $\mathcal{D}$. For example, the subspace of x-translations $\mathcal{A}(100000)$ is mapped to the subspace of y-translations $\mathcal{A}(010000)$ by the rigid transform $R_z(\pi/2) = (\boldsymbol{t} = \mathbf{0}, \boldsymbol{\theta} = (0, 0, \pi/2))$, i.e. a permuting rotation of $\pi/2$ about the z-axis.

Definition 21 In this context a *permuting rotation*, or *permutation*, is a rotation by a multiple of $\pi/2$ radians about a coordinate axis.

In fact, conjugacy forms an equivalence relation[7] $\sim$ on axis-aligned subspaces of $\mathcal{D}$. Let $\mathcal{U}$, $\mathcal{V}$, and $\mathcal{W}$ be such subspaces. Then $\mathcal{U} \sim \mathcal{U}$ via conjugation by the identity transform (reflexivity), $\mathcal{U} \sim \mathcal{V}$ by A implies $\mathcal{V} \sim \mathcal{U}$ by A^{-1} (symmetry), and $\mathcal{U} \sim \mathcal{V}$ by A and $\mathcal{V} \sim \mathcal{W}$ by B implies $\mathcal{U} \sim \mathcal{W}$ by BA (transitivity).

Definition 22 The equivalence classes under $\sim$ are called *conjugacy classes.*

Theorem 1 *There are exactly 20 conjugacy classes of the axis-aligned subspaces of $\mathcal{D}$. Four are one-member classes, 12 are three-member classes, and four are 6-member classes.*

Proof By exhaustive enumeration. This will be more than just tedious as it will introduce a nomenclature and symbology for the joint catalog.

First consider the subspaces $\mathcal{A}(000000)$ and $\mathcal{A}(111111)$. These form two of the singleton conjugacy classes, called **F** (*fixed*) and **G** (*general*), because they are the only zero resp. six-dimensional subspaces.

Next consider the 7 remaining translation-only subspaces $\mathcal{A}(xxx000)$. The three single-translation subspaces are equivalent: translation along one axis is mapped to

[7] A more general version of this property is well known, and is covered in e.g. [147].

translation along another by a permutation about the third. These form a three-member conjugacy class called $\mathbf{P}$, the *prismatic* class. Similarly, the three two-translation subspaces are pairwise equivalent by permutation about the common axis, and form the three-member conjugacy class $\mathbf{P}^2$. Finally, the three-translation subspace is unique and forms a third singleton conjugacy class, $\mathbf{P}^3$. $\mathbf{P}, \mathbf{P}^2$, and $\mathbf{P}^3$ are all distinct from each other because their included subspaces all differ in dimension, and distinct from $\mathbf{F}$ and $\mathbf{G}$ for the same reason.

Similarly, there are 7 remaining rotation-only subspaces $\mathcal{A}(000xxx)$. The three single-rotation subspaces are pairwise equivalent by permutation about a perpendicular axis, and form the *revolute* conjugacy class $\mathbf{R}$. Three subspaces allow rotation about any axis through the origin in one coordinate plane, are pairwise equivalent by permutation about the common axis (i.e. at the intersection of the planes), and form the *swing* conjugacy class $\mathbf{W}$. I call this *swing rotation* after [10], and I call the plane containing the possible axes of rotation the *swing plane*. Finally, the three-rotation subspace is unique and forms the fourth and final singleton conjugacy class, $\mathbf{S}$ (*spherical*). $\mathbf{R}, \mathbf{W}$, and $\mathbf{S}$ are distinct from each other because their included subspaces differ in dimension, and all are distinct from the previously defined classes either by difference in total dimension or in the number of translational dimensions.

The 8 subspaces $\mathcal{A}(xxx111)$ have unconstrained rotation. We have already covered two: $\mathbf{G} = \mathcal{A}(111111)$ and $\mathbf{S} = \mathcal{A}(000111)$. The remaining six form two more three-member conjugacy classes: $\mathbf{PS}$ permits one axis of translation, and $\mathbf{P}^2\mathbf{S}$ permits two. Subspaces in $\mathbf{PS}$ are pairwise equivalent by permutation about the axis perpendicular to their translation axes, and subspaces in $\mathbf{P}^2\mathbf{S}$ are pairwise equivalent by permutation about the common translation axis. By similar arguments the remaining six subspaces $\mathcal{A}(111xxx)$ with unconstrained translation split into two three-member conjugacy classes, $\mathbf{P}^3\mathbf{R}$ and $\mathbf{P}^3\mathbf{W}$ with one resp. two-axis rotation. $\mathbf{PS}, \mathbf{P}^2\mathbf{S}, \mathbf{P}^3\mathbf{R}$, and $\mathbf{P}^3\mathbf{W}$ are distinct from each other and from previously defined classes by difference either in number of translational or number of rotational dimensions.

So far we have covered 28 of the 64 total axis-aligned subspaces of $\mathcal{D}$, and identified all four of the singleton equivalency classes as well as 8 of the 12 three-member classes.

The remaining 36 subspaces have either one or two axes of translation and either one-axis or swing rotation.

The 18 remaining one-rotation subspaces are divided into two categories depending on whether the rotation axis is perpendicular to one/both translation axis/axes. There are two conjugacy classes in each case: **C** (*cylindrical*) and $\mathbf{P}^2\mathbf{R}$ have a rotation axis parallel to one resp. one of two translation axis/axes; **PR** and **E** (*planar*, from the German *ebene*[8]) have a rotation axis perpendicular to one resp. both of two translation axis/axes. **C** and **E** are each three-member classes of subspaces pairwise equivalent by permutation about the axis perpendicular to both rotation axes in the pair; $\mathbf{P}^2\mathbf{R}$ is a six-member class of subspaces pairwise equivalent by first a permutation about the common translation axis in the pair (if necessary) and then a permutation about the axis perpendicular to both rotation axes (if not already parallel); **PR** is another six-member class of subspaces pairwise equivalent by first a permutation about the axis perpendicular to both translation axes in the pair (if not already parallel) and then a permutation about the axis perpendicular to both rotation axes (if not already parallel). **C** and **PR** are distinct from each other due to the parallelism/perpendicularity of the rotation axis relative to the translation axis, and similar for $\mathbf{P}^2\mathbf{R}$ and **E**. **C** and **PR** are each distinct from $\mathbf{P}^2\mathbf{R}$ and **E** by difference in the number of translational dimensions, and similarly all four new of the new classes are distinct from previously defined ones by difference in either the number of translational or the number of rotational dimensions.

The final 18 subspaces all permit swing rotation. Adjoining swing rotation with one translational axis results in two conjugacy classes $\mathbf{P}_{\|}\mathbf{W}$ and $\mathbf{P}_{\perp}\mathbf{W}$ depending on whether the translation axis is parallel resp. perpendicular to the swing plane. $\mathbf{P}_{\|}\mathbf{W}$ is a 6-member equivalency class and $\mathbf{P}_{\perp}\mathbf{W}$ is a three-member class; in each class subspaces are pairwise equivalent by permutation about the axis perpendicular to both translation axes in the pair. Adjoining swing rotation with two translational axes results in the two final conjugacy classes $\mathbf{P}^2_{\|}\mathbf{W}$ and $\mathbf{P}^2_{\perp}\mathbf{W}$ depending on whether one of the translational axes is perpendicular to the swing plane. $\mathbf{P}^2_{\|}\mathbf{W}$ is a three-

[8]**P** is already taken, and Reuleaux was German [73].

member equivalency class, and $\mathbf{P}^2_\perp\mathbf{W}$ is a six-member class; subspaces in $\mathbf{P}^2_\|\mathbf{W}$ are pairwise equivalent by conjugation about the common translation axis in the pair, and subspaces in $\mathbf{P}^2_\perp\mathbf{W}$ are pairwise equivalent by first a permutation about the common translation axis in the pair (if necessary) and then a permutation about the intersection line of the swing planes (if necessary). $\mathbf{P}_\|\mathbf{W}$ and $\mathbf{P}_\perp\mathbf{W}$ are distinct from each other due to the relative parallelism/perpendicularity of the translation axis relative to the swing plane, and similar for $\mathbf{P}^2_\|\mathbf{W}$ and $\mathbf{P}^2_\perp\mathbf{W}$. $\mathbf{P}_\|\mathbf{W}$ and $\mathbf{P}_\perp\mathbf{W}$ are each distinct from $\mathbf{P}^2_\|\mathbf{W}$ and $\mathbf{P}^2_\perp\mathbf{W}$ by difference in the number of translational dimensions, and similarly all four of these final classes are distinct from previously defined ones by difference either in the number of translational or the number of rotational dimensions.

□

Some additional observations:

- every space in every class contains the identity transform $\mathbf{0}$, because every axis-aligned subspace contains $\mathbf{0} \in \mathcal{D}$
- the 5 lower pair joints other than helical correspond to the equivalence classes $\mathbf{S}, \mathbf{P}, \mathbf{R}, \mathbf{C}$, and $\mathbf{E}$
- $\mathbf{P}, \mathbf{R}, \mathbf{W}$, and $\mathbf{S}$ are indivisible, but the remaining 16 classes can be assembled by serial composition thereof.
- Let $\mathbf{R}$+$\mathbf{R}$+$\mathbf{R}$ be the serial concatenation of three mutually perpendicular revolute joints, and similarly, let $\mathbf{W}$+$\mathbf{R}$ be the serial concatenation of a revolute followed by a swing joint, again with perpendicular rotation axes. $\mathbf{S} \neq \mathbf{R}+\mathbf{R}+\mathbf{R}$ and $\mathbf{S} \neq \mathbf{W}+\mathbf{R}$ due to the possibility of gimbal lock both in $\mathbf{R}$+$\mathbf{R}$+$\mathbf{R}$ and in $\mathbf{W}$+$\mathbf{R}$
- $\mathbf{R}+\mathbf{R}$, the serial concatenation of two perpendicular revolute joints, is called a *universal* joint. $\mathbf{W} \neq \mathbf{R}+\mathbf{R}$, i.e. a swing joint is different from a universal joint, due to differences in the *induced twist*, a concept which is introduced in a similar situation in [10]. However, it is the case that $\mathbf{W}+\mathbf{R} = \mathbf{R}+\mathbf{R}+\mathbf{R}$ because the additional revolute freedom allows the induced twist to be compensated.

Formal Definition of $\mathcal{Y}$

Summarizing, the four one-member equivalence classes of axis-aligned subspaces of $\mathcal{D}$ are

$$\mathcal{Y}_1 = \{\mathbf{F}, \mathbf{G}, \mathbf{P}^3, \mathbf{S}\}, \tag{3.24}$$

the 12 three-member classes are

$$\mathcal{Y}_3 = \{\mathbf{P}, \mathbf{P}^2, \mathbf{R}, \mathbf{W}, \mathbf{PS}, \mathbf{P}^2\mathbf{S}, \mathbf{P}^3\mathbf{R}, \mathbf{P}^3\mathbf{W}, \mathbf{C}, \mathbf{E}, \mathbf{P}_\perp\mathbf{W}, \mathbf{P}^2_\parallel\mathbf{W}\}, \tag{3.25}$$

and the four 6-member classes are

$$\mathcal{Y}_6 = \{\mathbf{P}^2\mathbf{R}, \mathbf{PR}, \mathbf{P}_\parallel\mathbf{W}, \mathbf{P}^2_\perp\mathbf{W}\}. \tag{3.26}$$

Definition 23 The 20 equivalence classes essentially give the *joint catalog*

$$\mathcal{Y} = \mathcal{Y}_1 \cup \mathcal{Y}_3 \cup \mathcal{Y}_6, \tag{3.27}$$

except that the catalog is a set of joint types Y which are subspaces of $SE(3)$, not equivalence classes. This is simply addressed by picking a canonical member from each class identifying a representative subspace of $\mathcal{D}$, and thus $SE(3)$, as shown in table 3.1.

For brevity, I will consider the equivalence class to be a symbol for the representative subspace. So for example, when $Y = \mathbf{R}$ is considered a joint type, it refers to the subspace $\mathcal{A}(000001)$ (or sometimes just the index vector $\boldsymbol{a} = (0,0,0,0,0,1)$ or binary number $a = 000001$ depending on context).

Implementation

Computationally, the entire catalog could be encoded by a table with one entry per type, with all other code implemented in a sufficiently general way to handle any joint type uniformly. However, supporting an attractive and informative graphical rendering for each joint type seems to require some additional special-case code per joint type. Also, it is not strictly necessary to implement every joint type, both due to the fact that only $\mathbf{P}, \mathbf{R}, \mathbf{S}$, and $\mathbf{W}$ are indivisible and also due to Kempe's universality theorem showing the generativity of linkages based on $\mathbf{S}$ (or $\mathbf{R}$ in 2D) alone. Thus in the current software implementation I have chosen to provide only 11 of the full catalog of 20 types

$$\mathbf{R}, \mathbf{P}, \mathbf{C}, \mathbf{S}, \mathbf{E}, \mathbf{PR}, \mathbf{PS}, \mathbf{P}^2\mathbf{S}, \mathbf{P}^2, \mathbf{P}^3, \mathbf{G}, \tag{3.28}$$

with convenient names given table 3.1 and graphical renderings as shown in Figure 3-6.

		representative index						
equivalence class	class size	t_x	t_y	t_z	θ_x	θ_y	θ_z	implementation name
$\mathbf{R}$	3	0	0	0	0	0	1	revolute
$\mathbf{P}$	3	0	0	1	0	0	0	prismatic
$\mathbf{C}$	3	0	0	1	0	0	1	cylindrical
$\mathbf{S}$	1	0	0	0	1	1	1	spherical
$\mathbf{E}$	3	1	1	0	0	0	1	planar
$\mathbf{PR}$	6	1	0	0	0	0	1	pin-slider
$\mathbf{PS}$	3	1	0	0	1	1	1	point-slider
$\mathbf{P}^2\mathbf{S}$	3	1	1	0	1	1	1	point-plane
$\mathbf{P}^2$	3	1	1	0	0	0	0	cartesian2
$\mathbf{P}^3$	1	1	1	1	0	0	0	cartesian3
$\mathbf{G}$	1	1	1	1	1	1	1	general
$\mathbf{F}$	1	0	0	0	0	0	0	—
$\mathbf{P}^3\mathbf{R}$	3	1	1	1	1	0	0	—
$\mathbf{P}^2\mathbf{R}$	6	1	1	0	1	0	0	—
$\mathbf{W}$	3	0	0	0	1	1	0	—
$\mathbf{P}^3\mathbf{W}$	3	1	1	1	1	1	0	—
$\mathbf{P}_\perp\mathbf{W}$	3	0	0	1	1	1	0	—
$\mathbf{P}_\parallel\mathbf{W}$	6	1	0	0	1	1	0	—
$\mathbf{P}_\perp^2\mathbf{W}$	6	1	0	1	1	1	0	—
$\mathbf{P}_\parallel^2\mathbf{W}$	3	1	1	0	1	1	0	—

Table 3.1: The joint catalog.
Only entries with an "implementation name" are available in the current software implementation.

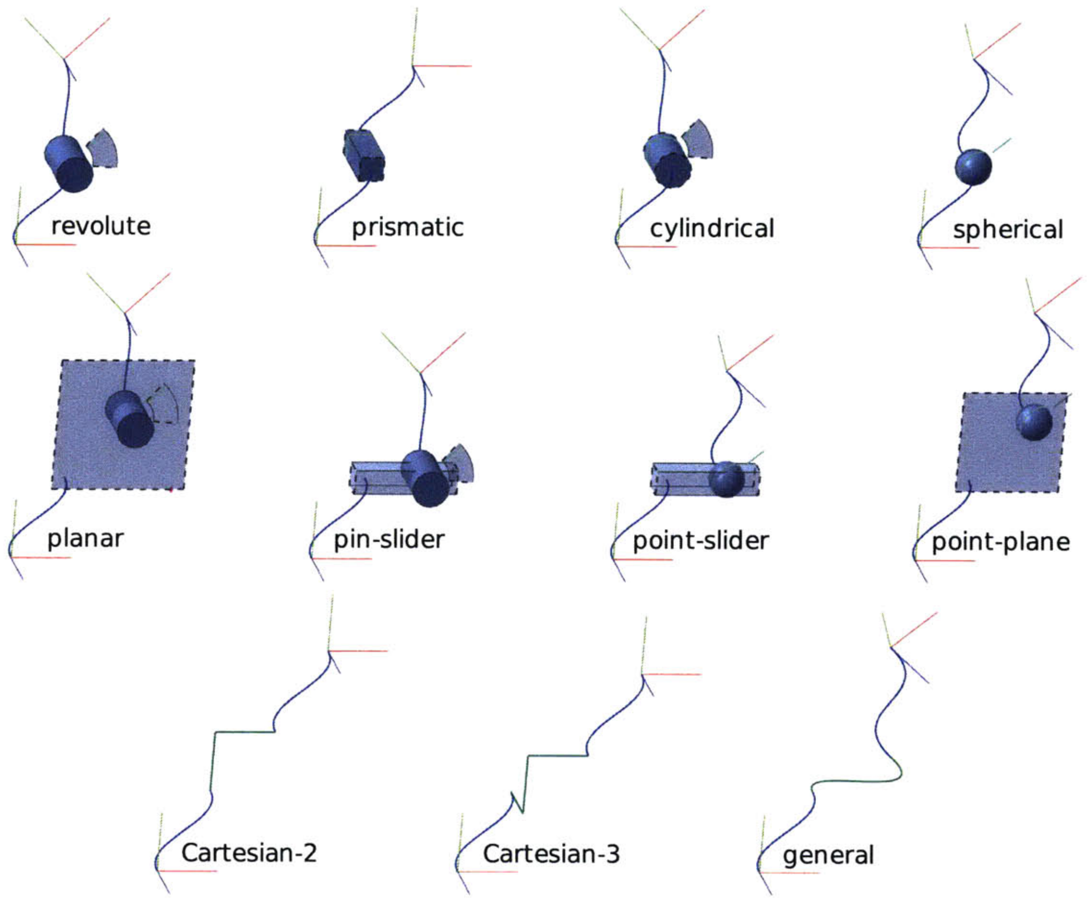


Figure 3-6: Joints available in the implementation.
Each joint type has a specific dynamic rendering in the 3D graphics window of the mixed real/virtual interface. Connections to the child and parent links are shown as Bézier curves; the link frames are rendered as red/green/blue axis triads.

3.3.5 Joint Motion Limits

Commonly, the mobility space $\mathcal{M}_j$ for a joint j is only a proper subspace of the prototypical space given by its type Y_j. For example, revolute joints in articulated robots are usually limited to a minimum and maximum rotation angle. Another use for joint limits is in modeling some types of contact changes, as explained in Chapter 6. The final part of the mobility representation O_j is a representation I_j for such limits, interpreted to "trim" or restrict the prototypical mobility space Y_j as symbolized in Eq. 3.7.

My method for representing joint limits is of the same flavor as the method used to form the joint catalog: I_j will identify another embedded subspace of $\mathbb{R}^6$; in this case I_j is selected from the family of axis-aligned bounding boxes.

Given the variety of mobility space topologies in the joint catalog, this representation for I_j balances generality with capability. Spherical and swing mobility are known [10] in particular to present a challenge—what is the most meaningful and concise way to specify limits for these multi-freedom rotations? Though this is an interesting question, I choose to avoid the issue. It is true that practical mechanical implementation of a spherical or swing joint would seem to require some motion limits, and such limits are also important for accurate models of biological joints (e.g. the human shoulder). However, spherical and swing joints seem to be quite rare in real articulated robots, with the exceptions being spatial parallel mechanisms (e.g. Gough-Stewart platforms), themselves relatively uncommon and specialized, and perhaps a handful of more esoteric mechanisms. A more flexible limit representation which better addresses these cases is a possible future extension.

My representation does work well for the more common types of limited mobility—prismatic, revolute, and combinations thereof.

Definition 24 The family of *axis-aligned bounding boxes*, with possibly infinite width in each dimension is

$$BB^d(\boldsymbol{l}, \boldsymbol{u}) = \{\boldsymbol{p} = (p_0, \dots, p_{d-1}) \in \mathbb{R}^d \mid \forall_{0 \leq i < d}\, l_i \leq p_i \leq u_i\} \tag{3.29}$$

$$\text{with } \textit{limit vectors } \boldsymbol{l} = (l_0, \dots, l_{d-1}), \boldsymbol{u} = (u_0, \dots, u_{d-1}) \in (\mathbb{R} \cup \{\pm\infty\})^d \tag{3.30}$$

$$\forall_{0 \leq i < d}\, l_i \leq u_i \text{ and } l_i = -\infty \iff u_i = +\infty. \tag{3.31}$$

Definition 25 I represent the *bounding space* I_j of a joint j by partitioning the limit vectors into translation and rotation components, which enables an additional constraint that is explained below:

$$I_\jmath = \left((\boldsymbol{l}_t, \boldsymbol{l}_\theta), (\boldsymbol{u}_t, \boldsymbol{u}_\theta)\right) \tag{3.32}$$

$$\text{with } (\boldsymbol{l}_t, \boldsymbol{u}_t) \text{ and } (\boldsymbol{l}_\theta, \boldsymbol{u}_\theta) \text{ satisfying Eqs. 3.30 and 3.31} \tag{3.33}$$

$$\begin{aligned} &\text{and either } \boldsymbol{l}_\theta = (-\infty, -\infty, -\infty) \text{ and } \boldsymbol{u}_\theta = (+\infty, +\infty, +\infty) \\ &\quad \text{or } BB^3(\boldsymbol{l}_\theta, \boldsymbol{u}_\theta) \subset B^3(2\pi). \end{aligned} \tag{3.34}$$

Then the bounding space is

$$BB^6\left((\boldsymbol{l}_t, \boldsymbol{l}_\theta), (\boldsymbol{u}_t, \boldsymbol{u}_\theta)\right) \subseteq \mathbb{R}^6, \tag{3.35}$$

and the formal semantics of the mobility space restriction are

$$\begin{aligned} &\text{with } Y = \mathcal{A}(a_0, \ldots, a_5) \text{ and } I = ((l_0, \ldots, l_5), (u_0, \ldots, u_5)) \\ &Y|_I = \left\{(p_0, \ldots, p_5) \in \mathbb{R}^6 \mid \forall_{0 \le \imath < 6} \text{ if } a_\imath = 0 \text{ then } p_\imath = 0 \text{ else } l_\imath \le p_\imath \le u_\imath\right\}. \end{aligned} \tag{3.36}$$

The ability to specify infinite bounds allows each axis of motion in a combined prismatic-revolute mobility space—**E**, **C**, or $\mathbf{P}^\jmath\mathbf{R}^k$ with $\jmath \in \{0, 1, 2, 3\}$ and $k \in \{0, 1\}$—to be limited (or not limited) independently. The constraint 3.34 means that either the orientation vector $\boldsymbol{\theta}$ is either limited to a subset of $B^3(2\pi)$ or it is not limited at all. Since dynamic reparametrization will always keep $\boldsymbol{\theta}$ within $\bar{B}^3(\pi)$, this simplifies finding an in-limits alias of $\boldsymbol{\theta}$ while still allowing revolute joint limits $l \le \theta_z \le u$ to be specified in a full $-2\pi < l \le u < 2\pi$ range (recall that the θ_x and θ_y components of a revolute joint's mobility are constrained to 0 already). Further discussion of this constraint is given in Section 4.9.1.

This completes the presentation of all of the parts of the joint mobility representation.

3.4 Joint Space, Configuration Space, and DoF

We now make some observations about, and definitions of, properties of both the mobility of a single joint and the combined mobility of all the joints in a linkage. Some of the terminology here is fairly standard in kinematics, but it is still helpful to see concretely how the various concepts are grounded in my framework.

3.4.1 Properties of the Mobility of a Single Joint

By construction, the mobility space $\mathcal{M}_j$ for a single joint j is a manifold, possibly with boundary, so it is a space with a well-defined dimension.

Definition 26 The number of *degrees of freedom* (DoF) f_j of j is the dimension of $\mathcal{M}_j$, and the number of *degrees of invariance* (DoI) i_j of j is

$$i_j = 6 - f_j. \tag{3.37}$$

The DoF themselves are the coordinate axes of $\mathcal{D}$ corresponding to non-zero entries in the binary index a of the prototypical axis-aligned subspace for the joint type Y_j, and similarly the DoI are the coordinate axes corresponding to the zero entries in a. For example, for a cylindrical joint $a = 001001$, the $f_j = 2$ DoF are (t_z, θ_z), and the $i_j = 4$ DoI are $(t_x, t_y, \theta_x, \theta_y)$.

Definition 27 The *state* $\boldsymbol{x}_j$ of a tree joint j is a column vector of the f_j components of its mobility transform corresponding to its DoF, and the state $\boldsymbol{y}_o$ of a closure joint o is a column vector of all 6 components of its mobility transform. The *invariant error* $\boldsymbol{e}_{io}$ of a closure joint o is a column vector of the i_o components of its mobility transform corresponding to its DoI.

Definition 28 For a single joint j, the *configuration space* $\mathcal{C}_j$ is equal to the mobility space $\mathcal{M}_j$. If j is unbroken then the mobility transform M_j is in $\mathcal{C}_j$, and is called a *configuration* of j; I call any assignment to M_j, even one which is not a configuration, a *pose* of j.

In robotics the term "configuration space" often refers more specifically to the space of *collision-free* configurations, but since no surface geometry is included in our model, there are no collisions. As mentioned at the beginning of the chapter, I do not focus on collision in most of this thesis, though it could be added in future extensions.

Joint Mobility Restriction and Generalization

The mobility space of a physical joint is typically fixed. However, there are contexts where it is nevertheless useful to consider virtually changing *restricting* or enlarging (*generalizing*) it. The possibilities are even more relaxed for virtual joints.

Definition 29 Let Y and I be the type and mobility of a joint with mobility space $\mathcal{M}$. (Virtually) changing to type Y' and/or limits I' s.t. the resulting mobility space $\mathcal{M}'$ is a proper subspace of $\mathcal{M}$, i.e.

$$(\mathcal{M}' = Y'|_{I'}) \subsetneq (\mathcal{M} = Y|_I), \tag{3.38}$$

is *joint mobility restriction.* The opposite procedure, (virtually) changing to type Y' and/or limits I' s.t. $\mathcal{M} \subsetneq \mathcal{M}'$, is *joint mobility generalization.*

Restriction and generalization are directly enabled by the SetType and SetLimits mutation primitives, Algorithms D.7 and D.8.

3.4.2 Properties of Whole-Linkage Mobility

All of these definitions extend to a full linkage L. Let J be the set of joints and $T \subseteq J$ be the tree joints with a particular ordering

$$T = \left(t_0, \cdots, t_{|T|-1}\right). \tag{3.39}$$

Definition 30 Then the *mobility space*[9] $\mathcal{M}$ of L is the product space of the individual tree joint mobility spaces, in order:

$$\mathcal{M} = \prod_{0 \le i < |T|} \mathcal{M}_{t_i}. \tag{3.40}$$

Definition 31 I call any point in $\mathcal{M}$ a *pose* of L.

$\mathcal{M}$ is a manifold, possibly with boundary, because it is the product space of such.

Definition 32 The number of *DoF* f of L is the dimension of $\mathcal{M}$:

$$f = \sum_{j \in T} f_j. \tag{3.41}$$

The number of *DoI* i of L is the sum of the number of DoI of the closure joints in L:

$$i = \sum_{j \in J \setminus T} i_j. \tag{3.42}$$

Definition 33 The *tree state* $\boldsymbol{x}$ of L is the concatenation of the states of its tree joints, in the order given above

$$\boldsymbol{x}^T = \left(\boldsymbol{x}_{t_0}^T, \cdots, \boldsymbol{x}_{t_{|T|-1}}^T\right) \in \mathcal{M} \subset \mathbb{R}^f. \tag{3.43}$$

Similarly, let

$$C = \left(o_0, \cdots, o_{|J \setminus T|-1}\right) \tag{3.44}$$

be an ordering of the c closure joints $J \setminus T$ of L.

[9]The term "joint space" is often used for $\mathcal{M}$, but I prefer "mobility space" to avoid confusion with the case of a single joint, where I use different definitions for the terms (Eqs. 3.10 resp. 3.7).

Definition 34 Then the *closure state* $\boldsymbol{y}$ of L is the concatenation of the states of its closure joints in order, and the *invariant error* $\boldsymbol{e}_\imath$ of L is the concatenation of the invariant errors of the closure joints in order

$$\boldsymbol{y}^T = \left(\boldsymbol{y}_{o_0}^T, \cdots, \boldsymbol{y}_{o_{|C|-1}}^T\right) \in \mathcal{D}^{|C|} \subset \mathbb{R}^{6|C|} \tag{3.45}$$

$$\boldsymbol{e}_i^T = \left(\boldsymbol{e}_{\imath o_0}^T, \cdots, \boldsymbol{e}_{\imath o_{|C|-1}}^T\right) \in \mathbb{R}^\imath. \tag{3.46}$$

$\boldsymbol{e}_\imath$ is a projection $\Pi_\imath$ of $\boldsymbol{y}$,

$$\boldsymbol{e}_i = \Pi_\imath \boldsymbol{y}, \tag{3.47}$$

with $\Pi_\imath$ an $[i \times 6|C|]$ binary matrix mapping $\boldsymbol{y}$ to $\boldsymbol{e}_i$ by selecting only the DoI of each closure joint. That is, each row p of $\Pi_\imath$ contains exactly one non-zero entry, say at column q, where q corresponds to the p'th entry in the concatenated sequence of all DoI of all joints in C.

Definition 35 A *configuration* of L is a point in $\mathcal{M}$ corresponding to a pose of the linkage in which no closure joint is broken, and the *configuration space* $\mathcal{C}$ of L is the subspace of $\mathcal{M}$ consisting of all its configurations

$$\mathcal{C} \subseteq \mathcal{M} \text{ s.t. } \forall_{\boldsymbol{x} \in \mathcal{M}}\ \boldsymbol{x} \in \mathcal{C} \iff \boldsymbol{x} \text{ is a configuration.} \tag{3.48}$$

Unlike the case for a single joint, it is entirely possible that $\mathcal{C} \neq \mathcal{M}$.

3.4.3 Local vs. Global Properties of $\mathcal{C}$

The global shape of $\mathcal{C}$ is a classic study in kinematics (e.g. [124])—it may be empty, it may be disconnected, and it may contain both discrete and continuous components. These are interesting and useful properties, but I do not focus on them. Instead, my approach is to rely as much as possible on the shape of $\mathcal{C}$ *local* to the current state $\boldsymbol{x}$

of the linkage. This makes my approach more related to the field of control than to planning.

Definition 36 The *forward kinematic mapping* FK of L computes $\boldsymbol{y}$ given $\boldsymbol{x}$

$$\text{FK} : \mathcal{M} \to \mathcal{D}^{|C|} \quad \text{FK}(\boldsymbol{x}) = \boldsymbol{y}, \tag{3.49}$$

and is computable by composing appropriate instantiations of Eqs. 3.13, 3.15, and 3.16; a detailed algorithm is given in Appendix G.

Figure 3-7 illustrates $\text{FK}(\boldsymbol{x})$.

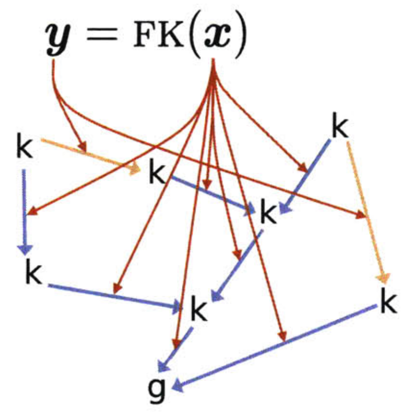


Figure 3-7: The forward kinematic mapping.
In this formulation, the *forward kinematic mapping* FK computes the *closure state* $\boldsymbol{y}$ as a function of the *tree state* $\boldsymbol{x}$. $\boldsymbol{y}$ is a concatenated vector of the closure joint $(\boldsymbol{t}, \boldsymbol{\theta})$ state vectors; $\boldsymbol{x}$ is a concatenated vector of the DoF of the tree joints.

The differentiability of the underlying operations implies that FK is itself differentiable; an algorithm to compute derivatives of FK will be presented in Section 4.9.2.

Definition 37 $\mathcal{C}$ is the kernel (zero set) of the related *forward invariant error* mapping

$$\text{FK}_i : \mathbb{R}^f \to \mathbb{R}^i \quad \boldsymbol{e}_i = \text{FK}_i(\boldsymbol{x}) = \Pi_i \text{FK}(\boldsymbol{x}). \tag{3.50}$$

And the local shape of $\mathcal{C}$ at $\boldsymbol{x}$ is approximated by the local linearization

$$\underset{[\imath \times f]}{J_\imath(\boldsymbol{x})} = \frac{\partial \boldsymbol{e}_\imath}{\partial \boldsymbol{x}} = \frac{\partial}{\partial \boldsymbol{x}} \mathrm{FK}_\imath(\boldsymbol{x}). \tag{3.51}$$

Specifically, the neighborhood of $\mathcal{C}$ at $\boldsymbol{x}$ is approximated by the tangent space of vectors $\Delta\boldsymbol{x}$ s.t.

$$\mathbf{0} = J_\imath(\boldsymbol{x})\Delta\boldsymbol{x}. \tag{3.52}$$

In robotics this approach dates back at least to Whitney's work in the late 1960s [164], and it has also been popular in graphics, where Gleicher called it the "differential control" approach [54]. For our present study, it will be shown in subsequent chapters that this local approach yields both a consistent theory for modeling virtual articulations and kinematic abstractions and also a useful implementation.

The few global concepts I use are as follows.

Definition 38 If $\mathcal{C} = \emptyset$ then L is *inconsistently over-constrained*, or just *inconsistent* or *over-constrained.* Otherwise I consider only the component of $\mathcal{C}$ containing the current configuration[10], and say that L is *consistent.* If this component is continuous then L is (locally) *under-constrained*; this is the common case. Otherwise, if the local component is discrete, i.e. a single point, then L is (locally) *rigid.*

3.5 Hierarchical Linkages

The framework presented so far is sufficient to model linkages of arbitrary topology, but it does not include any strong way to structure, or subdivide, large graphs. Subdivision can make sense in practice since many real-world robots contain repeated kinematic sub-structure, e.g. multiple copies of identical legs. When possible, breaking a model up into *decoupled* components can also speed up motion computations (cf. the ANALYZE algorithm in Chapter 4). Further, subdivision is important in using virtual articulations in an operator interface for high-DoF mechanisms (Chapter 5)—

[10]We can assume that the current pose of a consistent linkage L is actually a configuration, because the iterative solver acts to move any pose $\boldsymbol{x}$ onto $\mathcal{C}$, and then to keep it there.

in many cases the operator can specify motions independently for smaller parts of the model, and thus break up the job of motion specification. Taking this a step further, virtual articulations combined with model hierarchy enable my technique of *structure abstraction* (Section 3.5.3, below), where a complex kinematic subsystem is abstracted as a simpler virtual mechanism capturing the intended motion.

A unique feature of my framework is that it supports these kinds of subdivision by structuring the overall model into a hierarchy (tree) of properly nested *sub-linkages*.

Definition 39 Each *sub-linkage* L has the representation developed above (Def. 10 and its subsidiary components), with two modifications: some joints are *crossing joints*, detailed below, which connect between links in differing sub-linkages, and the ground link of a sub-linkage has a root joint, attaching it to the ground link of the enclosing (parent) linkage. Sub-linkages contain a reference p_L to this enclosing linkage and a *disposition* $d_L \in \{\emptyset, \text{driving}, \text{driven}\}$; a sub-linkage with $d_L = \emptyset$ is *simultaneous*. The *top-level* sub-linkage, at the root of the hierarchy, has $p_L = d_L = \emptyset$. The ground link of the top-level sub-linkage is referred to as g_0.

Figure 3-8 shows an example containing driving, driven, and simultaneous sub-linkages.

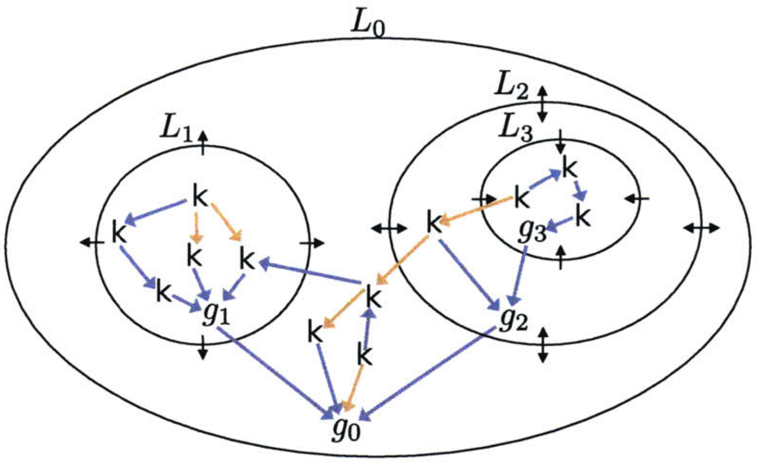


Figure 3-8: A hierarchical linkage
Sub-linkage L_1 is driving, L_2 is simultaneous, and L_3 is driven. Tree joints are represented by dark colored arrows; closure joints are light colored.

Definition 40 The *flattening* of a hierarchical linkage is a non-hierarchical linkage which results from simply ignoring all sub-linkage boundaries.

3.5.1 Sub-linkage Disposition

The disposition of a sub-linkage determines whether

- it is considered rigid with respect to its enclosing linkage (driving)
- its enclosing linkage is considered rigid with respect to it (driven)
- it is considered mobile in the same context as its enclosing linkage (simultaneous).

Simultaneous sub-linkages can serve to demarcate a connected kinematic sub-system, but do not actually change the way motion is computed. Driving and driven sub-linkages enforce new constraints on the possible relative motion of the sub-linkage with respect to its parent—in each case, one will act as if it were a single rigid link with respect to the other. For physical linkages, such a subdivision may only be really enforceable if the driving component is fully actuated (i.e. if there is rigid kinematic control authority over all DoF). But we are free to define the actuation of virtual articulations as we wish, so this kind of hierarchy is another area uniquely strong for mixed physical/virtual systems.

This chapter covers the structure of, and topological operations on, hierarchical linkages. Later, Section 4.10.1 in Chapter 4 will present a *hierarchical decomposition* algorithm which computes motion of driving and driven sub-linkages.

3.5.2 Crossing Joints

In the hierarchical case an overall spanning tree must still exist, rooted at the ground link g_0 of the top-level sub-linkage, and spanning all links in all descendant sub-linkages. To maintain this overall connectivity (and, generally, to be less than trivial), there must be *crossing joints* which connect between links in different sub-linkages.

Definition 41 Each joint j in a hierarchical linkage either connects two links in the same sub-linkage L, or is a *crossing* joint connecting a link in one sub-linkage L with a link in the enclosing sub-linkage. In both cases j is a member of the set of joints of L (only).

Note that crossing joints which e.g. connect links in different sibling sub-linkages, or from a sub-linkage to an ancestor or descendant more distant than parent or child, are not permitted. Only crossing joints to the parent or an immediate child sub-linkage are allowed.

I make a distinction between crossing joints depending on their parent-child directionality.

Definition 42 An *outcrossing* joint connects its child link in a sub-linkage to a parent link in the enclosing sub-linkage, and vice-versa for an *incrossing* joint.

In addition to the above constraint that crossing joints cross no more than one sub-linkage boundary, some additional topological restrictions are also imposed on crossing tree joints (there are no further constraints on crossing closures):

- **Constraints on Outcrossing Tree Joints**: Each sub-linkage L always has exactly one outcrossing tree joint (except for the top-level sub-linkage, which has none) connecting g_L to a link in the enclosing linkage E. This ensures that, considered individually, each sub-linkage has its own well-defined spanning tree. Re-grounding (see Algorithm 3.6, MAKEGROUND, later in this chapter) the sub-linkage automatically switches the outcrossing tree joint, and vice-versa.

- **Constraints on Incrossing Tree Joints**: There are no restrictions on incrossing tree joints for simultaneous or driving sub-linkages, but incrossing tree joints are disallowed for driven sub-linkages. Otherwise, changing the relative pose of links in the driven sub-linkage could change the relative pose of other links in the enclosing linkage, violating the semantics that the enclosing linkage is rigid with respect to a driven sub-linkage.

Also, the root joint of the ground link g_L of a sub-linkage L (other than the top-most) is always an outcrossing joint connecting it to the ground link of the enclosing linkage. The root joints of the other links in L connect to g_L.

3.5.3 Structure Abstraction

In computing, it has long been accepted that large systems can be effectively structured by subdividing them into smaller parts, defining *abstractions* to represent the *interface* each such part presents to the rest of the system. Ideally, the interface has significantly less complexity than the *implementation* (the abstracted subsystem).

I now show a particular way to use hierarchical linkages and virtual articulations to implement a version of abstraction in the domain of kinematics.

Definition 43 *Structure abstraction* is achieved by (a) encapsulating a connected part A of a linkage L s.t. A becomes demarcated as a simultaneous sub-linkage of of L, and (b) substituting a virtual linkage I for A in L, with I simultaneous in L and A a driven sub-linkage of I. A is the *abstracted sub-linkage* (the "implementation") and I is its *interface linkage.*

Figure 3-9 shows an example. The concept parallels traditional abstraction in computing: I should be simpler than A, but capture all of the behavior of A that would be relevant to the surrounding mechanism.

Making A a driven sub-linkage of I is not the only possible way to define a notion of abstraction in kinematics. For example, keeping A simultaneous could also make sense. However, the design choice to make A driven defines a fairly strong form of abstraction: motion of L, with I substituting for A, is *independent* of the actual motion of A. This permits A to be effectively decoupled, a simplification that can be helpful both (a) to an operator designing a motion for a high-DoF robot, and (b) to the system as it computes motions for a large linkage via the SOLVE algorithm in Chapter 4. Section 5.4 gives an example.

This decoupling power comes with a trade-off: there is no built-in constraint to ensure that A can reach every configuration to which it may be driven by I.

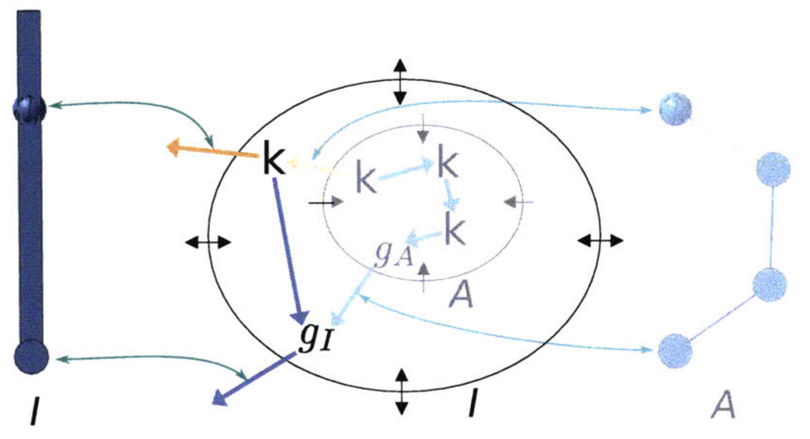


Figure 3-9: Structure abstraction.
In this example the "implementation" A, a series chain of four revolute joints, is virtually replaced by a simpler piston-like "interface" I. Also see Figure 1-3.

Definition 44 I is a *proper abstraction* of A if all of the reachable configurations of I, when embedded in the surrounding linkage L, drive reachable configurations of A.

It would be desirable to have an efficient algorithm that could determine, for any L, I, and A, whether I is a proper abstraction. This may be possible in some special cases, but unfortunately, the general case is easily shown to be hard.

Theorem 2 *Determining whether an abstraction is proper is PSPACE-hard.*

PROOF By reduction from the *reachability problem* for 2D revolute-joint linkages. Kempe's theorem establishes that the interface linkage I could be constructed s.t. the links which drive A could move to arbitrary spatial poses. But the problem of determining the *reachability* of arbitrary configurations of A from a starting configuration is known to be PSPACE-hard even for the restricted case of 2D linkages with only revolute joints [34, 70].

□

Thus, if a proper abstraction is required, it is up to the designer of that abstraction to ensure it. However, even improper abstractions can be useful when combined with the constraint prioritization features of the SOLVE algorithm—if the closure invariants in A are given higher priority (using sub-priorities, if necessary, Sec. 4.7.1) than those of, say, the crossing joints between I and A, then the motion of A will remain feasible. It will not exactly match the driving links in I, but because they are virtual, this can be acceptable.

3.6 Interacting with Model Structure

We have now covered the details of *what* a linkage is, including both its structure and its state. The rest of this chapter considers *how* particular linkage structures arise and evolve. Algorithms for this level of topological mutation have not often been presented in detail because there are typically costs (mechanical complexity, physical time, etc.) incurred to support topological dynamics in purely physical (traditional) linkages. But virtual articulations do not necessarily have these costs, and both operator-interface and compliant motion modeling applications call for adding, removing, and reconfiguring them on-line.

The main result here will be a set of *structure mutation* primitives that are convenient for common kinds of structural and topological changes. In some cases there will be associated manipulation of joint pose (vs. structure or topology), though the main discussion of linkage motion is left to Chapter 4.

The set of mutation primitives I present, summarized in table 3.2, is sufficient to build any linkage structure. For conciseness, this chapter only contains the more interesting operations—the remaining algorithms are collected in Appendix D. A set of lower-level helper functions, listed in table 3.3, is also given in Appendix D.

3.6.1 Notation and Assumptions

The mutation primitive algorithms are given in pseudocode that is block-structured and call-by-reference. Other than formal parameters, types are implied. Local variables are introduced with "let"; the symbol $\leftarrow$ indicates assignment. Error is implied whenever argument preconditions fail.

primitive name	chapter	page	description
AddLink	D	227	add a new link
SetLinkName	D	227	change the name of a link
AddJoint	D	227	add a new joint
SetJointName	D	227	change the name of a joint
MakeTree	3	87	change the disposition of a joint
MakeClosure	3	88	change the disposition of a joint
SetParent	3	88	re-attach a joint
SetChild	3	89	re-attach a joint
RemoveJoint	D	228	delete a joint
RemoveLink	D	228	delete a link
SetType	D	228	change the type of a joint
SetLimits	D	229	change the limits of a joint
Invert	3	90	invert a joint
MakeGround	3	91	switch to a different ground link
RepositionLink	D	229	move a link relative to adjacent joints
RepositionJoint	D	229	move a joint relative to adjacent links
Restructure	D	230	modify joint positioning transforms
Split	D	230	insert a new link and joint in-place
Merge	D	230	delete a joint and its child link
Encapsulate	3	92	carve out a new sub-linkage
Dissolve	3	92	merge a sub-linkage to its parent

Table 3.2: The structure mutation primitives.

Defs. 10 and 16 formally defined the linkage structure as a collection of several kinds of tuples. The pseudocode often references or mutates specific components of these tuples; where necessary the notation "p_w" indicates part p of a tuple within the definition of whole w. For example, n_j could refer to the name of a joint j, M_j could refer to its mobility transform[11], and n_{p_j} would be the name of its parent link.

Assumptions

Many algorithms use set operations, including adding and removing members of a set and checking set membership. The analyses assume these operations can be completed in $O(1)$ time, though common hashcode-based implementations are actually only $O(1)$ expected. The analyses also assume that the time required to iterate over the set of

[11]Technically, M_j is a component of the mobility representation O_j, which in turn is a component of the representation of joint j.

helper name	chapter	page	description
TREE?	D	232	test joint disposition
CLOSURE?	D	232	test joint disposition
ROOTLINK?	D	232	check if a link is parented to its root
ROOTJOINT?	D	232	check if a joint is a root
ROOTNAME	D	232	generate a root joint name
OUTCROSSING?	D	232	check if joint is outcrossing
INCROSSING?	D	232	check if joint is incrossing
CROSSING?	D	232	check if joint is incrossing or outcrossing
CLAMPV	D	233	clamp a $\boldsymbol{t}$ or $\boldsymbol{\theta}$ vector
CLAMPX	D	233	clamp a $(\boldsymbol{t}, \boldsymbol{\theta})$ transform
CMT	D	233	get a composite model transform

Table 3.3: Helper functions for the structure mutation primitives.

tree or closure joints adjacent to a link is proportional to the size of that set (and not, e.g., to the size of the set of all joints in the linkage). This is easily handled in the implementation with appropriate bookkeeping.

Maintaining Root Joints

A main design goal for the structure mutation primitives is that any sequence of valid mutations should result in a linkage which satisfies all of the properties set out in Definition 10 (and subsidiary definitions). In particular, the kinematic graph must remain connected, and the set of link parent joints must always form a directed spanning tree rooted at the ground link. In some cases, this involves checking for (and refusing to perform) invalid actions, such as re-parenting a joint to a link that is its own descendant. Where possible, however, validity is guaranteed by construction, and a key feature supporting this is the ever-presence the root joint for each link (Section 3.2.3). Root joints are maintained internally by the mutation algorithms. They are always of type **G**, are never limited, and cannot be explicitly renamed, re-attached, removed, inverted, repositioned, or merged.

3.6.2 Changing Joint Disposition and Reconnecting Joints

We now come to the first category of mutation primitives, which change the disposition (tree vs. closure) or the connectivity of a joint. Newly created joints (except for root joints) are always closures (cf. ADDJOINT on page 227). If no joint is ever changed to tree disposition then a pure Cartesian model results (Section 3.2.3), so whenever a reduced coordinate model is intended, some joints must be made tree joints. MAKETREE, Algorithm 3.1, does this by converting the prior tree parent of a joint j's child link c to closure, and replacing the tree parent of c with j. j's mobility transform is clamped to the allowed mobility space and limits as necessary. Thus, if j was a broken closure before the call to MAKETREE, c (and any descendants) will necessarily change global pose due to the clamping.

```
Algorithm 3.1: MAKETREE(j)
  Input: closure joint $j$ in linkage $L$ (i.e. $j \in J_L$)
  Output: $j$, now a tree joint
  for $i \leftarrow p_j$ to $g_0$ do if $i = c_j$ then error $c_j$ is an ancestor of $p_j$ else $i \leftarrow i_{p_p}$
  if OUTCROSSING?($j$) then MAKEGROUND($c_j$)
  else if INCROSSING?($j$) and $d_L$ = driven then
    error incrossing tree joint on driven sub-linkage
  else if ($c_j = g_L$) then MAKEGROUND($p_j$)
  $M_j \leftarrow$ CLAMPX($(\text{CMT}(pm_j)^{-1}\text{CMT}(cm_j))^{\phi_j}, Y_j, I_j$)
  $M_{p_{c_j}} \leftarrow \emptyset$ (previous parent of $c_j$ becomes a closure), $p_{c_j} \leftarrow j$
  return $j$
```

Unless c is the ground link or j is outcrossing, which are special cases, MAKETREE is $O(h)$ with h the spanning tree height of the parent link of j when updated CMTs are available (see discussion with CMT on page 233). If updated CMTs are not available then h is the maximum spanning tree height of the parent and child links of j.

When c is the ground link or j is outcrossing a re-grounding is triggered via MAKEGROUND, given below, and in that case the time complexity of MAKETREE is dominated by the call to MAKEGROUND.

MAKECLOSURE, Algorithm 3.2, is the complimentary operation to MAKETREE, and converts a joint currently in the spanning tree to closure disposition. The child link is re-parented to its root joint, whose mobility transform is updated so that the link does not change global pose due to the change in topology.

```
Algorithm 3.2: MakeClosure($j$)
    Input: tree joint $j$
    Output: $j$, now a closure
    $M_{r_{c_j}} \leftarrow \textsc{cmt}(c_j)$, $M_j \leftarrow \emptyset$, $p_{c_j} \leftarrow r_{c_j}$
    return $j$
```

MakeClosure is $O(1)$, assuming updated CMTs are available, or $O(h)$ where h is the spanning tree height of the child link if its CMT must be re-computed.

SetParent and SetChild, Algorithms 3.3 and 3.4, are the main operations for reconnecting existing joints, and thus for evolving the topological connectivity of a linkage. SetParent first checks that the new parent is not a descendant of the joint, and SetChild similarly checks that the new child is not an ancestor. Both operations maintain the disposition of the joint. For SetParent this requires no additional work, but to handle the case of changing the child of a tree joint, SetChild temporarily makes the joint a closure, which will re-parent the prior child to its root joint. Both operations also maintain the global poses of the adjacent links, with SetParent putting the "slack," i.e. the difference between the relative pose of the new parent and the joint's child vs that of the old parent and child, into the positioning transform P_j, and SetChild putting the slack into C_j.

```
Algorithm 3.3: SetParent($j, p$)
    Input: non-root joint $j$ in linkage $L$ (i.e. $j \in J_L$), link $p \in \{K_L \cup K_{p_L}\}$
    Output: $j$, now a child of $p$
    if $p \in K_{p_L}$ and $c_j \in K_{p_L}$ then error at least one endpoint must be in $L$
    if $n_j$ not unique in $p$ then error name not unique
    let $t \leftarrow \textsc{tree?}(j)$, $Q \leftarrow \textsc{cmt}(pm_j)$
    if $t$ then
        for $i \leftarrow p$ to $g_0$ do if $i = c_j$ then error $p$ is a descendant of $j$ else $i \leftarrow i_{p_p}$
    $p_j \leftarrow p$, $P_j \leftarrow \textsc{cmt}(p)^{-1}Q$
    if $t$ and $p \in K_{p_L}$ then MakeGround($c_j$)
    return $j$
```

SetParent is $O(h)$ where h is the spanning tree height of the new parent link, if updated CMTs are available, or the maximum of the heights of the new and old

Algorithm 3.4: SETCHILD(j, c)

```
Input: non-root joint $j$ in linkage $L$ (i.e. $j \in J_L$), link $c \in \{K_L \cup K_{p_L}\}$
Output: $j$, now the parent of or a closure attached to $c$
if $c \in K_{p_L}$ and $p_j \in K_{p_L}$ then error at least one endpoint must be in $L$
let $t \leftarrow$ TREE?($j$), $Q \leftarrow$ CMT($cm_j$)
if $t$ then
    if $c \in K_{p_L}$ and $d_L =$ driven then
        error incrossing tree joint on driven sub-linkage
    for $i \leftarrow p_j$ to $g_0$ do if $i = c$ then error $c$ is an ancestor of $j$ else $i \leftarrow i_{p_p}$
    MAKECLOSURE($j$)
$c_j \leftarrow c$, $C_j \leftarrow Q^{-1}$CMT($c$)
if $t$ then MAKETREE($j$)
return $j$
```

parent otherwise. SETCHILD is $O(h)$ where h is the height of the joint's parent if updated CMTs are available, or the maximum of the heights of the parent, the current child, and the new child otherwise.

3.6.3 Inverting Joints and Re-grounding

The next two mutation operations, INVERT and MAKEGROUND, are somewhat more complex. INVERT, Algorithm 3.5, flips the topology of a joint in-place, so that its prior parent becomes its new child, and vice-versa. For a closure joint this is a local procedure: the mobility space positioning transforms are swapped and inverted, the parent and child links are swapped, and the mobility inversion flag is flipped. The time complexity in this case is $O(1)$. For a tree joint, inversion triggers a re-grounding—the child link c of the joint to be inverted becomes the new ground link, and all the other joints on the tree path from c to the prior ground are also inverted. For simplicity, inversion is not supported for the case of crossing tree joints. The time complexity for inverting a tree joint is $O(h + |K|)$ where h is the spanning tree height of the parent of the joint to be inverted. The $O(|K|)$ term is due to the call to MAKEGROUND, at which point c will always be a root link.

MAKEGROUND, Algorithm 3.6, switches the ground link of a linkage, and also has two cases. The shorter one is when the new ground link l is not currently a root link, i.e. is not parented directly to the current ground link via its root joint. In this case

```
Algorithm 3.5: INVERT($j$)
Input: non-root joint $j$
Output: $j$, now inverted
let $p \leftarrow p_j$, $c \leftarrow c_j$
if CLOSURE?($j$) then
    if $n_j$ not unique in $c$ then error name not unique
    let $C \leftarrow C_j$, $P \leftarrow P_j$
    $p_j \leftarrow c$, $c_j \leftarrow p$, $C_j \leftarrow P^{-1}$, $P_j \leftarrow C^{-1}$, $\phi_j \leftarrow -\phi_j$
else
    if CROSSING?($j$) then error cannot invert crossing tree joint
    let $H$ be an empty sequence, $i \leftarrow p$
    while $\neg$ROOTLINK?($i$) and $i \neq g_0$ do
        if $n_{p_i}$ not unique in $i$ then error name not unique
        else append $p_i$ to $H$
        $i \leftarrow i_{p_p}$
    end
    foreach $i$ in $H$ in order do MAKECLOSURE($i$)
    MAKEGROUND($c$)
    foreach $i$ in $H$ in order do MAKETREE($i$)
return $j$
```

MAKEGROUND actually defers to INVERT on the parent joint of l. Though INVERT will itself call back to MAKEGROUND, there is no circularity because at the time of this re-entrant call, l will always be a root link. The time complexity in this case is the same as that of INVERT (tree joint case) on the parent joint.

Making a current root link l the ground link involves three steps. First, the root joints of all other links are re-parented to l. Second, a root joint is connected from the prior ground link to the new ground. Third, the root joint for l is re-attached wherever the old ground link's root joint was attached. The time complexity in this case is $O(|K|)$.

3.6.4 Operations on Sub-Linkages

The final algorithms presented here, ENCAPSULATE and DISSOLVE, Algorithms 3.7 and 3.8, give essential operations for creating and removing sub-linkages. ENCAPSULATE sub-divides an existing linkage by wrapping a sub-tree as a new sub-linkage; DISSOLVE does the opposite by merging an existing sub-linkage into its parent.

```
Algorithm 3.6: MakeGround(k)
  Input: non-ground link $k$ in linkage $L$ (i.e. $k \in K_L$)
  Output: $k$, now the ground link of $L$
  if RootLink?(k) then
    foreach joint $j$ s.t. $p_j = g_L$ if RootJoint?(j) do
      if tree?(j) then $M_j \leftarrow M_{r_k}^{-1} M_j$
      $p_j \leftarrow k$
    end
    let $O = (M_r, Y_r, I_r, \phi_r, P_r, C_r) \leftarrow (M_{r_k}^{-1}, \mathbf{G}, \emptyset, +1, \mathbf{0}, \mathbf{0})$ ▷Eq. 3.11
    let $r = (p_r, c_r, O_r, \Upsilon_r, n_r) \leftarrow (k, g_L, O, \emptyset, \text{RootName}(g_L))$ ▷Eq. 3.3
    $p_{r_k} \leftarrow p_{r_{g_L}}$, $p_{g_L} \leftarrow r$
    add $r$ to $J_L$, remove $r_{g_L}$ from $J_L$
    $r_{g_L} \leftarrow r$, $g_L \leftarrow k$
    if $k = g_0$ then remove $r_k$ from $J_L$
  else Invert($p_k$)
  return $k$
```

To use this version of ENCAPSULATE to build a linkage with more than two levels of hierarchy requires that the levels be encapsulated from the top down—encapsulating an existing sub-linkage is not allowed. A version of ENCAPSULATE which includes this capability is possible, but requires some more error checking to ensure that all sub-linkages are properly nested. Also, with only ENCAPSULATE and DISSOLVE, to mutate the disposition of a sub-linkage (e.g. from simultaneous to driven) requires re-creating it. A convenience operation to mutate sub-linkage disposition in-place is also possible, and again would consist mostly of verifying that the crossing joints satisfy the topological constraints for the new disposition.

If updated CMTs are available then ENCAPSULATE is $O(|K_S| + |J_S|)$, where K_S and J_S are the link and joint sets in the new sub-linkage. Otherwise it is $O(|K_S|h + |J_S|)$ where h is the maximum spanning tree depth of any link in K_S. The running time analysis for DISSOLVE is the same.

3.7 Summary

This chapter presented a new framework for modeling mixed real/virtual linkages (articulated systems) with particular support for genericity, mutability, scalability, and

```
Algorithm 3.7: ENCAPSULATE($\gamma, \chi, d$)
Input:  non-ground link $\gamma$ in linkage $L$ (i.e. $\gamma \in K_L$)
        set of crossing joints $\chi \subset J_L$
        disposition $d \in \{\text{driving}, \text{driven}, \emptyset\}$
Output: a new sub-linkage of $L$ rooted at $\gamma$
let $K_S = \{\gamma\}$, $J_S = \emptyset$ be the sets of links and joints for the new sub-linkage
 ▷ DFS from $\gamma$, pruning at $\chi$
let $Q \leftarrow \{\text{joint } j \mid p_j = \gamma \text{ or } (c_j = \gamma \text{ and CLOSURE?}(j))\} \setminus \chi$
while $Q \neq \emptyset$ do
    foreach $j \in Q$ in order do
        remove $j$ from $Q$, add $j$ to $J_S$
        if TREE?($j$) then
            add $c_j$ to $K_S$
            append $\{\text{joint } i \mid p_i = c_j \text{ or } (c_i = c_j \text{ and CLOSURE?}(i))\} \setminus \chi$ to $Q$
        end
    end
end
foreach $k \in K_S$ do if $k \notin K_L$ then
  error cannot encapsulate existing sub-linkage
foreach $j \in J_S$ do
    if $j \notin J_L$ then error cannot encapsulate existing sub-linkage
    if $j$ is crossing but does not satisfy constraints in Section 3.5.2 then
      error invalid crossing joint $j$
end
foreach $k \in K_S \setminus \{\gamma\}$ do $p_{r_k} \leftarrow \gamma$, $M_{r_k} \leftarrow \text{CMT}(\gamma)^{-1}\text{CMT}(k)$
$K_L \leftarrow K_L \setminus K_S$, $J_L \leftarrow J_L \setminus J_S$
return $(K_S, J_S, \gamma, L, d)$ ▷ new sub-linkage (Eq. 3.2)
```

```
Algorithm 3.8: DISSOLVE($L$)
Input: non-topmost sub-linkage $L$
Output: parent sub-linkage $p_L$ with the contents of $L$ merged in
foreach $j \in J_L$ do add $j$ to $J_{p_L}$
foreach $k \in K_L$ do add $k$ to $K_{p_L}$, $p_{r_k} \leftarrow g_{p_L}$, $M_{r_k} \leftarrow \text{CMT}(g_{p_L})^{-1}\text{CMT}(k)$
foreach sub-linkage $S$ s.t. $p_S = L$ do $p_S \leftarrow p_L$
return $p_L$
```

hierarchy, all features which become especially important in the presence of virtual links and joints. The first part of the chapter—through Section 3.5—developed the denotational aspects of the model, i.e. *what* a linkage is, with the main structure summarized in Definition 10 and its subsidiaries. The second part of the chapter—Section 3.6—formalized the operational aspects of *how* the topology of a linkage can evolve. Algorithms for computing the motion of linkages will be added in Chapter 4.

The way in which joint mobility is represented (Definition 16) is particularly important: I use a parametrization of $SE(3)$ based on a translation and rotation vector pair $(\boldsymbol{t}, \boldsymbol{\theta})$, with dynamic reparametrization (Appendix B) for $\boldsymbol{\theta}$. This parametrization supports a useful catalog of various joint types (Section 3.3.4), including all lower pairs except helical, and also admits a straightforward way to limit the motion of important kinds of joints (Section 3.3.5).

Also novel is my detailed design for hierarchical linkages (Section 3.5) which can help structure topologically large models, saving time both for the user and the motion computation and simulation algorithms. One interesting use for this kind of hierarchy is in my method of structure abstraction (Section 3.5.3), where a potentially complex sub-linkage is virtually replaced by a simpler interface.

Everything presented in this chapter has been implemented in an integrated software system (Figures 3-1 and 1-4) that serves both as a reference implementation and as the platform upon which the applications presented in Chapters 5 and 6 were tested[12].

[12]The current implementation does not actually include ENCAPSULATE and DISSOLVE, but substitutes a pair routines to attach and detach sub-linkages.

Chapter 4

Linkage Motion and Topological Decomposition Algorithms

Though it is possible to construct rigid[1] linkages (e.g. a planar triangular cycle of three links and three parallel-axis revolute joints), in robotics we are generally interested in linkages that move. This chapter presents algorithms that compute motions for linkages that are represented in the framework introduced in Chapter 3. These algorithms have all been implemented and integrated into the software system used for the applications detailed in Chapters 5 and 6. Figure 4-1 shows the architecture of this implementation, with the aspects covered in this chapter highlighted.

For consistent linkages (recall the terminology defined in Section 3.4), a *feasible* motion is given by a connected path in configuration space. For inconsistent linkages we instead seek a path through the linkage mobility space which minimizes breakage of closure joints; a related problem (for any linkage) is to find a path from a non-configuration to a nearest configuration. In this chapter, these ideas are reduced to two specific problems: *local assembly* (Def. 45) and *differential control* (Def. 48). Algorithms 4.1 and 4.5, Solve and Analyze, work in concert to solve both of these problems in a way that scales to large models with 100s of joints.

[1]Rigidity [56, 37] is actually a fairly subtle concept. When I speak of rigidity, I mean local (infinitesimal) rigidity, Def. 38, unless otherwise stated.

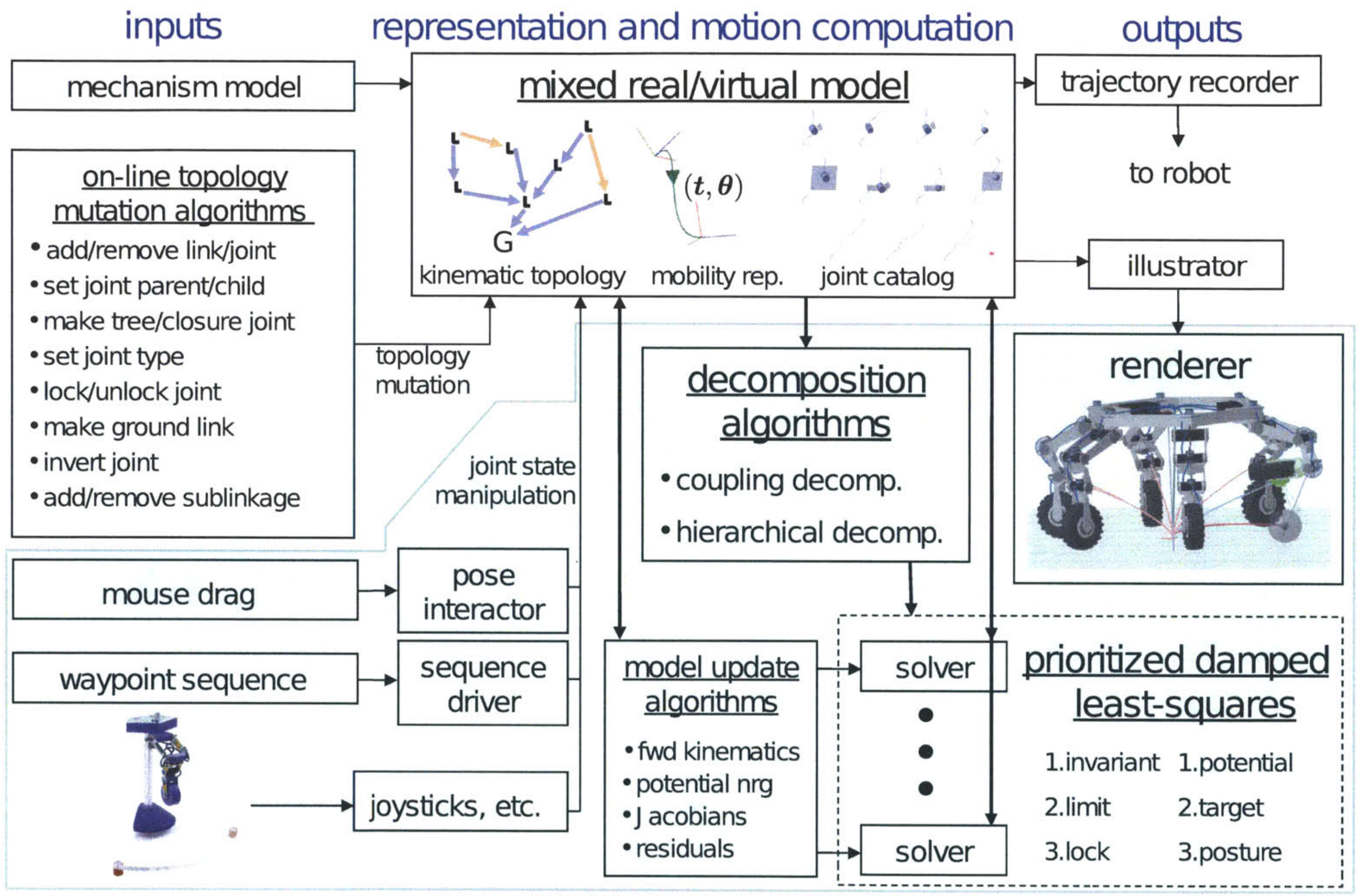


Figure 4-1: Motion-computing aspects of the implementation.

Local assembly comes into play not only in simulating robots that actually change structure, such as self-reconfiguring modular robots, but also for simulation or control of a robot as virtual modifications are added, removed, and reconfigured on-line. Differential control is a general formulation of both forward and inverse-kinematic control, and directly supports various modes of operation for mixed real/virtual structures, including click-and-drag interaction and trajectory following. (For the inverse case, i.e. when the operator requests motion of a link instead of operation of a joint, recall from Section 3.2.3 that link manipulation is equivalent to manipulation of the associated root joint.)

A related problem arises when simulating elastic and gravitational potential energy, which will also be introduced in this chapter (Section 4.6). This *kinostatic* simulation is achieved by continuously seeking to minimize the total potential energy stored in the system.

The SOLVE algorithm uses the differentiability of the $(\boldsymbol{t}, \boldsymbol{\theta})$ representation (Section B.4) to make local linear approximations of the model, and then iteratively solves local assembly, differential control, and kinostatic simulation by gradient descent in a least-squares sense with an SVD-based pseudoinverse. Nullspace projection is used for prioritizing the various constraints, which may conflict. Though this approach is well-known, several new innovations are introduced, including

- a design for six kinds of prioritized constraints—*invariant*, *limit*, *lock*, *potential*, *target*, and *posture* (Section 4.7)—which integrate with the rest of my framework, and which can help resolve motion in over-constrained cases

- an algorithm for analytically computing model Jacobians (Section 4.9.2) which leverages pre-computed composite model transforms (CMTs) and the $(\boldsymbol{t}, \boldsymbol{\theta})$ transform representation

- coordinated heuristics for detecting convergence, stall, and divergence (Section 4.8.2).

Because the time complexity of SOLVE is typically quadratic in the number of joints (Section 4.8.3), it is important to break up large linkages into independently

SOLVE-able components when possible. ANALYZE considers the topology and structure of a linkage, and attempts to break it into smaller parts which can be handled separately. It performs two kinds of decomposition: a *coupling decomposition* separates groups of tree joints supporting distinct sets of closures, and a novel *hierarchical decomposition* splits driving and driven sub-linkages (Section 3.5) from their enclosing linkage so they can be SOLVEd in the appropriate order.

The next two sections give the details of the local assembly and differential control problems and their solution by *prioritized damped least squares* (PDLS). The subsequent sections cover default joint poses, DoF limits, locked joints, and elastic and gravitational potential energy models, building up to a summary of my design for six constraint priority levels in Section 4.7.

The SOLVE algorithm is then presented which implements the PDLS iteration on these priority levels, followed by ANALYZE in Section 4.10 (though hierarchical decomposition is essential to compute motions of driving and driven sub-linkages, ANALYZE is otherwise an optimization—SOLVE can be applied alone to a whole non-hierarchical linkage).

The chapter concludes with a description of a set of *state interaction* modalities (Section 4.11) which complement the structure mutation operations of Chapter 3—whereas those support adding/removing/reconnecting joints, state interaction covers operating and moving existing joints and links.

4.1 Local Assembly by Linear Optimization

Informally, the idea of local assembly (Figure 4-2) is to compute a path from a given tree state $\boldsymbol{x}$ anywhere in the mobility space to a point $\boldsymbol{x}'$ on the configuration space of a linkage (Def. 35). This action is called for when simulating the construction of a linkage containing closed chains: when a chain-closing joint is first attached, or whenever the type, limits, or positioning transforms of any joint within a closed chain are modified, the state immediately after the modification will not necessarily be a configuration—even if it was previously—because the shape of the configuration space itself may have changed. In general the linkage must be re-assembled to a new configuration $\boldsymbol{x}'$ near $\boldsymbol{x}$.

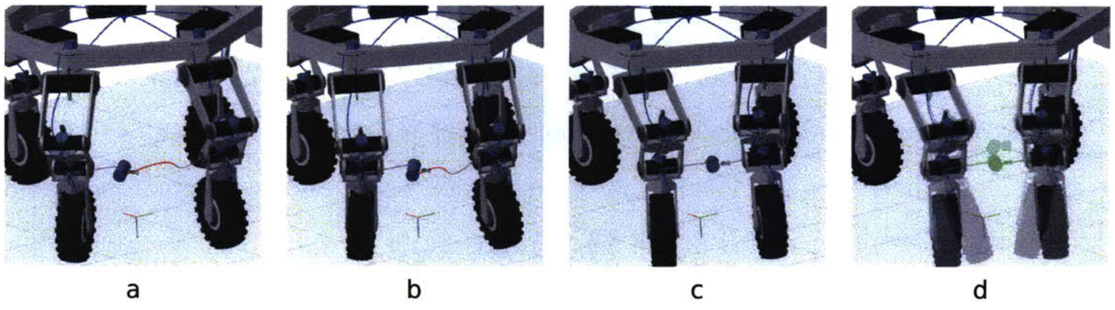


Figure 4-2: The local assembly problem.
In (a), a virtual revolute joint has been added that closes a chain between the "ankles" of two adjacent ATHLETE limbs. Initially, the chain is inconsistent because the relative pose that was specified for the new revolute joint with respect to one of the limbs differs from the actual current relative pose. A red curve indicates the error. The system solves the local assembly problem by moving the legs to remove the error, as shown in (b) and (c). Once the new topology is assembled, it can be operated (d) as originally intended.

To develop a formal definition of local assembly, recall that the forward kinematic mapping FK (Eq. 3.49) and the invariant error projection matrix Π_i (Eq. 3.47) multiply to give the forward invariant error mapping FK_i (Eq. 3.50) taking the current tree state $\boldsymbol{x}$ to the resulting invariant error $\boldsymbol{e}_i$:

$$\text{FK}_i : \mathbb{R}^f \to \mathbb{R}^i \quad \boldsymbol{e}_i = \text{FK}_i(\boldsymbol{x}) = \Pi_i \text{FK}(\boldsymbol{x}). \tag{4.1}$$

If the linkage was consistent, we can assume that prior to the structural change $\boldsymbol{e}_i = \mathbf{0}$ (otherwise we can assume that $\|\boldsymbol{e}_i\|$ was locally minimal). But the structural change may have also induced a change in the invariant error mapping, say from FK_i to FK'_i, and in general this will produce an $\boldsymbol{e}'_i = \text{FK}'_i(\boldsymbol{x})$ s.t. $\|\boldsymbol{e}'_i\| > \|\boldsymbol{e}_i\|$.

Definition 45 Given an initial tree state $\boldsymbol{x}$ of a linkage L, and a structural modification L' of L with forward invariant error mapping FK'_i, the *local assembly problem* is to find a modified tree state $\boldsymbol{x}'$ near $\boldsymbol{x}$ s.t. $\text{FK}'_i(\boldsymbol{x}')$ is locally minimized. The notion of "near" is intentionally left vague; in practice the path from $\boldsymbol{x}$ to $\boldsymbol{x}'$ will be computed as a gradient descent.

The approach I take to solve this problem, first popularized in robotics by Whitney [164] in the 1960s, is a local linear optimization based on the Jacobian pseudoinverse. FK'_i is locally modeled in the neighborhood of $\boldsymbol{x}$ by its Jacobian matrix

$$\underset{[i\times f]}{J'_i(\boldsymbol{x})} = \frac{\partial}{\partial \boldsymbol{x}} \text{FK}'_i(\boldsymbol{x}) \qquad . \tag{4.2}$$

$$\text{FK}'_i(\boldsymbol{x} + \Delta\boldsymbol{x}) \approx \text{FK}'_i(\boldsymbol{x}) + J'_i(\boldsymbol{x})\Delta\boldsymbol{x}. \tag{4.3}$$

Writing $J'_i(\boldsymbol{x})$ stresses the dependence of the Jacobian matrix both on FK'_i and on $\boldsymbol{x}$, but for brevity I will usually write simply J'_i, with the latter dependency implied. J'_i can be computed by multiplying the Jacobian J'_y of the full forward kinematic mapping $\text{FK}'(\boldsymbol{x})$ by the projection matrix Π_i:

$$\text{with } J'_y = \frac{\partial}{\partial \boldsymbol{x}} \text{FK}'(\boldsymbol{x}) \tag{4.4}$$

$$J'_i = \frac{\partial}{\partial \boldsymbol{x}} \text{FK}'_i(\boldsymbol{x}) = \Pi_i \frac{\partial}{\partial \boldsymbol{x}} \text{FK}'(\boldsymbol{x}) = \Pi_i J'_y. \tag{4.5}$$

(In fact, all of the Jacobians in this chapter are based on the Jacobian of the forward kinematic mapping. An algorithm for computing it is given in Section 4.9.2.)

If we set

$$\boldsymbol{x}' = \boldsymbol{x} + \Delta\boldsymbol{x} \tag{4.6}$$

then the local model can be considered a linear system with $\Delta\boldsymbol{x}$ the unknown vector:

$$\begin{aligned} \mathbf{0} = \text{FK}'_i(\boldsymbol{x}') = \text{FK}'_i(\boldsymbol{x} + \Delta\boldsymbol{x}) \approx \text{FK}'_i(\boldsymbol{x}) + J'_i\Delta\boldsymbol{x} = \boldsymbol{e}'_i + J'_i\Delta\boldsymbol{x} \\ -\boldsymbol{e}'_i = J'_i\Delta\boldsymbol{x}. \end{aligned} \tag{4.7}$$

If this model is valid over a large enough neighborhood, and if J'_i is invertible, then we can solve Eq. 4.7 directly for $\Delta\boldsymbol{x}$, e.g. by Gaussian elimination or by inverting J'_i:

$$-J_i'^{-1}\boldsymbol{e}'_i = \Delta\boldsymbol{x}_i \tag{4.8}$$

when the local model is sufficient and invertible.

But generally neither of these assumptions hold: we have to restrict the neighborhood because the configuration space may be arbitrarily curved, and we have to allow non-invertible Jacobians because the problem may be over- or under-constrained.

4.1.1 Iterative Damped Least Squares

We can limit the model to a local neighborhood by clamping the *residual* $\boldsymbol{e}'_i$ to a maximum magnitude, solving, and then reformulating the whole problem at the new tree state $\boldsymbol{x}'$. The solution process thus becomes an iteration, so from here on I drop the prime notation, e.g. writing simply $\boldsymbol{e}_i$ instead of $\boldsymbol{e}'_i$. The iteration is terminated once the residual magnitude falls below some threshold. This addresses the first failed assumption—that the local model is valid for a sufficient neighborhood—but does require setting clamping and termination thresholds, which will be covered below.

The other failed assumption is that J_i is always invertible. To handle the case that it is not, the (Moore-Penrose) pseudoinverse J_i^+ can be substituted. It is well known [83] that this pseudoinverse always exists and solves systems of the form 4.7 in a least-squares optimal sense: in the overconstrained singular case the L_2 (Euclidean) norm of the residual is minimized, in the under-constrained singular case the L_2 norm of the unknown $\Delta\boldsymbol{x}_i$ is minimized, and in the well-constrained non-singular case J_i^+ reduces to J_i^{-1}.

Putting clamping together with the pseudoinverse, Eq. 4.8 can be adapted to

$$-J_i^+\text{CLAMP}(\boldsymbol{e}_i, \gamma_i) = \Delta\boldsymbol{x}_i \tag{4.9}$$

$$\text{with CLAMP}(\boldsymbol{v}, \gamma) = \min(\gamma, \|\boldsymbol{v}\|)\frac{\boldsymbol{v}}{\|\boldsymbol{v}\|} \text{ if } \|\boldsymbol{v}\| > 0, \text{ and CLAMP}(\boldsymbol{0}, \gamma) = \boldsymbol{0}. \tag{4.10}$$

Iterating this computation works—in theory—for any J_i, with two main caveats: the approach is susceptible to getting stuck in local minima, and the speed of convergence is not guaranteed. In fact, local assembly only requires finding a local minimum—the idea is that the operator should configure the linkage near the desired solution before making the topological change which triggers a re-assembly. Convergence is neither too problematic; Section 4.8.3 gives some details.

In practice, the λ-*damped pseudoinverse* $J^{+\lambda}$ is used instead of J^{+} for improved numeric stability in near-singular configurations. Appendix E describes how to compute $J^{+\lambda}$ using the singular value decomposition (SVD).

Definition 46 Iteration 4.9 thus becomes

$$-J_{\imath}^{+\lambda_{\imath}}\text{CLAMP}(\boldsymbol{e}_{\imath}, \gamma_{i}) = \Delta \boldsymbol{x}_{\imath}, \tag{4.11}$$

which is called *damped least squares* (DLS) with residual clamping.

Buss recently gave a good review of this technique in [20], and observes that it was first used (though likely without clamping) in robotics by Wampler [162] and Nakamura and Hanafusa [107].

4.2 Adding Differential Control

We now move from local assembly to the differential control problem, which I formulate as an instance of waypoint following.

The idea of differential control is to find a feasible—or at least invariant error minimizing—motion which tracks a given partial path through the mobility space. Consider the motion of a linkage after a user has moved a link (Figure 4-3). If the link is not part of any closed chains, then the forward kinematic mapping will not be affected. Otherwise, to keep $\|\boldsymbol{e}_{\imath}\|$ (breakage of closure joints) minimized, the system may be required to not (fully) move the link as the user has requested, and/or it may be necessary to move some joints to compensate.

The first step to develop a formal definition of this problem is to form a vector $\boldsymbol{z}$ comprising all the DoF of both the tree and the closure joints in a linkage L. Recall

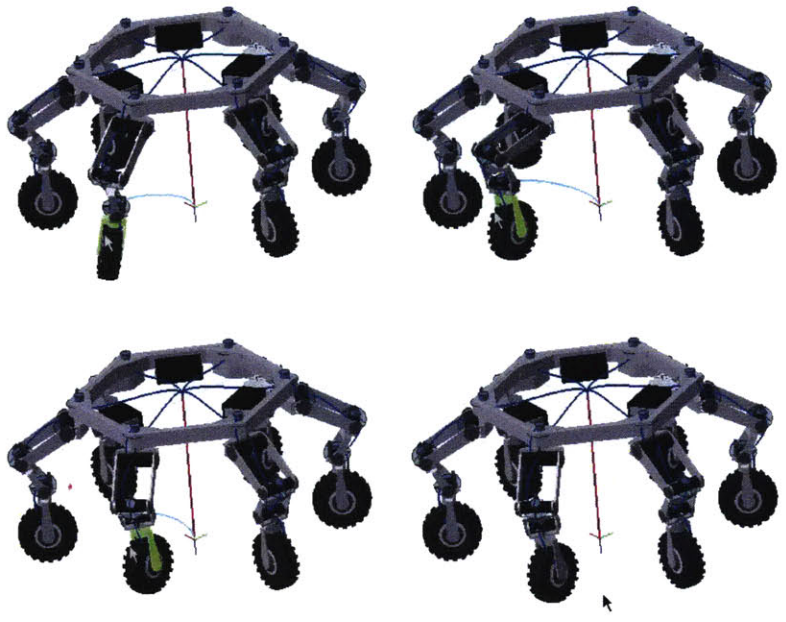


Figure 4-3: The differential control problem.
In this example the operator uses click-and-drag interaction with the mouse to interactively pose the last link in an ATHLETE limb. The hex deck is locked, so the system solves the differential control problem corresponding to the single-limb inverse kinematics.

that Eqs. 3.39 and 3.44 gave specific orderings T and C for the tree and closure joints comprising L, that the tree state $\boldsymbol{x}$ is a concatenation of the f DoF of the tree joints in order (Eq. 3.43), and that the closure state $\boldsymbol{y}$ is a concatenation of the $6|C|$ combined DoF and DoI of the $|C|$ closure joints in order (Eq. 3.45). i was also defined to be the total number of closure joint DoI, and Π_i as an $[i \times 6|C|]$ binary projection matrix selecting only the DoI from $\boldsymbol{y}$.

Let the total number of DoF of the closure joints in L be

$$d = 6|C| - i, \tag{4.12}$$

and define a new $[d \times 6|C|]$ binary projection matrix Π_{fc} which compliments Π_i by selecting only the DoF from $\boldsymbol{y}$. Specifically, each row p of Π_{fc} contains exactly one non-zero entry, say at column q, where q corresponds to the p'th entry in the concatenated sequence of the DoF of all joints in C.

Definition 47 The *total DoF state* $\boldsymbol{z}$ of L is a column vector formed by appending the closure joint DoF $\Pi_{fc}\boldsymbol{y}$ to the tree joint DoF $\boldsymbol{x}$, and Z is the corresponding total ordering of all of the joints in L:

$$z = f + d \tag{4.13}$$

$$\boldsymbol{z}^T = \left(\boldsymbol{x}^T, (\Pi_{fc}\boldsymbol{y})^T\right) \in \mathbb{R}^z \tag{4.14}$$

$$Z = (T, C)\,. \tag{4.15}$$

Definition 48 Given a *target waypoint* pair W_t comprising (1) a z-bit binary vector $\boldsymbol{w}_t$ identifying an axis-aligned subspace (Def. 19 on page 63) of $\mathbb{R}^z$ and (2) a particular point $\boldsymbol{t}$ therein[2]

$$W_t = (\boldsymbol{w}_t \in \{0,1\}^z, \boldsymbol{t} \in \mathbb{R}^t) \text{ with } t = ||\boldsymbol{w}_t||_1, \tag{4.16}$$

the *differential control* problem is to bring the components of $\boldsymbol{z}$ as close as possible to the corresponding components of $\boldsymbol{t}$ without breaking the linkage. The definition of "close" is left vague; in practice the sum of the squared errors are minimized.

The idea is that $\boldsymbol{t}$ represents a current desired *target pose* for a subset of the total DoF $\boldsymbol{z}$ of L; to follow a path, W_t can be periodically replaced by the operator—e.g. by dragging the mouse—with the system either interpolating intermediate waypoints or just switching directly from one to the next.

Let Π_t be a $[t \times z]$ binary projection matrix selecting the t current target DoF from the z total DoF. Π_t is induced by $\boldsymbol{w}_t$ in a manner similar to the previously defined projection matrices.

Definition 49 The *target residual* is

$$\boldsymbol{e}_t = \Pi_t \boldsymbol{z} \overset{s}{-} \boldsymbol{t} \tag{4.17}$$

[2]The notation $||\boldsymbol{w}_t||_1$ indicates the Manhattan norm of $\boldsymbol{w}_t$, i.e. the number of non-zero entries.

where the operation $\overset{s}{-}$ is a difference taken using nearest aliases of the involved rotation vectors. This will be made more precise below in Section 4.9.1, for now consider $\overset{s}{-}$ as equivalent to regular vector subtraction.

Definition 50 The *forward target error mapping* FK_t is

$$\text{FK}_t : \mathbb{R}^f \to \mathbb{R}^t \quad \boldsymbol{e}_t = \text{FK}_t(\boldsymbol{x}) = \Pi_t \begin{bmatrix} \underset{[f\times1]}{\boldsymbol{x}} \\ \underset{[d\times1]}{\Pi_{fc}\text{FK}(\boldsymbol{x})} \end{bmatrix} \overset{s}{-} \boldsymbol{t}, \tag{4.18}$$

Definition 51 The *target Jacobian* is the $[t \times f]$ matrix of partial derivatives of 4.18:

$$\underset{[t\times f]}{J_t} = \frac{\partial \boldsymbol{e}_t}{\partial \boldsymbol{x}} = \frac{\partial}{\partial \boldsymbol{x}}\text{FK}_t(\boldsymbol{x}) = \underset{[t\times z]}{\Pi_t} \begin{bmatrix} \underset{[f\times f]}{\text{I}_f} \\ \underset{[d\times f]}{\Pi_{fc} J_y} \end{bmatrix} \tag{4.19}$$

with I_f the $[f \times f]$ identity matrix.

4.2.1 Residual Compensation and Nullspace Projection

Just as DLS was applied to solve local assembly using the residual vector $\boldsymbol{e}_\imath$ and Jacobian $J_\imath$, it can also be applied to solve differential control by using $\boldsymbol{e}_t$ and J_t. However, separate DLS iterations for the two problems will produce different solution vectors $\Delta\boldsymbol{x}_\imath$ and $\Delta\boldsymbol{x}_t$ which must somehow be combined. The approach I take to forming this combination is based on a *nullspace*[3] *projection* matrix for $J_\imath$, a technique first popularized in robotics in the late 1970s by Liégeois [94], though not specifically for the assembly and control problems as defined here. The idea is to produce a modified $\Delta\boldsymbol{x}_t$ so that the sum

$$\Delta\boldsymbol{x} = \Delta\boldsymbol{x}_\imath + \Delta\boldsymbol{x}_t \tag{4.20}$$

[3]The nullspace of a matrix M is defined as the space of vectors $\boldsymbol{x}$ for which $M\boldsymbol{x} = \mathbf{0}$.

results in an $\boldsymbol{e}_\imath$ that is no larger than would have been produced by $\Delta\boldsymbol{x}_\imath$ alone, but also minimizes $\|\boldsymbol{e}_t\|$ within that constraint. Solution of the assembly problem is thus prioritized above solution of the control problem, i.e., a consistent linkage will always remain unbroken, even if that means sacrificing some fidelity in path following.

This two-level priority scheme, sometimes generally called the *task priority* approach [26], was later extended to an arbitrary number of cascaded priority levels by Siciliano and Slotine in [142], and recently some further improvements were reported by Baerlocher and Boulic in [11]. This last formulation is the basis of the SOLVE algorithm, as it turns out that more than two priority levels are useful in several different contexts. I call this *prioritized* damped least squares (PDLS) and introduce it here by giving the of the two-level solution for the local assembly and differential control problems. Appendix F reviews the generalization to an arbitrary number of priorities given in [11].

The $\Delta\boldsymbol{x}_\imath$ in Eq. 4.20 is taken directly from Eq. 4.11. $\Delta\boldsymbol{x}_t$ is computed by a similar iteration, but with two adjustments: the residual $\boldsymbol{e}_t$ is *compensated* by accounting for the higher-priority effects of $\Delta\boldsymbol{x}_\imath$, and the intermediate result for $\Delta\boldsymbol{x}_t$ is projected onto the nullspace of $J_\imath$:

$$-\left(J_t\left(I - J_\imath^+ J_\imath\right)\right)^{+\lambda_t}\left(\textsc{clamp}(\boldsymbol{e}_t, \gamma_t) + J_t\Delta\boldsymbol{x}_\imath\right) = \Delta\boldsymbol{x}_t. \tag{4.21}$$

The $J_t\Delta\boldsymbol{x}_\imath$ term performs the compensation, and the factor $(I - J_\imath^+ J_\imath)$ accomplishes the nullspace projection [11].

Insight can be gained into Eq. 4.21 by rearranging it to parallel Eq. 4.7 (clamping is omitted for clarity):

$$\begin{aligned} -\left(J_t\left(I - J_\imath^+ J_\imath\right)\right)^{+\lambda_t}\left(\boldsymbol{e}_t + J_t\Delta\boldsymbol{x}_i\right) &= \Delta\boldsymbol{x}_t \\ -\left(\boldsymbol{e}_t + J_t\Delta\boldsymbol{x}_\imath\right) &= J_t\left(I - J_\imath^+ J_\imath\right)\Delta\boldsymbol{x}_t. \end{aligned} \tag{4.22}$$

The residual (left side) of Eq. 4.22 now includes the effect, if any, of the higher-priority adjustment $\Delta\boldsymbol{x}_\imath$ on target following, and the forward model (right side) of the equation now ensures that when an L_2-minimal $\Delta\boldsymbol{x}_t$ is found (recall this minimization is a property of the pseudoinverse) it will have no effect on the closure joint DoI.

Though this two-priority setup does address both of the primary problems—local assembly and differential control—extension to more priorities will help support additional features, including joint locking and potential energy simulation. Also, the multi-priority PDLS implementation in [11] includes two additional capabilities: *posture variation* (Section F.1), and tree joint DoF limits (Section F.2). I apply the former to bias motion towards a default tree pose, and I extend the latter to also handle limits on closure joint DoF.

4.3 Setting a Default Pose with Posture Variation

I leverage the posture variation technique described in [11] (Section F.1) to draw the joints of a linkage toward a lowest-priority default pose, which is useful e.g. in cases where tracking the current target waypoint still leaves some ambiguity in the motion of the linkage. However, [11] only covered the equivalent of tree joint DoF; my formulation, which handles closure as well as tree joints, parallels target tracking (Eqs. 4.16 and 4.17).

Definition 52 The current *posture waypoint* is a pair

$$W_p = (\boldsymbol{w}_p \in \{0,1\}^z, \boldsymbol{p} \in \mathbb{R}^p) \text{ with } p = ||\boldsymbol{w}_p||_1, \tag{4.23}$$

and the *posture residual* $\boldsymbol{e}_p$ is computed[4] via a new $[p \times z]$ binary projection matrix Π_p induced by $\boldsymbol{w}_p$ in the same way that Π_t was induced by $\boldsymbol{w}_t$:

$$\boldsymbol{e}_p = \Pi_p \boldsymbol{z} \overset{s}{-} \boldsymbol{p}. \tag{4.24}$$

However, in this case the Jacobian will not be a $[p \times f]$ matrix, because the posture variation components corresponding to tree joint DoF will be handled directly by

[4]The operation $\overset{s}{-}$ was introduced on page 105.

the final nullspace projection. $\boldsymbol{e}_p$ must be partitioned into two sub-vectors, one for closure joint DoF and one for tree joint DoF, which will be handled separately:

$$\boldsymbol{e}_p^T = \left(\boldsymbol{e}_{pt}^T, \boldsymbol{e}_{pc}^T\right) \tag{4.25}$$

s.t. p_t, p_c are the lengths of $\boldsymbol{e}_{pt}$ and $\boldsymbol{e}_{pc}$, respectively, $p_t + p_c = p$.

Then the posture variation vector is

$$\Delta \boldsymbol{x}_p = -\boldsymbol{e}_{pt}. \tag{4.26}$$

Definition 53 If $p_c > 0$, $\boldsymbol{e}_{pc}$ is the residual vector and J_{pc} the *posture Jacobian* for a third and lowest *posture* priority level, computed by the *forward posture error mapping* FK_{pc}

$$\mathrm{FK}_{pc} : \mathbb{R}^f \to \mathbb{R}^{p_c} \quad \boldsymbol{e}_{pc} = \mathrm{FK}_{pc}(\boldsymbol{x}) = \Pi_{pc}\Pi_{fc}\mathrm{FK}(\boldsymbol{x}) \overset{s}{-} \boldsymbol{p}_c \tag{4.27}$$

$$\underset{[p_c \times f]}{J_{pc}} = \frac{\partial \boldsymbol{e}_{pc}}{\partial \boldsymbol{x}} = \frac{\partial}{\partial \boldsymbol{x}}\mathrm{FK}_{pc}(\boldsymbol{x}) = \Pi_{pc}\Pi_{fc}J_y. \tag{4.28}$$

Π_{pc} and $\boldsymbol{p}_c$, used in the above equations, are taken from corresponding tree/closure partitions of Π_p and $\boldsymbol{p}$:

$$\underset{[p \times z]}{\Pi_p} = \begin{bmatrix} \underset{[p_t \times f]}{\Pi_{pt}} & \\ & \underset{[p_c \times d]}{\Pi_{pc}} \end{bmatrix} \tag{4.29}$$

$$\boldsymbol{p}^T = \left(\boldsymbol{p}_t^T, \boldsymbol{p}_c^T\right). \tag{4.30}$$

Biasing linkage motion towards a default pose involving closure DoF is thus a first use for more than two priority levels.

4.4 Incorporating Limits on Joint DoF

Up to this point, we have glossed over an important constraint imposed by the joint mobility model defined in Chapter 3: if a joint j has defined limits I_j, what mechanism keeps the joint's pose within them? As for posture variation, [11] also presented a technique for handling limits, but only on tree joints (Section F.2). I extend this to also handle limits on closure joints, if any, by adding a fourth *limiting* priority level, this time inserting the new level between the previously defined invariant and target levels. I construct a quadratic penalty to drive limit-exceeding closure DoF back into compliance. No penalty is assigned to in-limits DoF, resulting in a "bathtub"-type differentiable penalty function (Eq. 4.32): while in-limits, the penalty is zero, but upon exceeding a limit the penalty increases quadratically.

Let m be the total number of closure DoF with set limits (recall from Section 3.3.5 that a DoF is always either unlimited or has both upper and lower limits), and let Π_m be a $[m \times d]$ binary projection matrix selecting the m limited DoF from the d total closure joint DoF. Π_m is induced by the sequence of limited closure DoF in the same manner as the previously defined Π_t and Π_p were induced by the binary vectors $\boldsymbol{w}_t$ and $\boldsymbol{w}_p$. Let $\boldsymbol{l}_c$ and $\boldsymbol{u}_c$ be the corresponding vectors of the m lower and upper closure DoF limits, respectively.

Definition 54 The limit EXCESS function is

$$\text{using } \boldsymbol{v}^T = (v_0, \ldots, v_{m-1}),\ \boldsymbol{l}^T = (l_0, \ldots, l_{m-1}),\ \boldsymbol{u}^T = (u_0, \ldots, u_{m-1})$$

$$\textsc{excess}(\boldsymbol{v}, \boldsymbol{l}, \boldsymbol{u}) = \boldsymbol{s} = (s_0, \ldots, s_{m-1})^T \text{ s.t. } s_\imath = \begin{cases} v_\imath - u_i & \text{if } v_\imath > u_\imath \\ 0 & \text{if } u_\imath \geq v_i \geq l_\imath \\ v_\imath - l_\imath & \text{if } v_\imath < l_i. \end{cases} \tag{4.31}$$

Definition 55 The *closure limit residual* vector $\boldsymbol{e}_m$ is the square of the excess for each DoF, and is computed by the *forward limit error mapping* FK_m:

$$\text{FK}_m : \mathbb{R}^f \to \mathbb{R}^m \quad \boldsymbol{e}_m = \text{FK}_m(\boldsymbol{x}) = \text{DIAG}(\boldsymbol{s}_c)\boldsymbol{s}_c \tag{4.32}$$

$$\text{with } \boldsymbol{s}_c = \text{EXCESS}(\Pi_m \Pi_{fc} \text{FK}(\boldsymbol{x}), \boldsymbol{l}_c, \boldsymbol{u}_c)$$

$$\text{and DIAG}(\boldsymbol{v} = (v_0, \ldots, v_{m-1})) = \begin{bmatrix} v_0 & & \\ & \ddots & \\ & & v_{m-1} \end{bmatrix}. \tag{4.33}$$

The *limit Jacobian* is the $[m \times f]$ matrix of partial derivatives of 4.32:

$$\underset{[m \times f]}{J_m} = \frac{\partial \boldsymbol{e}_m}{\partial \boldsymbol{x}} = \frac{\partial}{\partial \boldsymbol{x}} \text{FK}_m(\boldsymbol{x}) = 2\ \text{DIAG}(\boldsymbol{s}_c) \Pi_m \Pi_{fc} J_y. \tag{4.34}$$

By inserting the limit level just below the (highest-priority) invariant level, the solver will enforce joint limits to the greatest extent possible without violating the DoI of any closure joints, and target and posture waypoints will only be pursued to the extent possible without exceeding any DoF limits.

4.5 Locking Joints

It is often useful to temporarily hold the current pose of any joint. I call this *locking* the joint, and implement it in a way similar to target tracking for closure joints, and similar to limiting for tree joints. A new *lock* priority level is inserted just below the limit level that drives locked closure joints to hold the DoF state they had when originally locked. Locked tree joints, on the other hand, are removed entirely from the problem, in this case by treating them as if they temporarily had type **F** (fixed, i.e. zero DoF), with their mobility transform frozen. (In the implementation, tree joints are not actually set to type **F**, but are handled topologically as part of the ANALYZE algorithm. This will be detailed below.)

Let Γ be an ordering of the set of locked joints, partitioned into sub-sequences of tree and closure joints,

$$\Gamma \subset J = (\Gamma_t, \Gamma_c) \text{ with } \Gamma_t = \Gamma \cap T \text{ and } \Gamma_c = \Gamma \cap C, \tag{4.35}$$

where $J = T \cup C$ is the tree/closure partitioning of the joints in the linkage,

let k be the total number of DoF of all joints in Γ, and let k_t and k_c be the total number of DoF of the joints in Γ_t and Γ_c, respectively, so $k = k_t + k_c$. Finally, let $\boldsymbol{k}$ be a vector of the values of the k_c locked closure DoF at time of locking.

A $[k_c \times d]$ binary projection matrix Π_k is induced by Γ_c, similar to the way Π_{pc} was induced by $\boldsymbol{w}_p$ for posture variation, that in this case selects the k_c locked DoF out of all closure DoF.

Definition 56 The *lock residual* vector $\boldsymbol{e}_k$ is computed[5] by the *forward lock error mapping* FK_k

$$\text{FK}_k : \mathbb{R}^f \to \mathbb{R}^k \quad \boldsymbol{e}_k = \text{FK}_k(\boldsymbol{x}) = \Pi_k \Pi_{fc} \text{FK}(\boldsymbol{x}) \overset{s}{-} \boldsymbol{k}, \tag{4.36}$$

and the *lock Jacobian* is the $[k_c \times f]$ matrix of partial derivatives of 4.36:

$$\underset{[k_c \times f]}{J_k} = \frac{\partial \boldsymbol{e}_k}{\partial \boldsymbol{x}} = \frac{\partial}{\partial \boldsymbol{x}} \text{FK}_k(\boldsymbol{x}) = \Pi_k \Pi_{fc} J_y. \tag{4.37}$$

That handles locking for closure joints: the solver will attempt to freeze the DoF of locked closures to the extent that goal does not conflict with maintaining both the invariant and limit constraints, but compromising in target and posture tracking if necessary.

4.6 Elastic and Gravitational Potential Energy

The framework presented so far is purely kinematic—there is no representation of physical forces or energy. In some contexts, this is both sufficient and desirable, but in other cases some modeling of physics is necessary to capture the essence of

[5]The operation $\overset{s}{-}$ was introduced on page 105.

the problem. This section will detail the contents of Υ_j, the optional container for any parameters relating to physical force and energy modeling for joint j, and the computation of the motion of systems involving these quantities—simply by adding another PDLS level. As introduced in Section 3.1, I have chosen only to handle a subset of physical parameters which relate to the *quasistatic* effects of joint elasticity and link mass, i.e. elastic and gravitational potential energy (EPE and GPE, respectively).

When is Υ_j needed? A pure kinematic model is is sufficient whenever the robot has enough actuation to enable the active operation of any feasible trajectory.

Definition 57 In this context, a robot is *fully actuated* for a task if any path through the configuration space (of the robot plus any virtual articulations representing the task) can be actively controlled on the robot by its own actuators.

The high-DoF operations examples in Chapter 5 are all fully actuated in this sense, and utilize pure kinematic models. Pure kinematic modeling is desirable from at least two perspectives. First, it can be less computationally expensive to compute motions for pure kinematic models because they can often be decoupled into smaller sub-systems than if phyisics was involved. Specifically, CONSTRAINED? (page 133), needs to categorize an otherwise un-constrained closure joint j as constrained if it has $\Upsilon_j \neq \emptyset$; and TRACESUPPORTS (page 135) will then include the supports of j in ANALYZE, which in turn may not be able to produce as fine of a coupling decomposition as may have otherwise been possible.

Second, it faster and simpler for an operator to specify and modify a purely kinematic model, simply because there are no physics-related parameters to be defined. This becomes especially important in the mixed physical/virtual interface context when virtual joints are added and removed—the extra time required to specify (and to tune) their physics parameters could be significant vs. the time required to simply position and attach them kinematically.

But since pure kinematic modeling would not be sufficient for the kinds of proprioceptive/compliant locomotion systems studied in Chapter 6, I added the ability to model the essential physics of those systems. Because they are all *statically stable*—if actuation is stopped at any time, all motion of the system will also stop—all that is required is to model the potential energy stored in the physical system (i.e. kinetic energy can be ignored). The relevant potential energy storage breaks into just two components: energy stored in elastic joints, and the gravitational potential energy of links with non-negligible mass.

4.6.1 Modeling Elastic Potential Energy

Modeling elasticity is an interesting subject (e.g. [31]), and similar to the case for joint limits (Section 3.3.5), a fully general model that works for all joint types would be somewhat involved. But in practice, supporting just a subset of the possibilities is sufficient for a wide array of common situations. I choose to model *linear spring* elasticity for 1-DoF (i.e., revolute and prismatic) joints only (in fact, the only elastic joints in any of the examples in Chapter 6 are revolute, and are reasonably approximated as linear springs).

The elastic potential energy S_j stored in such a joint depends on three values: the current state d of the joint's DoF, a *stiffness factor* κ_j, and the *stiffness rest* state ξ_j:

$$S_j = \frac{1}{2}\kappa_j(d - \xi_j)^2. \tag{4.38}$$

This results directly from an integration of the linear spring law $f = kx$ where f is the spring restoring force, k is the spring constant, and x is the deflection of the spring. S_j will always be non-negative provided that $\kappa_j \geq 0$.

4.6.2 Modeling Gravitational Potential Energy

Unlike joint elasticity, it is not hard to model Gravitational potential energy in the general case. Technically, gravity affects links, not joints, as links model the solid parts of a robot which would have mass. But because I include a special virtual root joint r_l for every link l (Section 3.2.3), it is trivial to bookkeep the potential energy in Υ_{r_l}. This avoids adding extra complexity to the algorithms, and is another example of the homogenizing capabilities of virtual articulations.

For the root joint r_l of a link l (GPE is only allowed on root joints), the gravitational potential energy G_{r_l} depends on four values: the current joint transform X_{r_l} (which is also the CMT of l), the *center of mass* (CoM) $\boldsymbol{c}_{r_l}$ of l in its link frame, the *mass* m_{r_l} of l, and the *gravity acceleration vector* $\boldsymbol{g}$, which is a new global constant:

$$G_{r_l} = -m_{r_l}\boldsymbol{g} \cdot (X_{r_l}\boldsymbol{c}_{r_l}) = -m_{r_l}\boldsymbol{g}^T (X_{r_l}\boldsymbol{c}_{r_l}) . \tag{4.39}$$

This is an instantiation of the basic formula mgh for the gravitational potential of a solid body, where m is the mass, g is the gravitational constant, and h is the height of the body. In this formulation, g is encoded as the length of $\boldsymbol{g}$, and $\boldsymbol{g}/\|\boldsymbol{g}\|$ defines the "downward" direction in the global (ground link) frame. $(X_{r_l}\boldsymbol{c}_{r_l})$ is the transformation of the link CoM to this same global frame, so the dot product in Eq. 4.39 effectively computes "$-gh$" (negative because $\boldsymbol{g}$ is down, not up). G_{r_l} will be non-negative as long as $m_{r_l} \geq 0$ and $h = -(\boldsymbol{g}/\|\boldsymbol{g}\|) \cdot (X_{r_l}\boldsymbol{c}_{r_l}) \geq 0$; thus the computation is incorrect (in reality GPE can never be negative) if h is negative; in practice it is not too hard to structure the operation of the model so that this is avoided.

4.6.3 Formal Definition of Υ_j

Definition 58 Υ_j is a pair with different contents depending on whether EPE or GPE is represented (a single joint can have either EPE or GPE, but not both):

$$\Upsilon_j = \begin{cases} (\kappa_j, \xi_j) & \text{if } Y_j \in \{\mathbf{R}, \mathbf{P}\} \text{ and } j \text{ has elasticity} \\ (\boldsymbol{c}_j, m_j) & \text{if } j \text{ is the root joint of a link } l \text{ with mass} \\ \emptyset & \text{otherwise (no potential energy)} \end{cases} \tag{4.40}$$

with $\kappa_j \geq 0$ the stiffness factor, ξ_j the stiffness rest state

$\boldsymbol{c}_j$ the CoM of l in link frame, and $m_j \geq 0$ the link mass.

4.6.4 Minimizing Total Potential Energy

To simulate the motion of a linkage involving joints with $\Upsilon_j \neq \emptyset$, it is sufficient to add a PDLS priority level with residual equal to the sum of the total potential energy in the system. I insert this level between the previously defined lock and target levels, so that joint invariants, limits, and locked joints have higher priority than potential energy minimization, but target tracking and joint default poses (posture) have lower priority.

Let $S \subset J$ be the set of non-root joints with $\Upsilon_j \neq \emptyset$, i.e. the set of all joints with elastic potential energy models, and $G \subset J$ be the set of all root joints with $\Upsilon_j \neq \emptyset$, i.e. the set of all joints with gravitational potential energy models. Then the total potential energy E in the system is

$$E = \left(E_S = \sum_{j \in S} S_j \right) + \left(E_G = \sum_{j \in G} G_j \right). \tag{4.41}$$

Unlike previous PDLS levels, where the residual was a vector, in this case the residual is simply the scalar E.

Definition 59 For conformity with the other levels, the *potential energy residual* $\boldsymbol{e}_v$ is a 1-vector, i.e.

$$\boldsymbol{e}_v = (E), \tag{4.42}$$

unless there are no joints with $\Upsilon_j \neq \emptyset$, in which case $\boldsymbol{e}_v = \emptyset$ (i.e. the potential energy level is skipped entirely).

Since all Υ_j parameters (and $\boldsymbol{g}$) are constant in the context of SOLVE, the only variables affecting E are functions of the tree state $\boldsymbol{x}$.

Definition 60 The *potential energy Jacobian* J_v is computed as the sum of two $[1 \times f]$ Jacobians, one for EPE and the other for GPE:

$$\begin{aligned} \underset{[1\times f]}{J_v} &= \frac{\partial E}{\partial \boldsymbol{x}} = \frac{\partial E_S}{\partial \boldsymbol{x}} + \frac{\partial E_G}{\partial \boldsymbol{x}} \\ &= \underset{[1\times f]}{J_S} + \underset{[1\times f]}{J_G}. \end{aligned} \tag{4.43}$$

To derive J_S, recall Eq. 4.13 which defined $\boldsymbol{z}$ as the vector of combined tree and closure DoF. Then

$$J_S = \frac{\partial E_s}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \frac{\partial E_s}{\partial \boldsymbol{z}}\begin{bmatrix} \underset{[f\times f]}{\mathrm{I}_f} \\ \underset{[d\times f]}{\Pi_{fc}J_y} \end{bmatrix} \tag{4.44}$$

and $\partial E_S/\partial \boldsymbol{z}$ is a $[1 \times z]$ vector filled with zeros except at indices corresponding to the single DoF of joints j with defined EPE. The value at such an index is the derivative of Eq. 4.38 with respect to that DoF value:

$$\frac{\partial S_j}{\partial d} = \kappa_j(d - \xi_j). \tag{4.45}$$

J_G can be derived in a similar form. Let Π_{ft} be a new $[f \times 6|T|]$ binary projection matrix selecting only the f DoF from the full combined $6|T|$ DoF and DoI of all tree joints (Π_{fc} performs the same projection, but for closure joints).

Definition 61 The *total combined DoF and DoI* vector $\boldsymbol{\psi}$ is a column vector formed by appending the closure state $\boldsymbol{y}$ to the expansion of the tree state $\boldsymbol{x}$ by projection through Π_{ft}^T:

$$\underset{[1\times 6(|T|+|C|)]}{\boldsymbol{\psi}^T} = \left(\left(\Pi_{ft}^T\boldsymbol{x}\right)^T, \boldsymbol{y}^T\right). \tag{4.46}$$

Then

$$J_G = \frac{\partial E_G}{\partial \boldsymbol{\psi}}\frac{\partial \boldsymbol{\psi}}{\partial \boldsymbol{x}} = \frac{\partial E_G}{\partial \boldsymbol{\psi}}\begin{bmatrix} \underset{[6|T|\times f]}{\Pi_{ft}^T} \\ \underset{[6|C|\times f]}{J_y} \end{bmatrix} \tag{4.47}$$

and $\partial E_G/\partial \boldsymbol{\psi}$ is a $[1 \times 6(|T| + |C|)]$ vector filled with zeros except at indices corresponding to the 6 DoF of a root joint r_l with defined GPE. The sub-vector at such a location is the derivative of Eq. 4.39 with respect to the 6 DoF, $\partial G_{r_l}/\partial X_{r_l}$. Noting that transformation of $\boldsymbol{c}_{r_l}$ by X_{r_l} can be expanded as

$$\text{with } X_{r_l} = (\boldsymbol{t}_{r_l}, \boldsymbol{\theta}_{r_l})$$
$$G_{r_l} = -m_{r_l}\boldsymbol{g}^T\left(\textsc{rot}(\exp(\boldsymbol{\theta}_{r_l}))\boldsymbol{c}_{r_l} + \boldsymbol{t}_{r_l}\right), \tag{4.48}$$

where exp produces a unit quaternion from a rotation vector (Appendix B) and ROT produces a $[3 \times 3]$ rotation matrix from a unit quaternion (Appendix C),

$$\frac{\partial G_{r_l}}{\partial X_{r_l}} = \underset{[1\text{x}6]}{\frac{\partial G_{r_l}}{\partial(\boldsymbol{t}_{r_l}, \boldsymbol{\theta}_{r_l})}} = \left(\underset{[1\text{x}3]}{\frac{\partial G_{r_l}}{\partial \boldsymbol{t}_{r_l}}}, \underset{[1\text{x}3]}{\frac{\partial G_{r_l}}{\partial \boldsymbol{\theta}_{r_l}}} \right)$$
$$= \left(\underset{[1\text{x}3]}{-m_{r_l}\boldsymbol{g}^T}, \underset{[1\text{x}3]}{-m_{r_l}\boldsymbol{g}^T} \underset{[[3\text{x}3]\text{x}4]}{\frac{\partial \text{ROT}(\exp(\boldsymbol{\theta}_{r_l}))}{\partial \exp(\boldsymbol{\theta}_{r_l})}} \underset{[4\text{x}3]}{\frac{\partial \exp(\boldsymbol{\theta}_{r_l})}{\partial \boldsymbol{\theta}_{r_l}}} \underset{[3\text{x}1]}{\boldsymbol{c}_{r_l}} \right). \tag{4.49}$$

4.7 The Six Priority Levels

The six major sections above introduced a set of six ordered priority levels: invariant, limit, lock, potential, target, and posture (the order of the sections was different). Table 4.1 summarizes these and gives cross-references to the definitions for the corresponding quantities. Each level l is generically defined by its residual vector $\boldsymbol{e}_l$, its *height* h_l, which is the length of $\boldsymbol{e}_l$, and its $[h_l \times f]$ Jacobian matrix J_l. These are, in turn, induced by the structures described above. For example, the current target waypoint W_t induces $\boldsymbol{e}_t$ and J_t. Note that h_l may be zero for each level individually—for example if there are no locked closure joints then the height of the lock level would be zero. This effectively disables the level.

level l	induced by	height h_l	residual $\boldsymbol{e}_l$	Jacobian J_l	page
invariant	C (Eq. 3.44)	i	$\boldsymbol{e}_i$ (Eq. 4.1)	J_i (Eq. 4.5)	98
limit	C (Eq. 3.44)	m	$\boldsymbol{e}_m$ (Eq. 4.32)	J_m (Eq. 4.34)	109
lock	Γ_c (Eq. 4.35), $\boldsymbol{k}$	k_c	$\boldsymbol{e}_k$ (Eq. 4.36)	J_k (Eq. 4.37)	110
potential	$\Upsilon_\jmath$ (Eq. 4.40)	v	$\boldsymbol{e}_v$ (Eq. 4.42)	J_v (Eq. 4.43)	111
target	W_t (Eq. 4.16)	t	$\boldsymbol{e}_t$ (Eq. 4.18)	J_t (Eq. 4.19)	105
posture	W_p (Eq. 4.23)	p_c	$\boldsymbol{e}_{pc}$ (Eq. 4.27)	J_{pc} (Eq. 4.28)	107

Table 4.1: The six SOLVE priority levels, in order from highest to lowest.

4.7.1 Sub-Priorities

Sometimes it is useful to have an even finer partitioning of priorities—that is, to enforce some constraints within a single level above others. This is easily done by labelling each constraint with a *sub-priority* in $\mathbb{Z}^{0+}$ so that a partial order is established on constraints within the level. For clarity, this detail is skipped in the pseudocode for SOLVE, but it is available in the implementation.

4.8 The SOLVE Algorithm

A priority loop solving these six tasks forms the core of SOLVE, Algorithm 4.1. SOLVE also includes an enclosing limiting loop (priority and limiting loops are introduced in Appendix F) and finally a third, outermost *convergence loop*, which nominally iterates until the magnitude of the total residual at each level l is below a corresponding convergence threshold ϵ_l:

$$\forall_l \; \|\boldsymbol{e}_l\| < \epsilon_l. \tag{4.50}$$

It is possible that convergence could take many iterations, so there is a check to force termination of the convergence loop after a maximum amount of wall-clock time has passed. It is also possible that progress could grind to a halt before the convergence threshold is met, which would indicate that SOLVE has become stalled in a local minimum. Finally, is even possible that SOLVE could diverge—for example, if slightly reducing the residual of a high-priority level requires a large compromise in the residual of a lower-priority level. SOLVE may be terminated in that case as well. The full details of detecting convergence, stall, and divergence are given below.

In an interactive simulation context, SOLVE is automatically run after any change to model structure or state, and periodically thereafter as long as the system remains un-converged. This top-level sequencing is handled by the main simulation loop (Section 4.12), which also interleaves other tasks, such as rendering, and any structure and state updates requested by the user.

While SOLVE could process an entire linkage, taking a set of inputs delineating a *subset* of a linkage is more flexible. This is mainly in preparation for the coupling and hierarchical decompositions in Section 4.10, which will attempt to split a linkage into smaller independently solvable pieces, but also serves two other purposes:

- Tree joints not in the support of any closure joint (Def. 17) never need to be treated by SOLVE. By design, the only time the system might need to change the current state of such a joint is if the user changes the target, posture, or quasistatic parameters associated with it (recall that the structure mutation algorithms for changing its connectivity, type, and limits internally handle any incurred state changes). These changes can be easily made outside of SOLVE in the main simulation loop. Design choices, such as redirecting click-and-drag interaction on a link to operation of its root joint, and similar handling in the quasistatic extensions for link mass and gravity, ensure that tree joints which support no closures are truly independent.

- Locked tree joints, i.e. those listed in Γ_t, are *suppressed* and effectively invisible to SOLVE.

Figure 4-4 illustrates the partition of a linkage into the part fed to SOLVE and the remainder. The SOLVE input parameters C_s, T_s, and $\boldsymbol{x}_s$ are restricted to only the

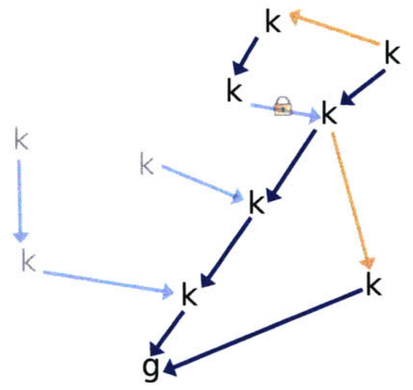


Figure 4-4: The part of a linkage processed by SOLVE.
Locked tree joints and all open-chains are suppressed (faded out in this figure).

part of the linkage that should be processed. Specifically, if the sets of closure and tree joints in the full flattened (Def. 40) linkage are C and T and the full tree state is $\boldsymbol{x}$, then $C_s \subset C$, $T_s \subset T$ is the set of unlocked tree joints supporting the closures in C_s, and $\boldsymbol{x}_s$ contains the f_s concatenated DoF of the tree joints in T_s.

Forming the restricted inputs C_s, T_s, and $\boldsymbol{x}_s$ involves some topological processing on the full linkage, e.g. separating out joints that are not part of any closed chains.

This will be taken care of in ANALYZE. For now, assume that $C_s = C$, that T_s is the union of all unlocked closure joint supports, and that $\boldsymbol{x}_s$ has been similarly restricted from $\boldsymbol{x}$. ANALYZE will also organize the correct handling of hierarchical linkages.

C_s, and the mobility models of its included joints, induce correspondingly restricted invariant and limit levels. The remaining inputs to SOLVE induce the lower four priority levels in table 4.1. We can also consider these to be correspondingly restricted, which requires only a little extra bookkeeping in the implementation. These restrictions are implied.

With its context now firmly defined, SOLVE is presented as Algorithm 4.1. Some of the finer aspects, such as the approaches I take to determining λ_l and γ_l, and to detecting termination of the convergence loop, are given next, followed by an analysis of both the asymptotic and practical time costs.

4.8.1 Adaptive Step Size and Damping

The step size γ_l and damping λ_l parameters are critical to the performance of SOLVE—too large a step or too little damping can cause instability, but small stepsize and high damping conversely lead to slow convergence. Furthermore, optimal values for these parameters depend both on a chosen metric and, significantly, on the particular structure and state of the model at hand. For example, higher damping values may be warranted when the model is in a near-singular configuration (i.e. with some singular value approaching zero), and lower values otherwise.

These issues have been considered before. For example, Deo and Walker introduce optimality criteria to compute damping factors [39]. I use a simpler heuristic approach for λ_l presented by Maciejewski and Klein in [97] and also recommended in [11]. For γ_l I present a novel adaptive speedup/slowdown approach based on heuristics that integrate with both the λ_l computations and with divergence detection (adaptive stepsize approaches in general are common in this kind of numerical computation, e.g. [85]).

The idea in [97] for automatically computing λ_l is to approximate the worst-case scaling effect that the smallest non-zero singular value σ_{min} could have on the

Algorithm 4.1: $\textsc{Solve}(C_s, T_s, \boldsymbol{x}_s, \Gamma_c, \boldsymbol{k}, \Upsilon, W_t, W_p)$

```
$\triangleright$ adapted from [11]
Input:  closure joints $C_s$, unlocked tree joints $T_s$ supporting $C_s$ (see text),
        current tree state $\boldsymbol{x}_s$ corresponding to the $f_s$ DoF of $T_s$,
        locked closure joints $\Gamma_c$ and corresp. frozen DoF values $\boldsymbol{k}$,
        potential energy model $\Upsilon$,
        target and posture waypoints $W_t, W_p$
Output: a new tree state $\boldsymbol{x}_s$
let $\boldsymbol{l}_t, \boldsymbol{u}_t$ be vectors of the tree joint DoF limits or $\pm\infty$ for unlimited DoF
repeat $\triangleright$ convergence loop
    compute all $\boldsymbol{e}_l$ and $J_l$ induced by Solve parameters at state $\boldsymbol{x}_s$
    check for convergence, divergence, stall, or timeout
    adapt all level clamping thresholds $\gamma_l$
    let $\Delta\boldsymbol{x}_s \leftarrow \boldsymbol{0}$, set all DoF unpinned
    repeat $\triangleright$ limiting loop (Section F.2)
        let $N \leftarrow \mathrm{I}_{f_s}$, foreach $0 \le i < f_s$ do if DoF $i$ pinned then $N[i,i] \leftarrow 0$
        foreach level $l$ from highest to lowest do $\triangleright$ priority loop (Ch. F)
            $\boldsymbol{e}_c \leftarrow \mathrm{CLAMP}(\boldsymbol{e}_l, \gamma_l) + J_l\Delta\boldsymbol{x}_s$, $J_r \leftarrow J_l N$
            adapt level damping factor $\lambda_l$
            $U\Sigma V^T \leftarrow \mathrm{SVD}(J_r)$, $J_r^+ \leftarrow V\Sigma^+U^T$, $J_r^{+\lambda_l} \leftarrow V\Sigma^{+\lambda_l}U^T$
            $\Delta\boldsymbol{x}_s \leftarrow \Delta\boldsymbol{x}_s - J_r^{+\lambda_l}\boldsymbol{e}_c$, $N \leftarrow N - J_r^+ J_r$
        end
        $\Delta\boldsymbol{x}_s \leftarrow \Delta\boldsymbol{x}_s - N\mathrm{CLAMP}(\boldsymbol{e}_{pt}, \gamma_p)$  $\triangleright$ apply posture variation
        let $\boldsymbol{s}_t \leftarrow \mathrm{EXCESS}(\boldsymbol{x}_s + \Delta\boldsymbol{x}_s, \boldsymbol{l}_t, \boldsymbol{u}_t)$
        foreach $0 \le i < f_s$ do if $\boldsymbol{s}_t[i] \ne 0$ then DoF $i$ is pinned
    until no newly pinned DoF
    $\boldsymbol{x}_s \leftarrow \boldsymbol{x}_s + \Delta\boldsymbol{x}_s$  $\triangleright$ dynamically reparametrizing (Section B.5) as necessary
    foreach $0 \le i < f_s$ do if DoF $i$ pinned then set $\boldsymbol{x}_s[i]$ to corresp. limit
until converged, diverged, stalled, or timeout
return $\boldsymbol{x}_s$
```

magnitude of $\Delta \boldsymbol{x}_s$, and then adjust λ_l accordingly to keep this magnitude below some threshold δ_{max}

$$\lambda_l = \begin{cases} 0 & \text{if } \sigma_{min} < \|\boldsymbol{e}_c\|/\delta_{max} \\ \sqrt{\sigma_{min}(\|\boldsymbol{e}_c\|/\delta_{max} - \sigma_{min})} & \text{otherwise} \end{cases} . \tag{4.51}$$

Effectively, larger λ_l leads to smaller $\|\Delta \boldsymbol{x}_s\|$, and vice-versa. In the current implementation δ_{max} is exposed as part of the problem specification; values in the range 0.0025 to 0.025 gave good results for the models presented in this thesis.

Whereas λ_l gives a measure of control over the magnitude of the output $\Delta \boldsymbol{x}_s$ of the locking loop, the stepsize γ_l controls the other end of the process, limiting the magnitude of the input residual vectors $\boldsymbol{e}_l$. I use an adaptive approach where each γ_l is initialized to ∞ and then adjusted on-line as indicated from the behavior of the prior iteration of the convergence loop. A *slowdown* is triggered whenever

- $\|\Delta \boldsymbol{x}_s\|$ exceeds a threshold $\beta_{slowdown}\delta_{max}$ proportional to the intended maximum
- or any λ_l exceeds a threshold $\lambda_{slowdown}$
- or the system is diverging at one of the three highest priority levels—invariant, limit, or lock (divergence detection is detailed below).

The mechanism for reducing the γ_l in a slowdown is to first initialize any γ_l that are still ∞ to the current corresponding $\|\Delta \boldsymbol{x}_s\|$, and then to scale all γ_l by a slowdown factor $\alpha_{slowdown} < 1$, clamping each to a minimum limit of γ_{min}.

Conversely, a *speedup* is triggered whenever

- $\|\Delta \boldsymbol{x}_s\|$ is below a threshold $\beta_{speedup}\delta_{max}$ proportional to the intended maximum
- and all λ_l are less than a threshold $\lambda_{speedup}$
- and the system is not diverging at any level.

A speedup is effected by scaling all γ_l by a speedup factor $\alpha_{speedup} > 1$, clamping each to a maximum limit of γ_{max}.

Like δ_{max}, the various speedup/slowdown parameters are also considered tuning parameters. They are summarized in table 4.2 along with the values used for the examples presented in this thesis.

parameter	value for robot operations examples (Chapter 5)	value for quasistatic analysis examples (Chapter 6)
$\beta_{slowdown}$	2.0	2.0
$\beta_{speedup}$	0.2	0.2
$\lambda_{slowdown}$	1.0	1.0
$\lambda_{speedup}$	0.15	0.15
$\alpha_{slowdown}$	0.5	0.5
$\alpha_{speedup}$	2.0	2.0
γ_{min}	0.001	0.00001
γ_{max}	0.5	0.01

Table 4.2: Adaptive stepsize parameters.

4.8.2 Detecting Termination

The termination checks for the convergence loop also depend on various parameters, and, in the case of divergence and stall, on any recent speedup/slowdown events.

I detect convergence by comparing the residual magnitude $\|e_l\|$ at each level to a given threshold ϵ_l (Eq. 4.50). In the current implementation I leave this threshold as part of the problem specification, and I also allow the operator to use the Manhattan norm instead of the Euclidean norm, with the reasoning that for a robot we may sometimes care most about the maximum error of any individual joint.

The actual value of the convergence thresholds will depend on the geometric and topological scale of each specific model, and also on the desired trade-off between simulation accuracy and speed. I have found that for most of the examples related to operating robots in this thesis (Chapter 5), which deal with robots with geometric scales on the order of 10s–1000s of cm, an ϵ_l value of 0.01 under a Manhattan norm was sufficient. The quasistatic models in Chapter 6 required smaller thresholds of

0.001 to 0.00001 to give sufficient accuracy and repeatability in the simulations, since in those cases millimeter-scale motion was significant.

Divergence is detected in a manner similar to convergence, but in this case we check for levels where $\|\boldsymbol{e}_l\|$ has increased from one iteration to the next. Also, such divergence is permitted for the lowest three priority levels—potential, target, and posture—since the operator might validly change the higher-priority constraints—joint invariants, limits, or locking—in ways that conflict directly with trajectory following or the minimum attainable kinostatic potential energy. Two constants are involved in the detection of divergence for any level: a tolerance μ_l, and a scale factor η_l. The level residual magnitude $\|\boldsymbol{e}_l\|$ is computed the same as for convergence checking, but now we also require the magnitude at the prior level, say $\|\boldsymbol{e}_{lprev}\|$. Level l is diverging iff (a)

$$\|\boldsymbol{e}_l\| > \mu_l + \eta_l \|\boldsymbol{e}_{lprev}\|, \tag{4.52}$$

and (b) there was no adaptive stepsize speedup or slowdown in the prior convergence iteration. As above, μ_l and η_l are specified as part of the problem setup; the values $\mu_l = 0.001$ and $\eta_l = 1.01$ sufficed for all the examples I have considered.

Finally, stall is detected when (a) the magnitude $\|\Delta\boldsymbol{x}_s\|$ of the update vector is less than a threshold δ_{min}, and (b) there was no adaptive stepsize speedup in the prior convergence iteration. δ_{min} is also considered part of the problem specification in the current implementation, and I again allow the operator to substitute the Manhattan for the Euclidean norm. For all the systems I considered δ_{min} values in the range 0.001 to 0.01 were sufficient under a Manhattan norm.

4.8.3 Time Complexity and Convergence

Assuming dense matrices, the most asymptotically expensive operations in the priority loop (the innermost loop) for any particular level l are (1) the $O(h_l f_s^2)$ operations spent in matrix multiplies to compute J_r and to update N, and (2) the SVD and pseudoinverse computations. See Appendix E for more on the cost of computing the SVD; briefly, these computations are also $O(h_l f_s^2)$ when $h_l \geq f_s$, and $O(h_l^2 f_s)$ otherwise. $h_l > f_s$ corresponds to an over-constrained priority level, and $h_l < f_s$ to an under-constrained level. Both are possible, but under-constraint is the common case, again because in robotics applications we typically study linkages that have some motion freedom. Thus the time complexity of each iteration of the the inner loop is generally $O(h_l f_s^2 + h_l^2 f_s)$, but commonly h_l is smaller than f_s so this reduces to $O(h_l f_s^2)$: the matrix multiplies involving the $[f_s \times f_s]$ nullspace projector dominate. The number of iterations of the priority loop is at most six, so these expressions are also the extrinsic time complexity for the loop itself, with h_l replaced by the maximum level height:

$$O(h_{max} f_s^2) \tag{4.53}$$
$$\text{with } h_{max} = \max_l h_l.$$

Turning to the limiting loop, in the worst case every DoF could be pinned one at a time. But again this is unusual in practice, not only does it mean that the optimal pose for every tree DoF is at a limit, but also that SOLVE discovers this one joint at a time. Thus, at worst, the combined limiting and priority loops are $O(h_{max} f_s^3 + h_{max}^2 f_s^2)$, but the common case is that the number of limiting loop iterations is a constant (and $h_{max} < f_s$), reducing the typical extrinsic limiting loop run time to $O(h_{max} f_s^2)$, the same as the priority loop.

The number of iterations of the outermost convergence loop, and the number of calls to SOLVE itself, recalling that it will be called periodically as long as the system remains un-converged, is a question of convergence. This will be covered below.

Though matrix multiplication typically dominates the asymptotic time complexity of the inner loop, in practical use with linkages up to 100s of joints the higher constant factors associated with the SVD lead it to dominate in actual computation time. Some example timing breakdowns are given for a large simulated linkage in Section 5.4. There are alternatives to the SVD, some of which are faster, but the per-iteration speed gains may come at the sacrifice of numerical stability, accuracy, and convergence rate [20].

Both the (common-case) quadratic asymptotic time complexity of Solve and the practical runtime of the SVD are important factors limiting the scalability of the system to large models, and are major drivers for coupling and hierarchical decomposition, which can reduce the size of the part of the linkage presented to Solve.

Convergence and Local Minima

The core iteration in Solve is essentially a generalization of Newton-Rhapson iteration, and its convergence properties are considered to be similar [9]. In well-conditioned cases, the magnitude of the residual can be halved at each iteration. Unfortunately, there are also cases where a poor choice for the initial system state will result in an iteration that fails to converge, though issues of failed (or slow) convergence are usually avoidable in practical applications. This is supported both by my experience and by others who use the method or its variants for articulated system simulation [9, 20].

As with any local optimization approach, local minima must be avoided by higher-level means. For example, in interactive operations contexts, the operator can set the global shape of the commanded trajectory to avoid local minima. The situation can be more challenging for applications in compliant motion, especially when quasistatic modeling is used to represent link mass and joint stiffness. This can easily create a complex potential energy landscape. One saving observation is that finding a local minimum is not necessarily "wrong" in this context, since the passive physical system would itself behave in the same way. In the implemented system I have neither encountered any insurmountable local minimum issue.

4.9 Residuals and Jacobians

The specifications of the system model addressed by Solve are contained in the per-level residual vectors $\boldsymbol{e}_l$ and Jacobians J_l. These, in turn, are functions of the tree state $\boldsymbol{x}_s$, and are computed essentially according to the formulas referenced in table 4.1. A few aspects of these computations bear further discussion:

- the restricted forward kinematic mapping $\boldsymbol{y}_s = \text{FK}_s(\boldsymbol{x}_s)$ needs to be computed for the part of the linkage under consideration by SOLVE
- in some cases the nearest-alias difference $\overset{s}{-}$ needs to be found
- the kinematic Jacobian J_y must be computed.

Appendix G gives the details for computing the forward kinematic mapping using the $(\boldsymbol{t}, \boldsymbol{\theta})$ transform representation. The remaining items, $\overset{s}{-}$ and J_y, are covered in the next two subsections.

4.9.1 Computing Nearest Alias Differences

Some of the residuals require finding the *nearest alias* difference $\overset{s}{-}$ for a vector of joint DoF. The three residuals in which this operation is used—target (Eq. 4.18), posture (Eq. 4.27), and lock (Eq. 4.36)—each set goals on subsets of joint DoF. For each level, these DoF goals can be visualized as a single point in $\mathbb{R}^{6n}$, where n is the number of joints involved, formed by copying the current mobility state of each joint and then overwriting each DoF that has a set goal. There is a complication with this picture, though: the successive aliasing of π-wide shells of the rotation vector space for each joint (cf. Appendix B) implies that there is a corresponding infinite family of aliases for the current state of each joint, and hence also for the goal point.

The result of subtracting two DoF vectors will depend on the particular aliases chosen. My approach to this is a modified subtraction $\overset{s}{-}$ which uses the alias giving the smallest (in an L_2 norm) 6-dimensional difference vector for each individual joint considered separately. Fortunately, the dynamic reparametrization scheme means we only need to check two aliases for each joint: the *canonical* alias with rotation vector in $B^3(\pi)$, and the non-canonical alias in the next π-wide annular shell $B^3(2\pi) \setminus B^3(\pi)$. Non-canonical aliases must be checked because the nearest alias to a canonical rotation could be either canonical or non-canonical (the algorithm CLAMPX on page 233 uses the same logic to find an in-limits alias).

Because $\overset{s}{-}$ only checks two aliases for each joint, and because the shortest difference is found on a joint-by-joint basis, the computation of $\overset{s}{-}$ adds only a constant factor above a standard vector difference.

4.9.2 Computing Jacobians

The final element necessary for SOLVE is the kinematic Jacobian J_y, the matrix of partial derivatives of FK. Computing this type of Jacobian is a common task in articulated robotics [20, 55, 116]. However most algorithms given in the literature

- often handle only prismatic and revolute joints
- are not written in terms of the 6-dimensional $(\boldsymbol{t}, \boldsymbol{\theta})$ parametrization and the corresponding exponential and logarithmic maps for $\boldsymbol{\theta}$ (Appendix B)
- do not give details necessary for handling joints with inverted mobility spaces
- do not explicitly leverage the composite model transforms that are available (e.g.) as a by-product of the above residual computations.

Given a procedure to compute FK (Appendix G), numeric estimation of J_y is always possible: compute FK $f + 1$ times, once at $\boldsymbol{x}$ itself and then adding a small delta to each component of $\boldsymbol{x}$ in turn, and finally divide the measured change in each output component by corresponding the change in input. But such numeric computation can be relatively expensive, and may not be as accurate as a direct computation of the analytic derivatives of FK. This reduced accuracy could affect the convergence rate of SOLVE.

In this section section an algorithm is presented for analytically computing the Jacobian which addresses all of the above issues. What is really needed in SOLVE is the *restricted* kinematic Jacobian

$$J_s = \frac{\partial}{\partial \boldsymbol{x}_s} \mathrm{FK}_s(\boldsymbol{x}_s), \tag{4.54}$$

so the focus is on that. The extension to computing the full J_y, should it be needed, is direct.

Each entry in J_s is the partial derivative of one component of the restricted closure state $\boldsymbol{y}_s$ with respect to one component of the restricted tree state $\boldsymbol{x}_s$, evaluated at the current value of $\boldsymbol{x}_s$. If $c_s = |C_s|$ then the lengths of $\boldsymbol{y}_s$ and $\boldsymbol{x}_s$ are $6c_s$ and f_s, respectively, and J_s is a $[6c_s \times f_s]$ matrix. J_s is the local linearization of FK_s:

$$\Delta \boldsymbol{y}_s = \text{FK}_s(\boldsymbol{x}_s + \Delta \boldsymbol{x}_s) - \text{FK}_s(\boldsymbol{x}_s) \approx J_s \Delta \boldsymbol{x}_s. \tag{4.55}$$

J_s is used in the computation of each of the priority level Jacobians J_l according to the equations referenced from table 4.1 (these equations are written in terms of J_y, but J_s can be substituted when the corresponding restrictions are also applied to the rest of the equation). In most cases, J_s is immediately left-multiplied by one or more binary projection matrices, which effectively select only certain rows. Furthermore, the columns of J_s, which correspond to the concatenated set of the f_s DoF of the $t_s = |T_s|$ tree joints, can be considered a subset of a the columns of a larger $[6c_s \times 6t_s]$ matrix J selected by a new $[f_s \times 6t_s]$ binary projection matrix Π_{fs}:

$$\underset{[6c_s \times f_s]}{J_s} = \underset{[6c_s \times 6t_s]}{J} \; \underset{[6t_s \times f_s]}{\Pi_{fs}^T}. \tag{4.56}$$

UPDATEJACOBIAN, Algorithm 4.2, computes this full J matrix; both for clarity and for completeness. In the implementation, it is not actually necessary to do this full computation, nor to explicitly represent or multiply any of the projection matrices—they are but a means of mathematically presenting the computations—but only to do some bookkeeping to compute just the necessary elements of J.

UPDATEJACOBIAN fills in J in a row-major order. Each span of 6 rows corresponds to a unique closure joint $c \in C_s$, and similarly each span of 6 columns corresponds to a unique tree joint $t \in T_s$. By iterating c over the set of closure joints, the outermost loop in UPDATEJACOBIAN scans down 6 rows at a time. For each c the inner loop iterates t over the supporting tree joints, scanning across the columns in blocks of 6 and skipping blocks corresponding to non-supporting joints. If t is not in the support of c then the corresponding $[6 \times 6]$ sub-matrix is $\mathbf{0}$. Otherwise, three parameters Q,

Algorithm 4.2: UPDATEJACOBIAN(L, C_s, T_s)

```
Input: flattened linkage $L$, closure joints $C_s$, supporting unlocked tree joints $T_s$
Output: $[6|C_s| \times 6|T_s|]$ Jacobian $J$ as in Eq. 4.56
UPDATERESTRICTEDCMTS($L, C_s$) ▷ if not already updated
let $J \leftarrow \mathbf{0}$
foreach closure joint $c \in C_s$ do
    foreach unlocked supporting tree joint $t$ in $\left(\hat{S}\downarrow_c, \hat{S}\uparrow_c\right)$ in order do
        if $\phi_c < 0$ then $Q \leftarrow X_{g_0 \leftarrow pm_c}$ else $Q \leftarrow X_{g_0 \leftarrow cm_c}$
        if $t \in \hat{S}\downarrow_c$ then
            if $\phi_c < 0$ then $Q \leftarrow (X_{g_0 \leftarrow pm_t})^{-1}Q$, $R \leftarrow X_{g_0 \leftarrow cm_t}$
            else $Q \leftarrow (X_{g_0 \leftarrow cm_t})^{-1}Q$, $R \leftarrow X_{g_0 \leftarrow pm_t}$
            parallel $\leftarrow (\phi_c = \phi_t)$
        else ▷ passed LCA
            if $\phi_c < 0$ then $Q \leftarrow (X_{g_0 \leftarrow cm_t})^{-1}Q$, $R \leftarrow X_{g_0 \leftarrow pm_t}$
            else $Q \leftarrow (X_{g_0 \leftarrow pm_t})^{-1}Q$, $R \leftarrow X_{g_0 \leftarrow cm_t}$
            parallel $\leftarrow (\phi_c \neq \phi_t)$
        if $\phi_c < 0$ then $R \leftarrow (X_{g_0 \leftarrow cm_c})^{-1}R$ else $R \leftarrow (X_{g_0 \leftarrow pm_c})^{-1}R$
        compute $J_{c \leftarrow t}$ from $Q$, $R$, and parallel (Appendix H)
        fill in $[6 \times 6]$ block corresponding to $(c, t)$ in $J$ from $J_{c \leftarrow t}$
    end
end
return $J$
```

R, and *parallel* are derived from the topological relationship of the mobility actions of c and t, and the CMTs of their mobility frames. Details of these parameters and their use to compute the $[6 \times 6]$ *sub-Jacobian* $J_{c \leftarrow t}$ are given in Appendix H.

For completeness, UPDATEJACOBIAN starts with a call to UPDATERESTRICTED-CMTS. However when called from SOLVE, it can be arranged to always run directly after residual computation, so the necessary CMTs will have already been updated. Omitting the call to UPDATERESTRICTEDCMTS, the asymptotic running time of UPDATEJACOBIAN is dominated by the zeroing of J in $O(c_s t_s)$.

The inner loop in UPDATEJACOBIAN needs to traverse only the *unlocked* tree joints in the support chains for each closure joint. I use the notation

$$\hat{S}\downarrow_c = S\downarrow_c \setminus \Gamma_t, \ \hat{S}\uparrow_c = S\uparrow_c \setminus \Gamma_t \tag{4.57}$$

to indicate the support chains for a closure joint c with locked tree joints removed. For now assume this incurs no additional computational cost; the appropriate joint sequences will be computed in ANALYZE.

4.10 The ANALYZE Algorithm

All of the computations necessary for SOLVE have now been presented. While SOLVE can be applied directly to a full non-hierarchical linkage, the quadratic time complexity of each priority iteration and the cost of the SVD make it important to try to minimize the size of the part of the linkage that is processed: SOLVE can potentially be invoked separately on different parts of the linkage with a lower total runtime than if it were applied in a single call to the whole linkage. Also, SOLVE itself does not enforce the relative motion constraints on driving and driven sub-linkages with respect to their enclosing linkage.

Sometimes, decouplings are inherent in the structure of the model. For example, a kinematic task[6] involving only the arms of a humanoid robot can be solved independent of a second task involving only the legs. The first version of ANALYZE, presented as Algorithm 4.5, computes this *coupling decomposition*. Section 4.10.1 extends this to also incorporate a *hierarchical decomposition* which partitions sub-linkages along driving/driven boundaries.

Decomposition methods have been previously studied in constraint solving [77, 67]; ANALYZE is related in particular to algorithms presented by Light and Gossard [95] and by Welman [163], which also essentially perform coupling decompositions. However, ANALYZE is novel both in (1) its explicit use of the spanning tree/closure joint linkage structure, and (2) its handling of locked joints which helps support hierarchical decomposition (described below).

ANALYZE partitions the set of constrained[7] closure joints into a set $\mathcal{P}$ of p disjoint

[6]I.e. a fully actuated (Def. 57) task where the robot pose is always determined by actuators, overriding any other physical forces including gravity.

[7]CONSTRAINED?, Algorithm 4.3 on page 133, gives a precise definition of what makes a joint constrained.

components,

$$\mathcal{P} = (C_0, \cdots, C_{p-1}) \tag{4.58}$$
$$\text{s.t. } C = (C_0 \cup \cdots \cup C_{p-1} \cup \{\text{all unconstrained closures}\}),$$

so that SOLVE can be invoked separately on each. A corresponding set $\mathcal{T}$ of sets of unlocked supporting tree joints T_i is implied; for each C_i

$$T_i = \bigcup_{c \in C_i} (\hat{S}{\downarrow}_c \cup \hat{S}{\uparrow}_c) \quad \text{(cf. Eq. 4.57).} \tag{4.59}$$

The challenge is to form the finest partition $\mathcal{P}$ while ensuring that there is no overlap between any two T_i:

$$\forall_{0 \le i,j < p}\ i \ne j \implies (T_i \cap T_j) = \emptyset. \tag{4.60}$$

If there were such an overlap, then it would be possible that the state of some tree joint is changed by distinct invocations of SOLVE, and these changes could possibly conflict.

By construction, there is exactly one closure joint c for each topological cycle in the kinematic graph for a linkage L, and the rest of the joints in each such cycle are the support chains $S{\downarrow}_c$ and $S{\uparrow}_c$. Ignoring the issue of locked joints for the moment, we are looking for the *biconnected components* of L, considered as an undirected graph[8]: two distinct closures are in the same BCC iff their support cycles overlap, as illustrated in Figure 4-5.

A classical result in graph theory is that BCCs can be found in $O(|J|)$ [154]; ANALYZE is asymptotically slower in the worst case (analysis below), but unlike the classical algorithm, it handles locked joints in a way that will support hierarchical decomposition. Because locked tree joints are considered constant in SOLVE, ANALYZE omits them from the support chains—recall that the unlocked supports $\hat{S}{\downarrow}$ and $\hat{S}{\uparrow}$ are exactly what SOLVE and UPDATEJACOBIAN need. ANALYZE will separate

[8]A biconnected component of an undirected graph is a maximal connected sub-graph which remains connected even after the removal of any edge.

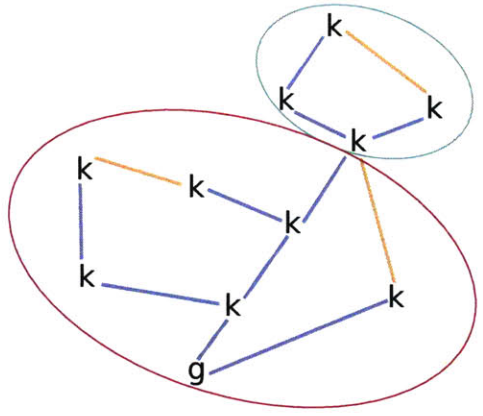


Figure 4-5: Coupling decomposition.
In this example, the overall linkage is separated into two biconnected components. Tree joints are drawn as dark lines; closure joints are light lines.

components which overlap only at locked joints.

ANALYZE operates in two phases. In the first phase, the recursive subroutine TRACESUPPORTS, Algorithm 4.6, performs a DFS of the linkage spanning tree. Each time a closure joint is encountered its supports are traced and stored in the $S\downarrow$, $S\uparrow$, $\hat{S}\downarrow$, and $\hat{S}\uparrow$ sequences. In addition, the set of closure joints supported by each tree joint t is saved in *supported-closures*$_t$, and the set of all constrained closures is collected in U. TRACESUPPORTS is asymptotically $O(|J| + |T||C|)$ (recall $C = J \setminus T$), due to (1) the DFS and (2) tracing the $O(|T|)$ support chain for each of the $O(|C|)$ constrained closures.

TRACESUPPORTS makes use of two helper functions, CONSTRAINED? and LOCKED?, Algorithms 4.3 and 4.4, to determine the status of each joint. With some simple bookkeeping, both of these functions can be implemented in $O(1)$.

```
**Algorithm 4.3**: CONSTRAINED?($j$)
  **Input**: joint $j$
  **Output**: **true** iff $j$ is constrained
  **return** $(Y_j \neq \mathbf{G})$ or $(I_j \neq \emptyset)$ or $j \in \Gamma$  $\triangleright$ *has DoI, limited DoF, or is locked*
         or (DoF of $j$) $\cap\, \boldsymbol{w}_t \neq \emptyset$  $\triangleright$ *in current target waypoint*
         or (DoF of $j$) $\cap\, \boldsymbol{w}_p \neq \emptyset$  $\triangleright$ *in current posture waypoint*
         or $\Upsilon_j \neq \emptyset$  $\triangleright$ *affects elastic or gravitational potential*
```

```
Algorithm 4.4: LOCKED?($j$)
  Input: joint $j$
  Output: true iff $j$ is locked
  return ($j \in \Gamma$)
```

The second phase of ANALYZE applies the recursive subroutine EXTRACTPART, Algorithm 4.7, repeatedly until all of the constrained chain closures have been assigned to one of the C_i in $\mathcal{P}$. The total time spent in EXTRACTPART is $O(|T||C|)$ because (1) the $O(|T|)$ support of each closure joint is traversed exactly once, and (2) the $O(|C|)$ set of *supported-closures* for each of the $O(|T|)$ supporting tree joints is also traversed once.

With these running times for each phase, the full ANALYZE algorithm is thus $O(|J| + |T||C|)$, or just $O(|J|^2)$ since $|T||C|$ is at worst proportional to $|J|^2$ (an example pathological case is given in Appendix G).

```
Algorithm 4.5: ANALYZE($L$)
  Input: flattened linkage $L$
  Output: partition $\mathcal{P}$ of constrained closure joints (Eq. 4.58)
          corresponding set $\mathcal{T}$ of sets of supporting unlocked tree joints
          complete support chains $S{\downarrow}_c, S{\uparrow}_c$ for each constrained closure
          unlocked support chains $\hat{S}{\downarrow}_c, \hat{S}{\uparrow}_c$ for each constrained closure
  let $T \subset J$ be the set of tree joints and $C = J \setminus T$ the set of closures in $L$
  foreach $j \in C$ do clear $S{\downarrow}_c$, $S{\uparrow}_c$, $\hat{S}{\downarrow}_c$, and $\hat{S}{\uparrow}_c$; mark $j$ unassigned
  foreach $j \in T$ do let supported-closures$_j \leftarrow \emptyset$; mark $j$ unassigned
  let $U \leftarrow \emptyset$  ▷the constrained closures will be collected here
  $U \leftarrow$ TRACESUPPORTS($g_0, U$)
  let $\mathcal{P} \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$
  while $U \neq \emptyset$ do
      let $c$ be the first element of $U$
      $(U, C_i, T_i) \leftarrow$ EXTRACTPART($c, U, \emptyset, \emptyset$)
      add $C_i$ to $\mathcal{P}$ and $T_i$ to $\mathcal{T}$
  end
  return ($\mathcal{P}, \mathcal{T}$), support chains as side-effect
```

```
Algorithm 4.6: TraceSupports($l, U$)
  Input: start link $l$, collected constrained closures $U$
  Output: $U$ updated with any newly found constrained closures
          support chains for the newly found constrained closures
  foreach $j$ s.t. $p_j = l$ do
    let $breadcrumb_l \leftarrow j$, $i \leftarrow c_j$
    if TREE?($j$) then $U \leftarrow$ TraceSupports($i, U$)
    else if CONSTRAINED?($j$) then
      add $j$ to $U$
      repeat ▷ trace down $S\downarrow_j$
        $p \leftarrow p_i$, append $p$ to $S\downarrow_j$, $i \leftarrow p_p$
        if $\neg$LOCKED?($p$) then append $p$ to $\hat{S}\downarrow_j$, add $j$ to $supported\text{-}closures_p$
      until $breadcrumb_i \neq \emptyset$
      ▷ $i$ is now the LCA of $c_j$ and $p_j$
      repeat ▷ trace up $S\uparrow_j$
        $c \leftarrow breadcrumb_i$, append $c$ to $S\uparrow_j$, $i \leftarrow c_c$
        if $\neg$LOCKED?($c$) then append $c$ to $\hat{S}\uparrow_j$, add $j$ to $supported\text{-}closures_c$
      until $c = j$
    end
  end
  $breadcrumb_l \leftarrow \emptyset$
  return $U$, support chains as side-effect
```

```
Algorithm 4.7: ExtractPart($c, U, C_i, T_i$)
  Input: initial closure $c$, remaining unexplored closures $U$
         set $C_i$ of constrained closures in component
         set $T_i$ of unlocked supporting tree joints supporting $C_i$
  Output: $U$ with $C_i$ removed, $C_i$, and $T_i$
  remove $c$ from $U$, add $c$ to $C_i$, mark $c$ assigned
  foreach unlocked supporting tree joint $t$ in $\left(\hat{S}\downarrow_c, \hat{S}\uparrow_c\right)$ do
    if $t$ unassigned then
      add $t$ to $T_i$, mark $t$ assigned
      foreach unassigned closure $u$ in $supported\text{-}closures_t$ do
        $(U, C_i, T_i) \leftarrow$ ExtractPart($u, U, C_i, T_i$)
    end
  end
  return $(U, C_i, T_i)$
```

4.10.1 Hierarchical Decomposition

With a few modifications, ANALYZE can be amended to also perform a hierarchical decomposition so that (1) a driving sub-linkage is solved before the enclosing sublinkage; (2) vice-versa for a driven sub-linkage; (3) a simultaneous sub-linkage is solved together with its parent. These constraints on the sequencing of SOLVE calls can be enforced by a new partial order on the $(\mathcal{P}, \mathcal{T})$ partition returned by ANALYZE.

Definition 62 Each *solve component* $(C_i, T_i) \in (\mathcal{P}, \mathcal{T})$ will be assigned a *solve round* in $\mathbb{Z}^{0+}$ corresponding to the partial order of SOLVE calls.

The ordering of calls to SOLVE within the same round is not constrained, but all calls for one round must be completed before advancing to the next round.

Consider a graph $\mathcal{L}$ with vertices corresponding to sub-linkages and edges to their parent-child relationships. Because all sub-linkages must be properly nested, one perspective is that $\mathcal{L}$ is a tree rooted at the top-level sub-linkage. However, it is also useful to consider the edges in $\mathcal{L}$ to be directed according to sub-linkage disposition. Specifically, let the edge connecting a driving sub-linkage to its parent be directed toward the parent, vice-versa for a driven sub-linkage, and let the edge connecting a simultaneous sub-linkage to its parent be bi-directed. $\mathcal{L}$ is thus a directed graph, and its *condensation*—the directed acyclic graph formed by collapsing each strongly connected component (SCC) into a single vertex—defines the desired partial order. Figure 4-6 illustrates these concepts.

The amendment of ANALYZE will produce a partition such that no solve component extends beyond a single SCC of $\mathcal{L}$. However, due to the coupling decomposition, there may still be more than one solve component per SCC, and it is also possible that a solve component extends beyond the boundaries of a simultaneous sub-linkage. Finally, all, but not necessarily only, the solve components within the same SCC of $\mathcal{L}$ are in the same solve round—there may be solve components in other SCCs also in that round.

ASSIGNROUNDS, Algorithm 4.8, marks each sub-linkage with a unique identifier of the SCC of $\mathcal{L}$ to which it belongs, and also labels it with its solve round. Each

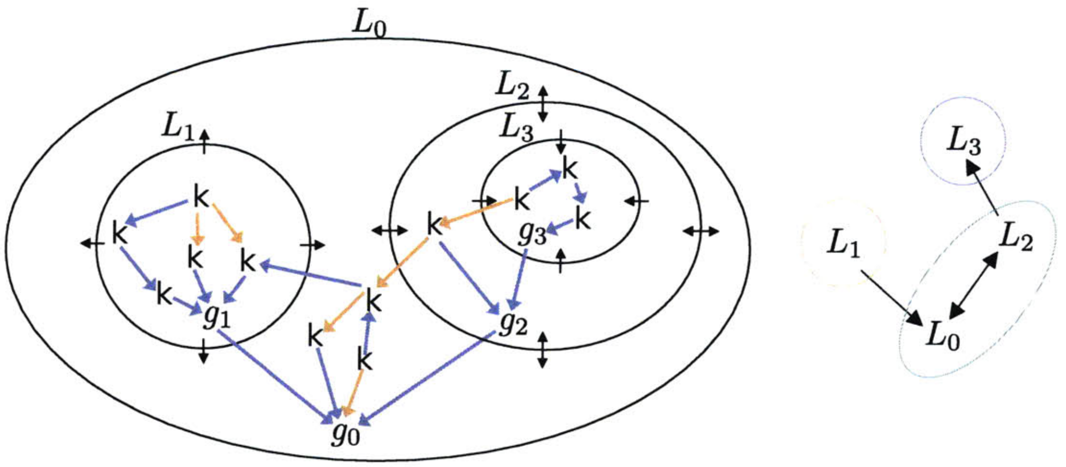


Figure 4-6: Hierarchical decomposition.
In this example, a hierarchical linkage (left) is decomposed according to the driving/driven relationships among its sub-linkages. The condensation graph of the resulting directed graph, right, defines a partial order for solve components.

solve component inherits the solve round of the sub-linkage(s) containing it.

Definition 63 A sub-linkage is a *source*, i.e. it is in solve round 0, iff (1) it's the top-level or any driving sub-linkage and (2) it has no driving descendant reachable only via a chain of simultaneous sub-linkages.

AssignRounds relies on two recursive sub-routines, FindSources and MarkSCCs (Algorithms 4.9 and 4.10). The top-level call to FindSources performs a DFS on $\mathcal{L}$ to find all source sub-linkages, and so is $O(|\mathcal{L}|)$, including all recursive calls. Each call to MarkSCCs marks the SCC identifier and solve round on all sub-linkages in an SCC. The total time spent across all calls to MarkSCCs is thus also $O(|\mathcal{L}|)$, making AssignRounds itself $O(|\mathcal{L}|)$.

```
Algorithm 4.8: AssignRounds($\mathcal{L}$)
  Input: sub-linkage graph $\mathcal{L}$ with top-level sub-linkage $L_0$
  Output: each sub-linkage is marked with its solve round
  foreach sub-linkage $L \in \mathcal{L}$ do $round_L \leftarrow -1$
  let set of source sub-linkages $U \leftarrow \emptyset$, number of SCCs $n \leftarrow 0$
  FindSources($L_0, U$)
  foreach $L \in U$ do $n \leftarrow n +$ MarkSCCs($L, 0, n$)
```

Algorithm 4.9: FINDSOURCES(L, U)

```
Input: sub-linkage $L$, collected source sub-linkages $U$
Output: true if driven by descendant
let $dbd \leftarrow$ false $\triangleright$ driven by descendant
foreach sub-linkage $M$ s.t. $p_M = L$ do
  if FINDSOURCES$(M, U)$ then $dbd \leftarrow$ true
if $\neg$ $dbd$ and $((L = L_0)$ or $(d_L =$ driving$))$ then add $L$ to $U$
return $(d_L =$ driving$)$ or $(dbd$ and $(d_L = \emptyset))$
```

Algorithm 4.10: MARKSCCS(L, r, i)

```
Input: sub-linkage $L$, solve round $r$, SCC identifier $i$
Output: the number of SCCs marked
$solve\text{-}round_L \leftarrow r$, $SCC_L \leftarrow i$
let $n \leftarrow 1$ $\triangleright$ number of marked SCCs
foreach sub-linkage $M$ s.t. $p_M = L$ do
    if $(round_M < 0)$ and $(d_M \neq$ driving$)$ then
        if $d_M = driven$ then $n \leftarrow n +$ MARKSCCS$(M, r+1, i+n)$
        else $n \leftarrow n +$ MARKSCCS$(M, r, i)$ $\triangleright M$ is simultaneous
    end
end
if $L \neq L_0$ then
    if $(round_{p_L} < 0)$ and $(d_L \neq$ driven$)$ then
        if $d_L = driving$ then $n \leftarrow n +$ MARKSCCS$(p_L, r+1, i+n)$
        else $n \leftarrow n +$ MARKSCCS$(p_l, r, i)$ $\triangleright L$ is simultaneous
    end
end
return $n$
```

Amending ANALYZE

Hierarchical decomposition is enabled by making three amendments to the previously defined ANALYZE algorithm:

- a call to ASSIGNROUNDS is inserted at the beginning of ANALYZE
- a sort of the $(C_i, T_i) \in (\mathcal{P}, \mathcal{T})$ in order of increasing solve round is inserted at the end of ANALYZE
- the calls LOCKED?(p) and LOCKED?(c) in TRACESUPPORTS are replaced with LOCKEDWRT?(p, j) and LOCKEDWRT?(c, j), respectively.

The first two amendments take care of setting up the solve rounds, and the last ensures that ANALYZE always breaks SOLVE calls at driving/driven sub-linkage boundaries. This is done by generalizing the notion of locking so that each tree joint can be considered locked with respect to some closures but not to others: besides checking if a tree joint t is explicitly locked, LOCKEDWRT?(t, c), Algorithm 4.11, also checks if t and c are not in the same SCC (normally, crossing joints are treated as members of the inner sub-linkage, but crossing joints on driving sub-linkages are an exception: they are treated as if in the super-linkage).

Algorithm 4.11: LOCKEDWRT?(t, c)

```
Input:  tree joint $t$ in sub-linkage $L_t$ (i.e. $t \in J_{L_t}$)
        closure joint $c$ in sub-linkage $L_c$ (i.e. $c \in J_{L_c}$)
Output: whether $t$ is to be considered locked with respect to $c$
if $t \in \Gamma_t$ then return true
let $i \leftarrow scc_{L_c}$
if CROSSING?$(c)$ and $(d_{L_c} =$ driving$)$ then $i \leftarrow scc_{p_{L_c}}$ ▷see text
return $(i \neq scc_{L_t})$
```

The running time of ANALYZE thus amended becomes $O(|J| + |T||C| + |\mathcal{L}| \log |\mathcal{L}|)$ due to the sorting of solve rounds (of which there are at most $|\mathcal{L}|$, the total number of sub-linkages).

4.11 Interacting with Model State

Whereas Section 3.6 and Appendix D presented a comprehensive set of operations for evolving the structure of a linkage model, corresponding operations to mutate state are relatively simple because the heavy lifting—propagating the effects of the mutations—is done by SOLVE.

There are just two essential categories of state change operations: changing DoF target or posture values, and changing the potential energy model for a joint. Both can be exposed to the user as simple discrete mutation operations, but for the first category it is also helpful to provide more intuitive means such as

- *click-and-drag* manipulation—the operator selects links or joints with the mouse, and drags them to effect motion in an intuitive way. Because the standard mouse input device only reports two axes of translational motion, special attention needs to be paid when the manipulated object has more than two DoF. For example, different mouse buttons and/or keyboard modifiers can be used to select the DoF to operate. Computationally, drag gestures are discretized and used to repeatedly set new target waypoints for the manipulated DoF.

- *special purpose input devices*—mouse-based manipulation is general-purpose and requires no special hardware, but can only control two axes of motion simultaneously. This limitation can be addressed, for example, by providing application-specific input devices with mobility that more closely matches that of the manipulated linkage model. One example of such a device is presented in Appendix I.

- *trajectory following*—here the operator essentially specifies a script for the motion of a set of DoF, for example, by entering a sequence of waypoints and the time intervals that should elapse from one to the next. The system then effects the motion by successively generating target waypoints along the specified trajectory.

All of these methods are available in the implementation, with examples given in

Chapter 5.

A few new structure-changing operations are also implied by the constructs introduced in this chapter. Adding a target or posture goal on a DoF which currently has none is handled as a structure mutation, as is the operation of removing such a goal. Joint locking and unlocking are also handled as structure mutations.

4.12 The Simulation Loop

The processing required to handle state and structure changes could be considered both in batch and in interactive contexts. The focus here—and in the software implementation developed with this thesis—is on the latter, though same general techniques should also be adaptable for non-interactive use.

At the top level, the interactive implementation is organized into a SIMULATION-LOOP, Algorithm 4.12. The main idea is to interleave the externally requested model updates with calls to SOLVE to automatically evolve the model state. ANALYZE is called after structure changes to (re-)partition the linkage, and SOLVE is then invoked in turn for each resulting part. The wall-clock time limit in SOLVE helps ensure that the simulation loop will not block excessively in any given iteration, so rendering can also be interleaved in the loop.

The simulation loop effectively serializes access to the linkage data structures, providing a baseline single-threaded synchronization model. Extensions should be possible to exploit parallelism on multiprocessor or multicore systems. For example, a rendering thread could potentially be overlapped with the simulation thread, and multiple SOLVE calls on independent parts of the linkage could be executed in parallel. Such ideas are future work.

4.13 Summary

While Chapter 3 developed the structure and topology of the linkage representation, this chapter has added the other half of the picture by showing how to compute

Algorithm 4.12: SimulationLoop(L)

```
Input: linkage $L$
repeat
    apply any pending structure and state changes to $L, \boldsymbol{x}, \Gamma, \boldsymbol{k}, \Upsilon, W_t,$ and $W_p$
    (including those from user input devices and any active trajectory drivers)
    if structure changed then $(\mathcal{P}, \mathcal{T}) \leftarrow$ Analyze($L$)
    if state changed then
        ▷ update independent tree joints, see page 119
        foreach tree joint $j \in J$ if $supported\text{-}closures_j = \emptyset$ do
            if $j \notin \Gamma_j$ then ▷ $j$ not locked
                if $\Upsilon_j \neq \emptyset$ then set $j$ to its potential energy minimum
                else if $(DoF\ of\ j) \cap \boldsymbol{w}_t \neq \emptyset$ then set indicated DoF to targets
            end
        end
        UpdateAllCMTs($L$)
    end
    if at least 1/desired FPS since last render then render $L$
    foreach $(C_i, T_i) \in (\mathcal{P}, \mathcal{T})$ in order do
        Solve($C_i, T_i, \boldsymbol{x}_i, \Gamma_c, \boldsymbol{k}, \Upsilon, W_t, W_p$)
        if $(C_i, T_i)$ last in round then UpdateAllCMTs($L$)
    end
until program exit
```

useful motions of a linkage. Together, these make a comprehensive framework for constructing, interacting with, and simulating models of articulated robots combined with virtual articulations.

In developing computations for linkage motions which solve the *local assembly* and *differential control* problems, this chapter has covered a lot of ground. These problems were first given formal definitions, and their solution via local linearization and *prioritized damped least squares* was developed. I have not simply instantiated this well-known technique, but specifically applied it to the particular challenges of mixed real/virtual systems. In particular, a suite of six priority levels were introduced that solve local assembly and differential control, with extended functionality for specifying default linkage pose, handling DoF limits and locked joints, and minimizing elastic and gravitational potential energy.

A Solve algorithm implementing the PDLS iteration of these priority levels was presented, along with supporting algorithms and formulas for computing Jacobians, residuals, and stepsize and damping factors. All necessary analytic details were presented based on the 6D $(\boldsymbol{t}, \boldsymbol{\theta})$ rigid transform parametrization.

Because each iteration of SOLVE is essentially quadratic in the number of joints, and to correctly sequence the motion of driving and driven sub-linkages, an ANALYZE algorithm was also presented to compute *coupling* and *hierarchical* decompositions, potentially breaking a large linkage into smaller independently solvable pieces.

All of the data structures and algorithms presented in this chapter (and in Chapter 3) have been implemented, and the resulting integrated software constitutes the testbed upon which the experiments in later chapters are based.

Chapter 5

Operating High-DoF Robots

This chapter presents a set of applications examples for the models and algorithms developed in Chapters 3 and 4, where virtual articulations and kinematic abstractions form the basis for a new kind of *operations interface* for specifying and constraining motion in high-DoF robots, i.e. robots with large numbers (10s to 100s) of joints. This is one of two application domains explored in this thesis; Chapter 6 applies virtual articulations and kinematic abstractions to substantially different problems in the area of modeling compliant locomotion.

Due to their application flexibility, high-DoF robots are increasingly common. This flexibility is especially attractive in space exploration, where the extreme costs of transportation from Earth are balanced by maximizing versatility, reusability, and redundancy in the delivered hardware. Such considerations have been a prime motivation for NASA/JPL's development of the 36-DoF All-Terrain Hex-Legged Extra-Terrestrial Explorer (ATHLETE) [167], with which astronauts will collaborate in our planned return to explore the Moon (Figure 5-2). A main feature of this chapter is the set of examples in Section 5.3, where virtual articulations in the mixed real/virtual interface are used to operate inspection and manipulation tasks on the ATHLETE hardware that would have been challenging in prior systems [159].

What makes High-DoF operations difficult? One issue is that there are often many ways the robot could move to achieve a task (i.e., there is often a high degree of redundancy), and some motions may be better than others due to secondary goals.

Sometimes a human operator can quickly visualize a desired motion, but in prior interfaces the expression of this motion to the operations system could be a tedious bottleneck. The approach described in this chapter allows the operator to graphically specify a broad class of motions (recall the generativity conferred by Kempe's theorem) using virtual articulations. For topologically large systems, structure abstraction (Section 3.5.3) can be applied to hierarchically subdivide the problem.

The core idea is to use virtual articulations to (1) constrain and (2) parametrize the intended motion (Figure 5-1, and recall Figures 1-3 and 1-1). Using the mixed real/virtual interface, the operator is presented with a graphical model of the robot and a catalog of available virtual joints (Figure 3-6). To constrain motion for a particular task, the operator instantiates joints from this catalog and interconnects them to the links of the actual robot—and/or to newly instantiated virtual links—thereby constructing arbitrary virtual extensions to the actual robot kinematics. Virtual joints can be erected to parametrize specific task DoF; for example the long prismatic virtual joint in Figure 5-6 parametrizes the length of a trenching motion. By closing kinematic chains, virtual articulations can also constrain whole-robot motion, thus narrowing the space of possible motions for a redundant task to those that satisfy the operator's intentions. The virtual Cartesian-3 joint in Figure 5-6 is an example of such a constraint. Virtual links can serve as interconnection points for more complex constructions—the chain of two prismatic and two revolute virtual joints in Figure 5-6 is interspersed with three virtual links—and can also model task-related coordinate frames or world objects, e.g. as in the inspection example in Figure 5-5.

The mixed real/virtual interface system computes local assembly motions as virtual articulations are constructed. From there, the operator can move any joint or link, and the system interactively solves the differential control problem in real-time to find a compatible motion for all joints which best satisfies all constraints. For example, in the trenching task, the operator can effectively command "trench from -0.9m to +0.4m" by operating the corresponding virtual prismatic joint, or they may simply drag the constrained end effector with the mouse. The motions can be validated in simulation and then executed on the hardware, as we have done for ATHLETE.

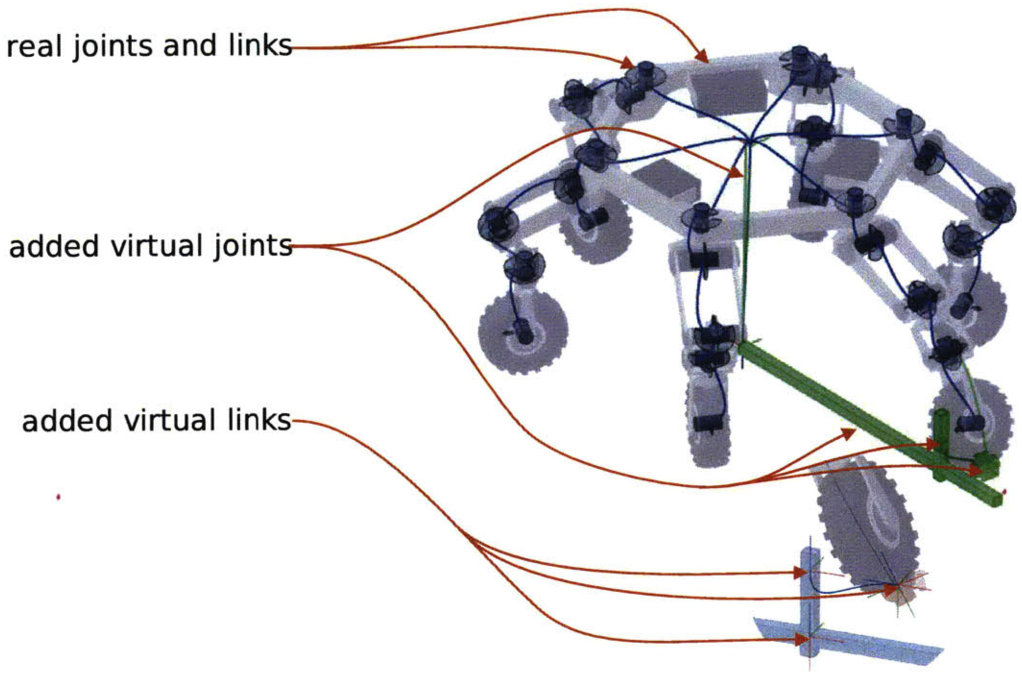


Figure 5-1: Parametrizing and constraining motion with virtual articulations. In this trenching task (also see Figure 5-6), a collection of virtual prismatic and revolute joints parametrize the main task motion, while a virtual Cartesian-3 joint constrains the deck to maintain its orientation.

Another key advance is that this approach to high-DoF operations is generally applicable to kinematic operations in articulated robots of any topology, allowing both open- and closed-chains as well as both over- and under-constraint. To demonstrate this topology-independence, in addition to ATHLETE, Section 5.4 gives a detailed example of operating a large-scale deformation motion in a self-reconfiguring modular robot. Two layers of structure abstraction are used in this case, which break up the motion specification problem for the operator, and also enable the system to decompose the full structure into smaller independently solvable parts (Section 4.10.1).

The next section discuss handling under- and over-constraint, two key challenges in high-DoF operations. The assumptions and limitations of the virtual articulation approach are then summarized in Section 5.2, with the ATHLETE and self-reconfiguring robot examples following in the subsequent two sections. Finally, Section 5.5 situates this new work in the context of other previously reported methods.

5.1 Handling Under- and Over-Constraint

Under-constrained systems are especially common in high-DoF robots, because often they are applied in tasks which involve only a few degrees of constraint, whereas there may be many more degrees of freedom available in (the active part of) the robot. Unlike some prior approaches, which are aimed towards fully constraining the motion [76, 78, 113], sometimes referred to as "redundancy resolution" in robotics (e.g. [19], Section 7.6), my perspective is that the operator should be free to add as much or as little constraint as desired.

At the lowest level, the least-squares nature of the PDLS solver (Chapter 4 and Appendix F) gives a basic ability to find reasonable motions in any under-constrained system: at each iteration a shortest step (by a Euclidean metric) is taken in joint space, resulting in incrementally minimal motion. Said another way, at a fine scale, the system will produce piecewise linear moves from one configuration to the next. A more roundabout trajectory might also be feasible, but would likely be surprising.

On top of this foundation, the operator may construct chain-closing virtual articulations to express specific motion constraints, and thus reduce redundancy as much as desired. Figure 1-3 gives a basic example, though this technique is pervasive in the whole approach and is also demonstrated in every example in this chapter.

Over-constraint, where the feasible configuration space is actually empty, is also a possibility. While at first this may seem a serious issue, in the presence of virtual articulations, it may be allowable—if necessary, virtual closure joints can be broken. Again, my perspective is to permit over-constraint, and to provide the operator with tools to handle it at two levels.

The lowest-level handling of over-constraint is again conferred by the least-squares nature of PDLS. If any individual priority level is over-constrained, then the least-squares solution will minimize the squared error across all constraints in the level. This is useful and can produce intuitive behavior from the operator's perspective. For example, when multiple closure joints need to be broken, the amount of breakage can be balanced across them all.

The PDLS priority levels (Section 4.7) provide a higher-level tool for handling over-constraint: satisfaction of constraints at a lower priority level will not compromise satisfaction at higher levels, even when the constraints conflict. Consider the spherical object inspection task in Figure 5-5. In this case we use a special-purpose hardware device called TRACK to pose the limb holding the inspection camera. But there is also a virtual spherical joint constraining the camera. TRACK has no haptic feedback, so while the operator will generally try to pose it near to a feasible configuration, invariably this will diverge from the strict spherical constraint surface, over-constraining the limb. The spherical joint constraint is modeled at the invariant level, and TRACK's pose is modeled at the target level, so the system will sacrifice the latter for the former. The overall effect is as if the virtual spherical joint was physically present and rigidly constraining the motion, and as if there were a breakable elastic connection between the pose of TRACK and the pose of the actual limb.

5.2 Assumptions and Limitations

While under- and over-constraint can be handled, this is not to say that the mixed real/virtual interface can work any miracles for physical systems that are truly under- or over-constrained. Essentially, to the extent that the under- or over-constraint can be restricted to just the virtual articulations, they can be controlled as desired.

Many robots are applied in practice in fully-actuated contexts (Def. 57) and are thus not physically under-constrained; the examples studied in this chapter all fall into this category. While such systems can still be over-constrained, if they can be assembled then at least the over-constraint is self-consistent.

Some under-actuated physical systems are statically determined in the sense that their state will correspond to a local minimum of gravitational and elastic potential energy. The quasi-static modeling capabilities of the mixed real/virtual interface can compute realistic motion for such systems, though I have yet to use these features in a high-DoF operations context (they are used in the compliant locomotion applications examples in Chapter 6).

Overall, some other limitations of using virtual articulations and structure abstraction to operate robots should be noted:

- the main thrust is to specify spatial motion trajectories; speed and force commanding, for example, are not directly addressed
- though Kempe's theorem ensures that a large class of trajectories will be constructable even given a limited catalog of joints, some will be out of reach. For example, a true helical joint cannot be synthesized from the joint catalog (Section 3.3.4) currently provided in the mixed real/virtual interface. Nonholonomic constraints, including rolling contact, are also not possible.
- when structure abstractions are applied, care must be taken either to ensure that they are proper (Def. 44), or that operating them outside the reachability of their implementation causes no unwanted effect.

With its capabilities and limitations thus outlined, the next sections move on to demonstrating the usefulness of the approach in two different robotic systems.

5.3 Experiments with ATHLETE

This section presents the first of two specific application studies, in this case using virtual articulations to operate whole-robot motions in ATHLETE, a new robot under development at NASA/JPL.

The All-Terrain Hex-Legged Extra-Terrestrial Explorer is under development for use in future Lunar missions, where it will potentially aid human explorers in various ways [165, 166, 167]. Figure 5-2 shows two instances of the current hardware.

ATHLETE weighs about 1000kg and has six identical limbs, each with six revolute joints and a terminal wheel (Figure 5-3 shows the joint axes). All joints have harmonic drivetrains and active-off brakes. The limbs attach to a hexagonal deck which is about 2.5m in (circumscribed) diameter and about 1.8m above the ground in nominal pose (foreground in the figure).

Various modes of locomotion are possible, including both walking (with the wheel brakes applied) and rolling. The limbs may also be used for manipulation and inspection tasks, which is the focus of the operations interface presented in this chapter.

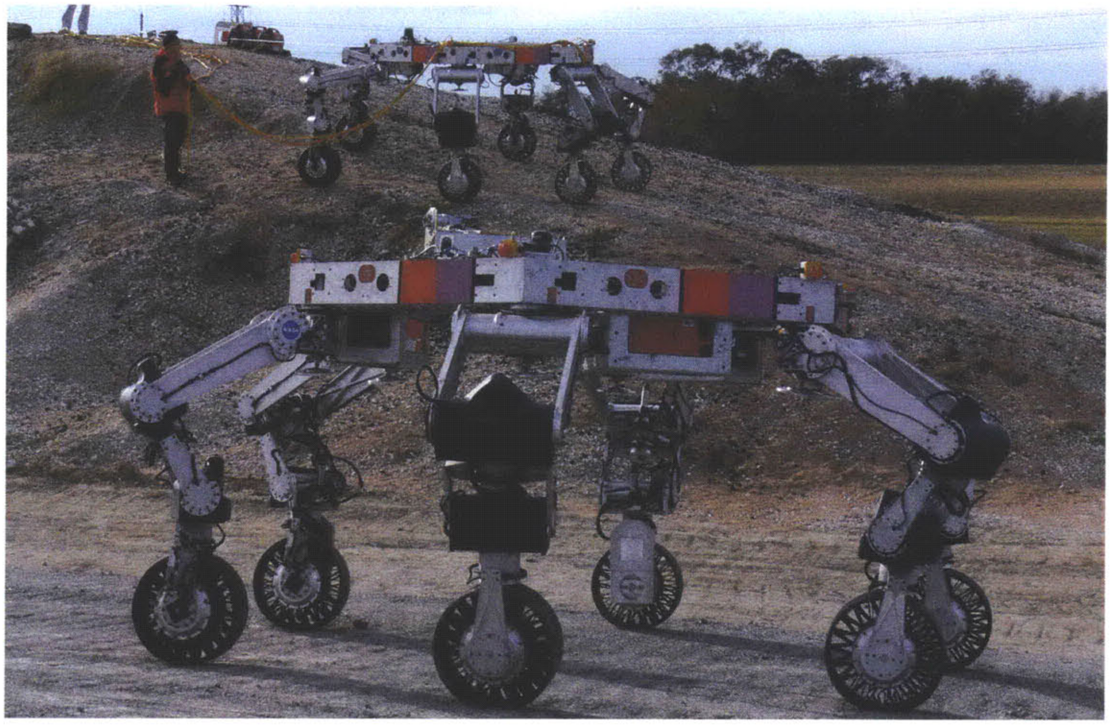


Figure 5-2: All-Terrain Hex-Legged Extra-Terrestrial Explorer (ATHLETE). The prototypes shown here are 1/2 scale with respect to the projected flight hardware. Image courtesy NASA/JPL/Caltech.

The six limbs of the robot are intended to be useful both in various kinds of locomotion (terminal wheels allow both rolling and walking), and also for manipulation and inspection tasks. Other researchers are considering the locomotion problems [62, 150, 144]; here, we focus on manipulation and inspection. The original operations interface for these kinds of motions, based on fairly standard approaches for prior systems both at JPL and more broadly in industrial robotics, consisted of three primitives:

- MOVE_JOINTS—simultaneous forward kinematic operation for all 36 kinematic DoF. Essentially equivalent to a board with 36 knobs, each commanding the position of one joint.

- MOVE_TOOL—inverse kinematics for one leg alone. A 6-DoF pose is specified for the end effector (EE), which in ATHLETE corresponds to the terminal link carrying the wheel. An analytic inverse kinematics algorithm computes a set of corresponding joint angles to pose the EE as requested with respect to the body (the hexagonal deck), which remains fixed in space.

- MOVE_BODY—inverse kinematics for the body alone. All wheels are locked in place, and a 6-DoF pose is specified for the hexagonal deck. These constraints reduce directly to inverse kinematics problems for the six individual limbs, which are moved accordingly to pose the deck.

While MOVE_TOOL is obviously useful, because it can only control one leg at a time, any task requiring multi-limb motion would need to be coordinated by some additional mechanism. Both MOVE_JOINTS and MOVE_BODY can move all the limbs together, but the former has obvious shortcomings (it simply puts the full burden of motion specification on the operator), and the latter is limited to controlling body posture alone.

Even in the case of single-limb motion, MOVE_TOOL is not always ideal. For example, some tasks do not constrain all 6 DoF of the EE pose; consider a drilling task—rotation of the EE about the drill axis may be unconstrained. And there are often multiple discrete solutions for fully constrained poses, i.e., ATHLETE's "elbows" can generally kink in different directions to reach the same EE pose. Thus, as an initial contribution, I designed and built a direct-manipulation input device that mimics one ATHLETE limb, called the Tele-Robotic ATHLETE Controller for Kinematics (TRACK) [103] (Figure I-1). Appendix I gives some details.

Though master-slave direct manipulation is not a new idea, Matt Heverly, chief mechanical engineer on the ATHLETE project and a frequent driver of the robot, said [64] "This is a fantastic new [device] that will allow us to command ATHLETE in a highly intuitive and efficient way." And, informally, another ATHLETE operator commented that TRACK "is like giving someone an ice scraper where before all they had were their fingernails." These comments highlight the tedium of using traditional operations methods (MOVE_JOINTS, MOVE_TOOL, MOVE_BODY) for a complex and highly capable high-DoF system like ATHLETE. But TRACK neither solves the whole problem—used alone, it still provides no mechanism for constraining motion or for coordinating motion of multiple limbs.

The remainder of this section shows five examples where virtual articulations aid the operation of specific whole-robot motions which are rapid for human operators to conceptualize but difficult to express in more traditional interfaces. Four of these were tested successfully on the hardware [159], one of which shows the combined use of both TRACK and the mixed real/virtual interface (Figure 5-5).

The overall procedure for the hardware experiments was to

1. load a model of ATHLETE in the mixed real/virtual interface (Figure 5-3)
2. design all virtual articulations and motions in simulation, using the system to compute local assembly and differential control motions
3. export the final robot motion as a joint space waypoint sequence (i.e. an automatically generated sequence of MOVE_JOINTS commands), generating a new waypoint whenever any joint moves at least 2°
4. check the sequence in a previously validated simulator (JPL's ATHSIM)
5. execute the sequence as a position-controlled trajectory on the hardware

Four different scenarios were tested on the hardware (each was repeated at least twice, though in this case the repeatability is actually a property of the hardware, not the interface system):

- Figures 5-4 and 5-5: a limb-mounted camera inspects a roughly spherical object while maintaining a constant distance
- Figures 5-1 and 5-6: a trench is inspected, with the support legs moving the deck to extend reachable trench length
- Figure 5-7: a rigidly mounted side-facing camera is made to pan and tilt with the motion both parametrized and constrained by virtual revolute joints

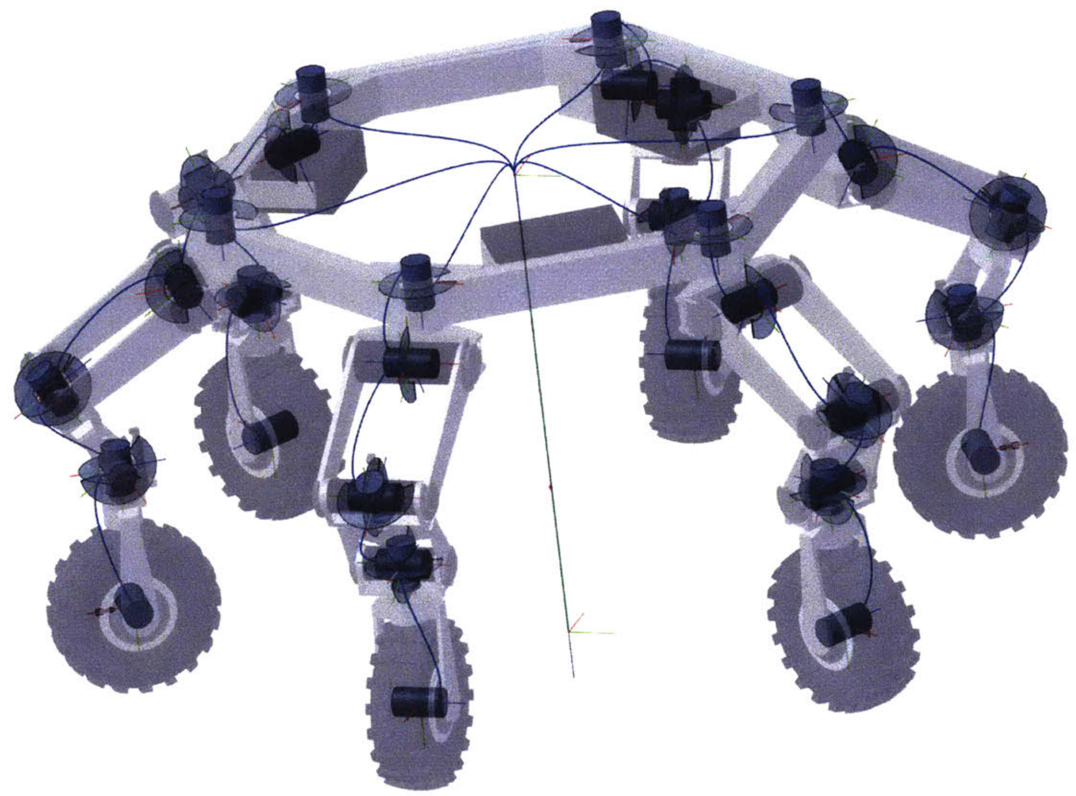


Figure 5-3: ATHLETE model in the mixed real/virtual interface.
A model of ATHLETE was provided by the RSVP team at NASA/JPL/Caltech in VRML format for use in the mixed real/virtual interface. The model is structured as a tree: the root is the ground frame (lower center) which connects to a link representing the ATHLETE hexagonal deck. From there, six branches of six revolute joints and six links model the limbs. (The wheel rotations are also modeled, but were locked in the experiments.)

- Figure 5-8: two limbs execute a pinching maneuver with the pinch distance and angles controlled by virtual prismatic and revolute joints

One additional experiment—inspecting a crew module, shown in Figure 5-9—was simulated, but we did not have the opportunity to implement it on the hardware.

For the object inspection task, the operator directly models the constant camera distance constraint using a virtual spherical joint connecting the object (itself represented as a virtual link) and the camera. The space of reachable viewpoints is extended (vs. moving only the camera limb alone) by using the five other limbs to lean the hexagonal deck, but because the deck often carries a payload, it should remain flat. This is expressed by a virtual Cartesian-3 joint connected between the deck and the world frame. After configuring these virtual articulations the operator can drag the camera, e.g. with the mouse, to scan the object.

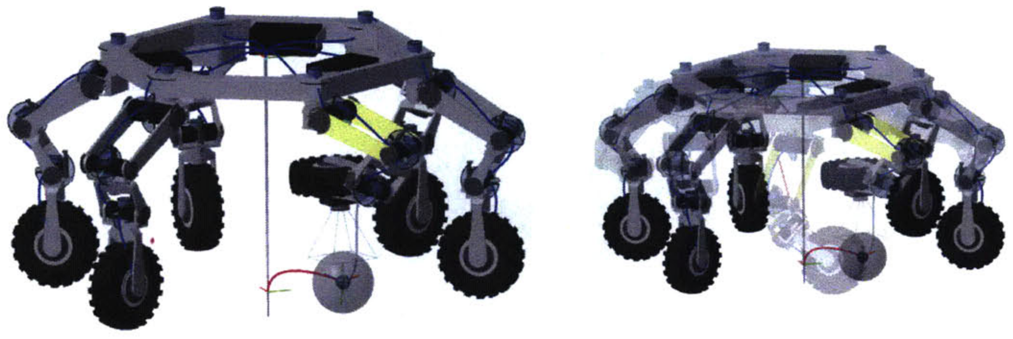


Figure 5-4: Inspecting an object using the mixed real/virtual interface.

Descriptions of the other experiments are similar, though for the object inspection experiment, one additional wrinkle was added: TRACK was integrated as a further aid to motion specification. One complication was that, since TRACK is strictly an open-loop device (it contains no haptic feedback), nothing prevents an operator from moving it in ways which violate the constraints (in this case, the virtual spherical surface). This example shows that the SOLVE priority levels (Section 4.7) can mitigate the issue: the spherical joint closure constraint is higher-priority than the EE target tracking command from TRACK. Thus, when the operator moves TRACK off the constraint surface, the system will simply move the camera to a nearest feasible pose, sacrificing fidelity of target tracking instead of breaking the joint.

For the bi-manual experiment the robot was partially supported by an overhead crane, as simultaneously raising two limbs is not supported on the current hardware. The crane served as a safety-backup in the other experiments.

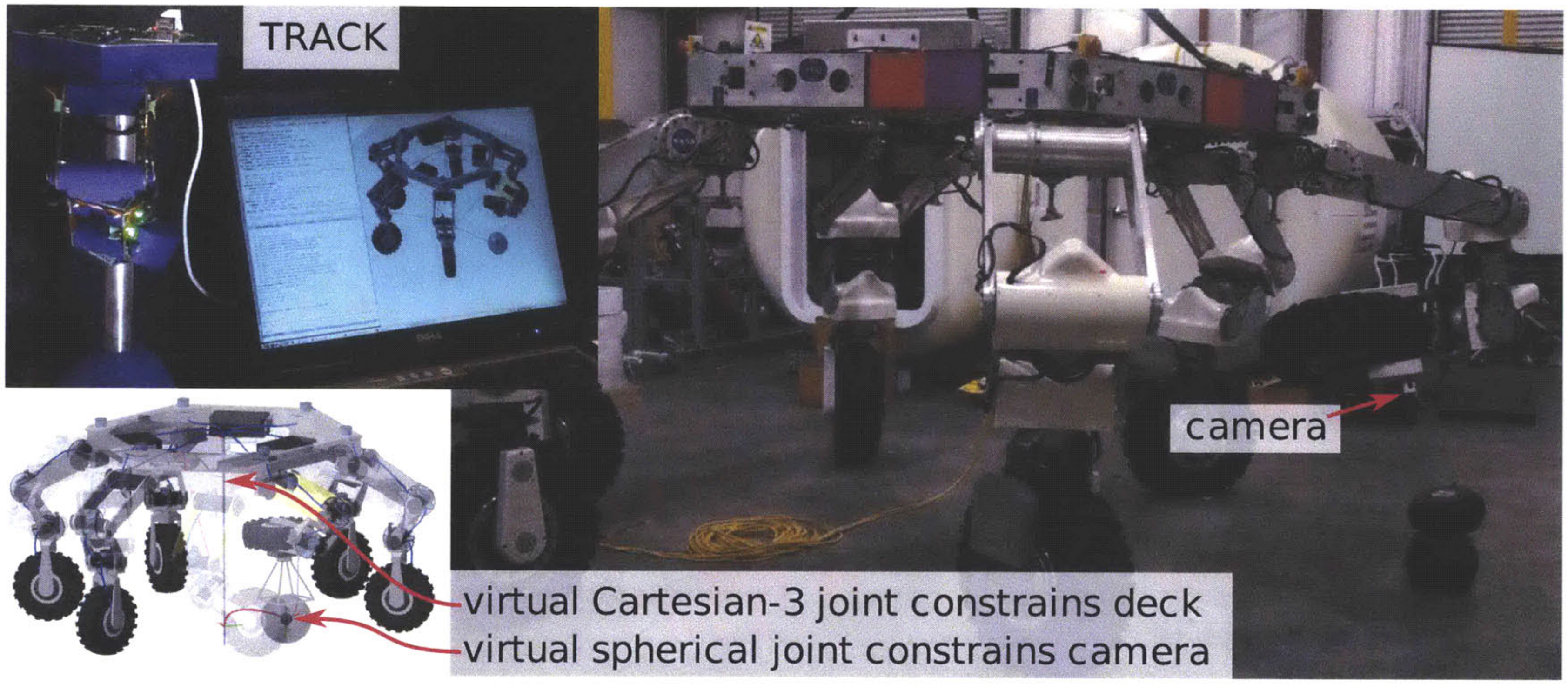

TRACK
camera
virtual Cartesian-3 joint constrains deck
virtual spherical joint constrains camera

Figure 5-5: ATHLETE inspecting an object with a leg-mounted camera.
In this experiment a leg-mounted camera inspects an object from a wide range of viewpoints, while a virtual spherical joint centered on the object maintains a fixed focal distance. The mixed real/virtual interface automatically computes postural motions that greatly extend the range of reachable viewpoints vs. actuating a single leg alone. Also, for this task the TRACK interface device (Appendix I) was combined with the virtual articulation system.

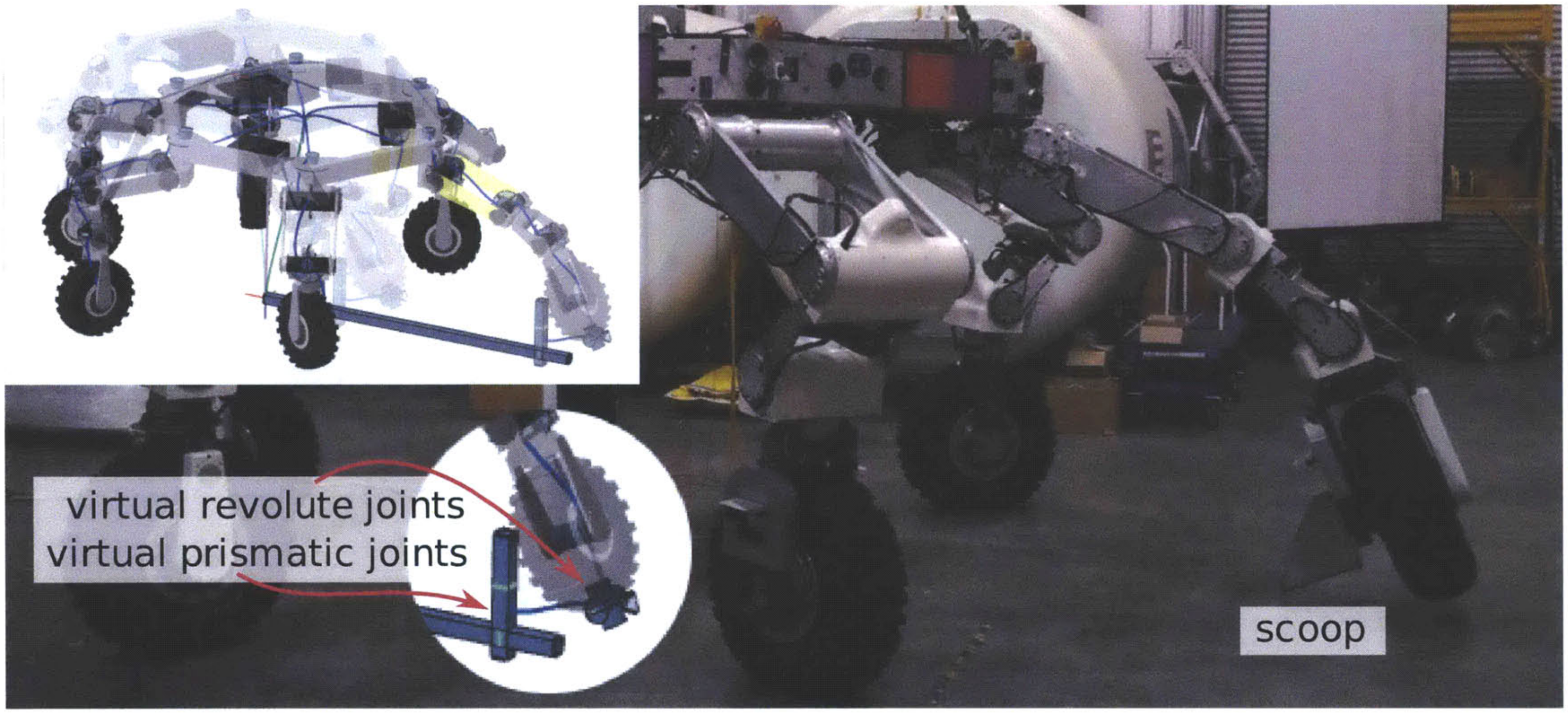


Figure 5-6: ATHLETE performing a trenching motion.
In this experiment a scoop is mounted to one limb and could be used to dig a trench, though actual digging would require some added force control. Virtual prismatic joints constrain and parametrize the length and depth of the trench, virtual revolute joints set the angle of the scoop, and a Cartesian-3 joint keeps the deck at a constant orientation with respect to the ground (e.g. in case that the deck is carrying an additional payload). Though a similar trenching motion could be operated using inverse kinematics for the active limb alone, the addition of constrained postural motion approximately doubles the total possible trench length.

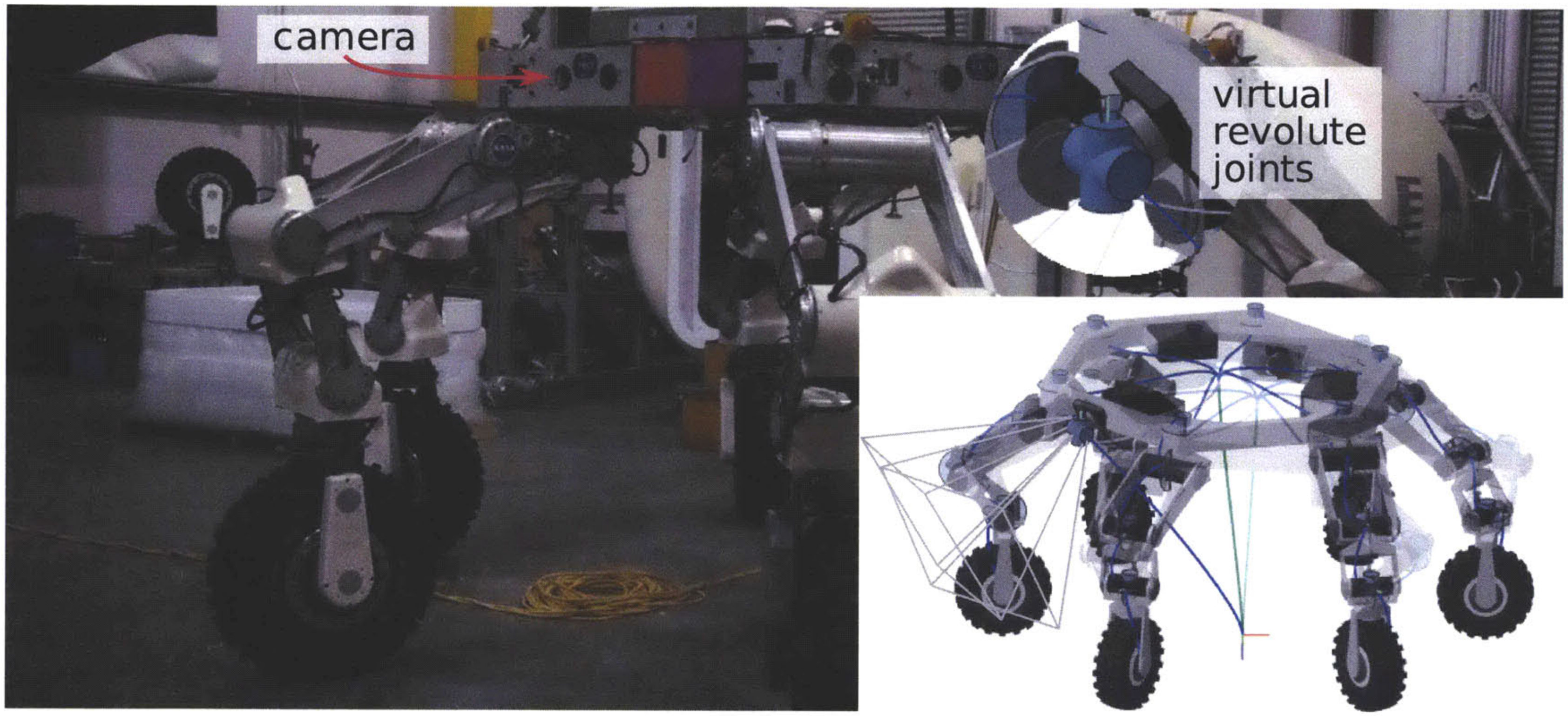


Figure 5-7: Pan/tilt of an ATHLETE camera via postural motions.
ATHLETE has a number of side-facing cameras rigidly mounted to its deck. For an extended field of view, pan and tilt can be emulated by postural motions where all legs cooperatively rotate the deck about the camera's center of focus. Such motion can be operated with two crossed-axes virtual revolute joints. With this construction in place, quantified operations commands such as "pan 30 degrees left, tilt 10 degrees down" have a specific meaning. The least-squares solver of the mixed real/virtual interface finds a sufficient whole-robot motion (standard driving pose, Figure 5-2, is set as the default posture).

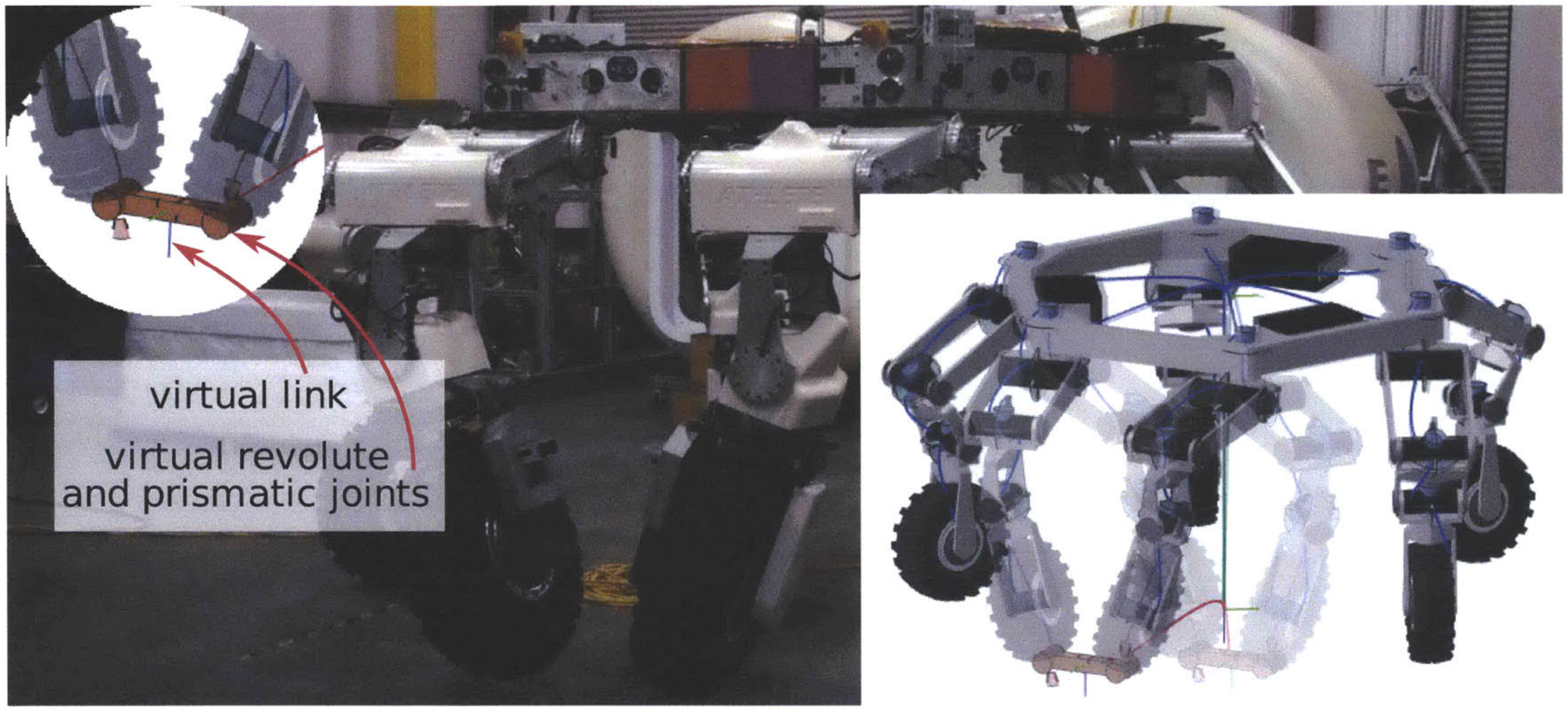


Figure 5-8: ATHLETE performing bi-manual manipulation motions.
In this bi-manual operations experiment, a virtual link models a movable object that could be grasped by pinching between two adjacent wheels (actually grasping an object would require the addition of force control, which is not yet implemented). The operator can simply drag this virtual link around in the mixed real/virtual interface; the system finds compatible kinematic motions for the legs. Motion is automatically limited to the reachable workspace by constraint prioritization, since the joint closure constraints are higher priority than the object pose target. Virtual prismatic and revolute joints additionally parametrize the grip.

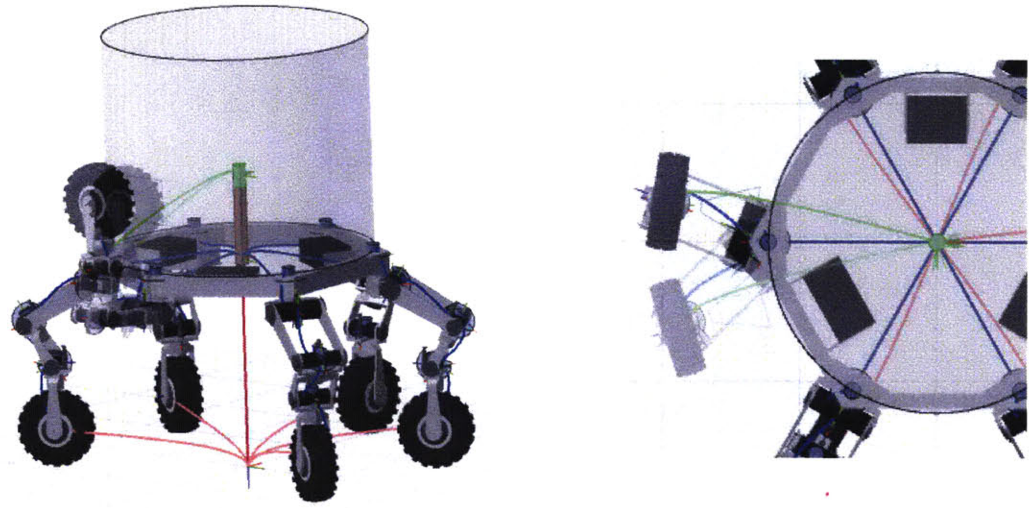


Figure 5-9: Simulation of ATHLETE inspecting a cylindrical crew module. In some scenarios, a cylindrical crew module may be affixed to the deck. A leg-mounted camera could be used to inspect its surface in a cylindrical scanning motion. Here, a virtual prismatic joint controls the elevation of the scan, and a virtual revolute joint controls the radial angle.

5.4 Scaling to 100s of Joints: Simulation of a Large Modular Robot

This section presents a second application in high-DoF operations, here for a simulated robot with 100s of joints, demonstrating the scalability of both the theory and the implementation of the mixed real/virtual interface. Two layers of structure abstraction are applied in this example, breaking up the operator's motion specification task and also enabling hierarchical decomposition for efficient motion computation.

The system in this section is a particular example of a *self-reconfiguring* modular robot. Modular SR robots, studied since at least the late 1980s [53, 105, 28], are systems which can globally form arbitrary shapes by re-arranging many smaller interconnected units [136, 135, 104]. This is another class of robot where large numbers of joints are possible—though hardware has so far been limited to a few 10s of modules [90, 16], work has been done in simulation [174] with larger numbers of up to 1 million [51].

The SR system concept introduced here is itself novel, and consists of active modules with the same kinematics as the *Shady* climbing and sun-shading robot presented later in this thesis (Section 6.6), plus passive modules in the form of simple bars [158, 42]. It is called *Multishady* because of its basis in the Shady kinematics. Appendix K gives additional details and research context.

Multishady is a true SR system, and is capable of arbitrary topological reconfiguration by attaching and detaching modules (Figure K-1). This kind of reconfiguration has been a main study in SR theory (e.g. [174, 137, 22, 5]), and is summarized by the *shape metamorphosis* problem [27]: given a start shape and a goal shape, find a (ideally short) sequence of module attach/move/detach operations which achieves the specified reconfiguration.

Chain-type modular robots [179], of which Multishady is an example, are also capable of *deformation* motions where the inter-module connectivity remains constant, and modules cooperatively use their internal kinematic DoF to effect a global shape-change. Operating this type of motion—i.e. providing an interface where an operator can conveniently specify general deformations in potentially large modular assemblies—has been under-explored; using virtual articulations to constrain and parametrize coordinated motion is a natural fit. By their nature, SR systems can assume arbitrary topologies, and one of the strong points of the virtual articulation approach is topology independence. Structure abstraction (Section 3.5.3) is also particularly useful, in part because repeated sub-structures are common in large SR constructions. (A related idea in topological reconfiguration is the concept of *meta-modules* [157, 112, 137, 119].)

The main result in this section is interactive operation of a tower involving 120 Multishady joints and over 150 additional virtual joints (Figure 5-11). The tower is built from a repeating block sub-structure. The operator first defines a particular kind of constrained motion at the block level, explained in the next section, and this induces the full-tower motions presented in the following section. Structure abstraction plays a key role, and enables a hierarchical decomposition which keeps an otherwise sluggish motion computation snappy (Figure 5-12).

5.4.1 Hierarchical Control for a Tower Block

In the 2D Multishady concept each active module contains two rotating grippers which can attach to passive bars (there is also a direct extension of the idea to 3D, see Appendix K). Though each module is small, a large structure, such as a tower, can potentially be constructed by assembling many modules. Doing this in reality would involve serious structural considerations—the total tower size would effectively be limited by the strength of the constituent modules. One way to mitigate this to some extent may be to build the tower using multiple chains of modules in parallel, i.e., building a thicker tower.

Here we consider the kinematic operation of a deformation in such a "thick" tower; for example, if a camera was affixed to the link at the top of the tower, then it could be moved about to inspect the supporting structure, as shown in Figure 5-11. Kinematic operation is not the only concern—mechanical stress and strain also matters—but it is still an interesting problem.

In this case two parallel chains run along the length of the tower, cross-braced together at intervals. The tower thus has a repeating block sub-structure, and the problem of operating it can be broken down along the same lines. Figure 5-10 zooms in on one such block, and shows the motion constraints the operator has designed, using two levels of kinematic abstraction. Extrinsically, the operator wants the tower block to act as if it had two overall DoF: left/right tilting, and up/down expansion and contraction. A highest-level abstraction is thus implied which replaces the entire block with a chain of one revolute and one prismatic joint, shown at left in the figure. But the actual Multishady modules making up the block—the implementation of this abstraction—have more complexity. They form two legs, which can be considered as four-bar linkages. Zooming in to a single leg, the desired extrinsic motion in this case is effectively like a piston: the end-to-end distance of the constituent chain of modules sets the length of the leg, and the endpoints can also pivot to allow the leg to rotate. This implies a second level of abstraction, shown in the middle of the figure, where each leg is virtually replaced by a piston-like assembly of virtual prismatic and revolute joints[1]. Finally, a Cartesian-2 constraint is added to keep the middle link of each leg parallel with the axis of the leg, shown at right in the figure. In this case, the abstractions can be kept proper (Def. 44) by sufficiently limiting the range of motion of the joints in the interface linkages.

It is important to note that these particular motion constraints, and this particular set of structure abstractions, are merely the designs of the operator in this instance. Other constraints and abstractions are possible: the idea is that the operator conceptualizes a desired motion, and then expresses it by designing virtual articulations and structure abstractions.

Williams and Mahew [168] considered a similar bending-tower scenario, and even structured the motion of their tower using a similar decomposition. Their study was significantly smaller in topological scale (30 DoF total); their implementation was apparently hard-coded for one particular structure; and their decomposition was manually applied, whereas my hierarchical decomposition algorithm automatically and assigns solvers to the decomposed sub-linkages.

5.4.2 Interactive Operation of a 15-Block Tower

Once the operator is satisfied with the operation of a single block, a number of them can be strung together to model towers of varying height. This final assembly is done at the highest level of abstraction, so that the top-level linkage is simply a linear chain of alternating revolute and prismatic joints along the "backbone" of the tower[2]. The operator can then interactively specify a motion, e.g. by click-and-drag interaction with the mouse, for any link or joint in the top-level linkage. The system will solve the differential control problem in and respond with a deformation as shown

[1]The top joint of each leg is actually modeled as a point-slider joint, with the translational axis perpendicular to the plane of the paper, because the mixed real/virtual interface does not currently have a true 2D modeling mode. Though a revolute joint (with rotation axis perpendicular to the page, as for the other revolute joints) would be technically correct here, it would also over-constrain the linkage when considered in 3D. Such over-constraint is permissible, but degrades performance.

[2]This particular arrangement is reminiscent of the backbone-curve method that has been proposed for controlling hyper-redundant robots [30], and also of the method presented in [168], but note that structure abstraction is a more general concept.

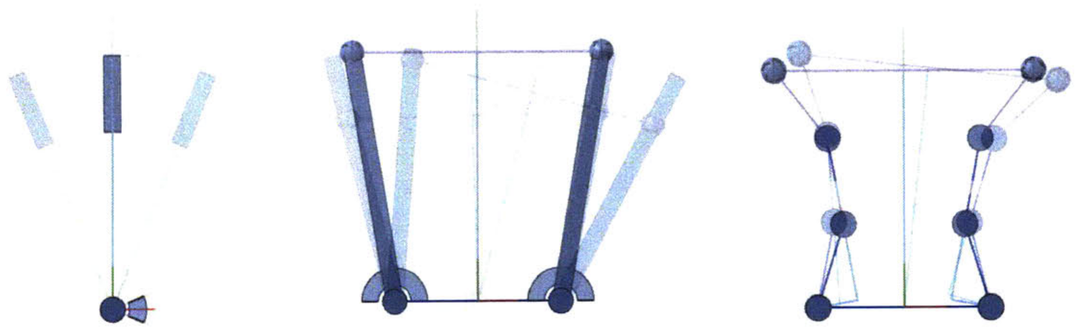


Figure 5-10: Operating a tower block with two levels of structure abstraction. The actual modules comprising a tower block (right figure, solid geometry) can move in a variety of ways. The operator intends only a subset of this motion. Extrinsically, the block should only have two DoF: it should be able to tilt left and right and to expand up and down (left figure). This forms the highest-level abstraction. Within this, a secondary constraint is that each 4-bar leg linkage should effectively act like a piston, with the middle link remaining parallel to the axis of the piston. Cartesian-2 joints, shown in the rightmost figure, enforce this constraint (also see Figure 1-3), and the leg assemblies are then each abstracted using revolute and prismatic assemblies as shown in the middle figure (also see Figure 3-9).

in Figure 5-11.

In this example, structure abstractions help the operator in the same way that traditional algorithmic abstraction aids the architect of a large software system: they break the problem up by enforcing high-level semantic invariants on sub-systems. The designer (in either situation) is thus not required to address the whole detailed problem at once.

These structure abstractions will also result in a hierarchical decomposition (Section 4.10.1) with three Solve rounds: First, the backbone will move as the operator has specified. Second, the top-level abstraction for each block will drive the mid-level abstractions for each leg. Finally, those abstractions will drive the motion of the modules that ultimately comprise the legs. The first round will consist of a single call to Solve, and the size of the system will scale linearly as the tower grows in height. The later rounds are different: there, growing the tower will result in a larger number of independent calls to Solve, but the system size in each call will remain constant.

What if structure abstraction had not been used? In fact, the same tower motion

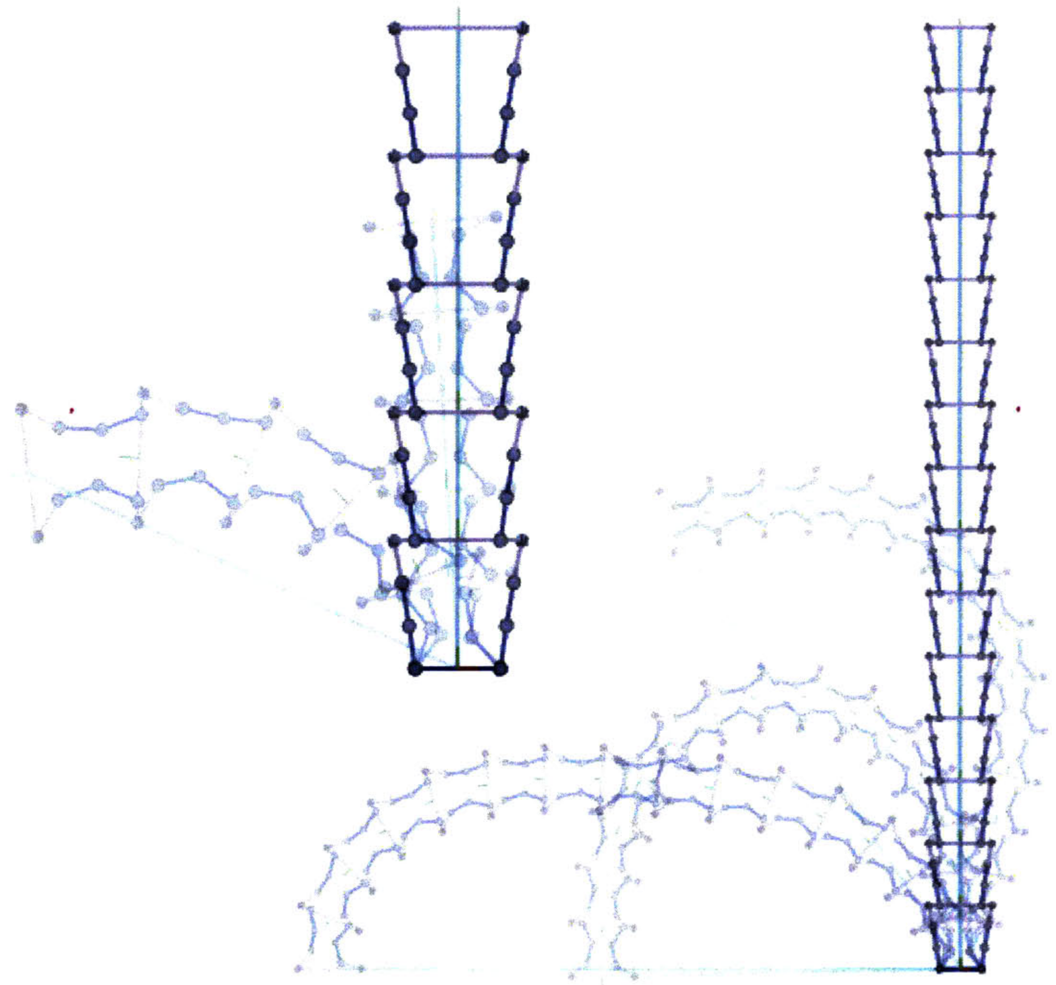


Figure 5-11: Operating deformations in 5- and 15-block towers.
This figure shows the interactive operation of a large-scale simulated tower formed by stacking block modules as shown in Figure 5-10. The large tower contains 15 blocks, with a total of 271 total joints (108 tree, 91 closure). 120 represent real joints in the robot modules, and the remainder are virtual joints involved in the block leg constraints and the structural abstractions. In these examples the operator is dragging the top of the tower with the mouse (both position and orientation can be controlled), and the mixed real/virtual interface solves the differential control problem in real time to deform the tower. The block-level structural abstractions are not only useful to the operator to sub-divide the motion specification problem; they also enable significant hierarchical decomposition, which keeps the per-iteration simulation computation time below 20ms even for the 15 block tower. A similar tower operated without the abstractions required over 100ms (Figure 5-12).

would theoretically result if the entire structure were constructed as a single flat (non-hierarchical) linkage. But in this case there would be only one SOLVE round. Like the first round in the hierarchical case, the size of the system in this giant call to SOLVE would scale linearly as the tower grows in height, but the constant factors would be greater, and it turns out that this makes a significant difference in computation time for even towers of modest height (~ 5 blocks).

To see this, I conducted an experiment where I built towers of varying heights, up to 15 blocks, in four different ways. The resulting measured[3] computation times are comparatively plotted in Figure 5-12. The "plain" tower used no hierarchy, and it also omitted the constraining Cartesian-2 joints. The "constrained" tower added the Cartesian-2 joints, and the "hierarchical" tower added the structure abstractions, but not the Cartesian-2 joints. Finally, the "constrained hierarchical" tower included both the abstractions and the Cartesian-2 joints, as originally presented.

In each case, adding the Cartesian-2 constraints increases the computation time, and adding the structure abstractions decreases it. This is as expected, considering the effects of each change on the size of the systems that are presented to SOLVE. The most interesting comparison is between the "constrained" tower, the worst case, and the "constrained hierarchical": the latter remains roughly on-par with the unconstrained versions, at about 20ms per simulation iteration, which is acceptable for interactive response. But all the additional Cartesian-2 closure joints severely impair the performance of the system for the "constrained" tower by forcing SOLVE to handle large systems; performance degrades to over 100ms per iteration for a 15 block tower, which results in very sluggish response as the operator drags the tower.

These timings are of course relative to the speed of the workstation on which they were run. Furthermore, the hierarchical decomposition does not change the asymptotic cost of SOLVE, which is still (typically) quadratic in the number of joints solved in any single system. But the lower constant factors for the hierarchical case mean that in practice, larger systems can be handled before reaching the limits of

[3]Each test was run interactively, the only mode currently implemented in the mixed real/virtual interface, on a lightly loaded workstation.

interactivity.

As a final set of data points, Figures 5-13 and 5-14 show the breakdown in computation time for the various parts of the SOLVE algorithm for each of the four versions of the tower. All but the unconstrained hierarchical version show two interesting features. First, there is a pronounced cross-over point in each of these three graphs, where the pseudoinverse, nullspace projector, and restricted Jacobian computations all become relatively more expensive than most of the others. This is simply explained—these computations all have quadratic dependence on the system size, while the others are linear, but may have higher overhead. The second interesting feature in these cases is that the SVD computation nearly always takes the largest fraction of time by a significant margin. This is neither surprising—it too is quadratic in the system size, and is known to have relatively high constant factors (Appendix E). The unconstrained hierarchical tower does not show these features, or at least they are not nearly as pronounced, because the absence of the added constraints keeps the worst-case overall system size significantly smaller.

5.5 Context in High-DoF Operations

Though it does not appear that any prior authors have used virtual articulations to build a general-purpose operations interface, there are some related systems. For example, Flückiger [52] describes a virtual reality operator interface, where a model of the robot and its surroundings is provided to the operator for pose manipulation. The mixed real/virtual interface goes beyond this by allowing the operator to virtually change and augment the kinematic structure itself.

Sections 2.5.1 and 2.5.2 summarize the possibilities to use existing physics simulation frameworks and interactive geometric constraint solvers, respectively, as substitutes for the mixed real/virtual interface. Both have significant drawbacks.

With respect to prior work in the specific field of operations methods, the new ideas demonstrated in this chapter can be considered to fill a gap between existing low-level methods, including forward and inverse kinematic control, and existing high-

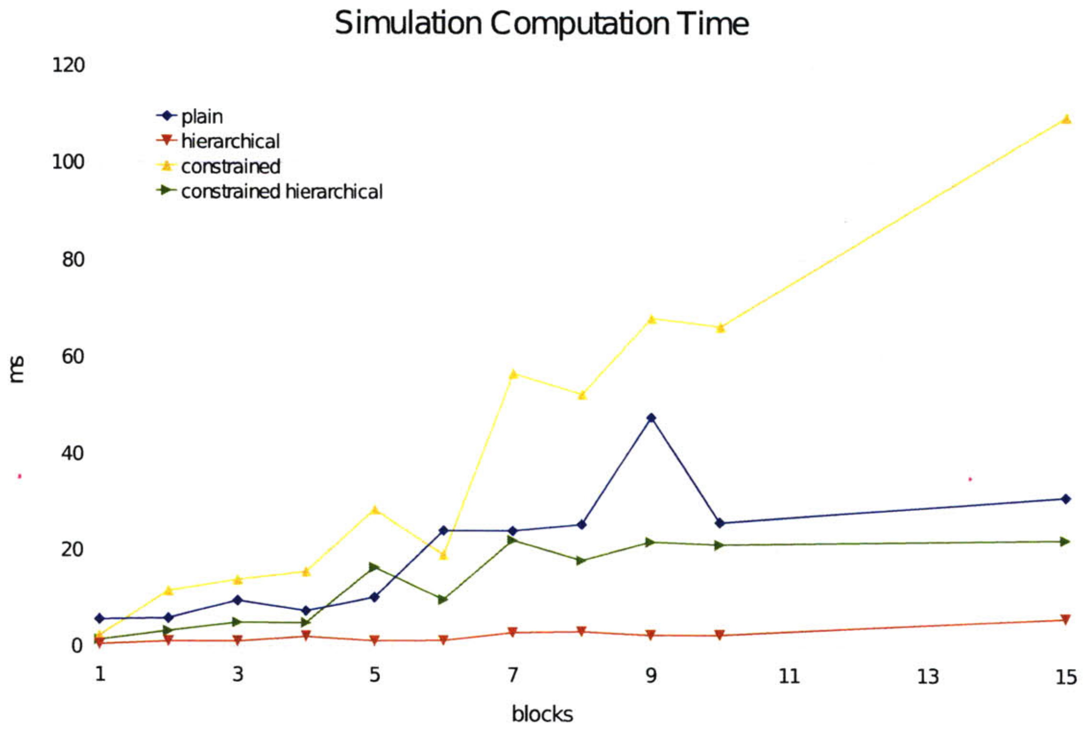


Figure 5-12: Simulation computation time for the modular tower.
This figure compares the average measured per-iteration simulation computation time (in milliseconds) for four different versions of the modular tower, all as a function of the tower height in blocks. To maintain interactive response to the operator, times in the low 10s of ms are preferable. The different tower versions are explained in the text; a main observation is that adding the block leg constraints causes the computation time to rise dramatically unless structure abstractions are also added.

level methods such as goal-based motion planning and programing-by-demonstration.

5.5.1 Previous Low-Level Methods

Bare kinematic control without higher-level goals or constraints is potentially tedious in the high-DoF case given the high dimension of the joint space (for ATHLETE, this could mean manually turning all 36 MOVE_JOINTS knobs simultaneously). Task priority and task space augmentation approaches [26] extend this to support high-DoF motion, including specified constraints, but do not themselves offer any particular way to *design* those constraints. The virtual articulation approach provides a concrete

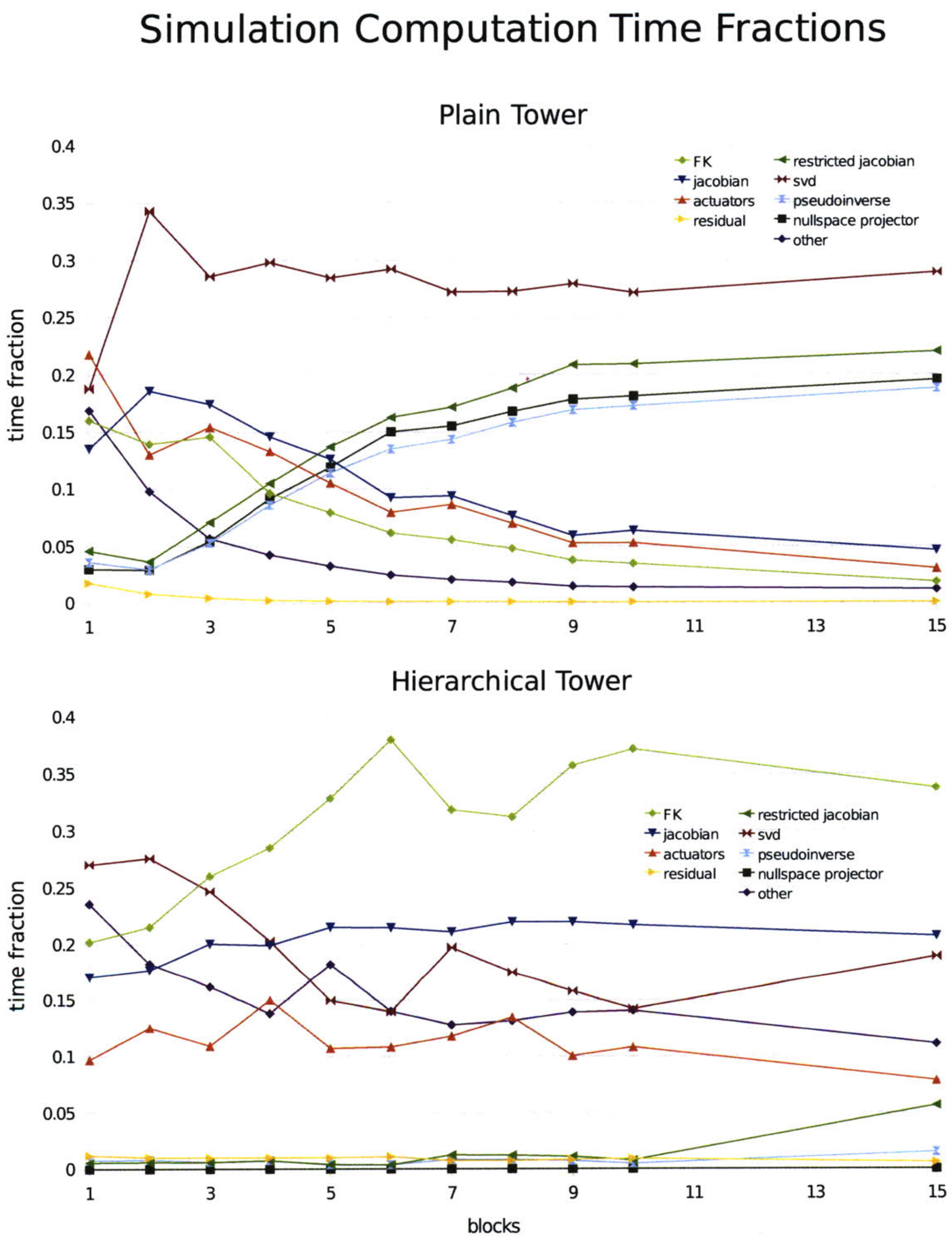


Figure 5-13: Unconstrained tower relative computation times.
See text for discussion.

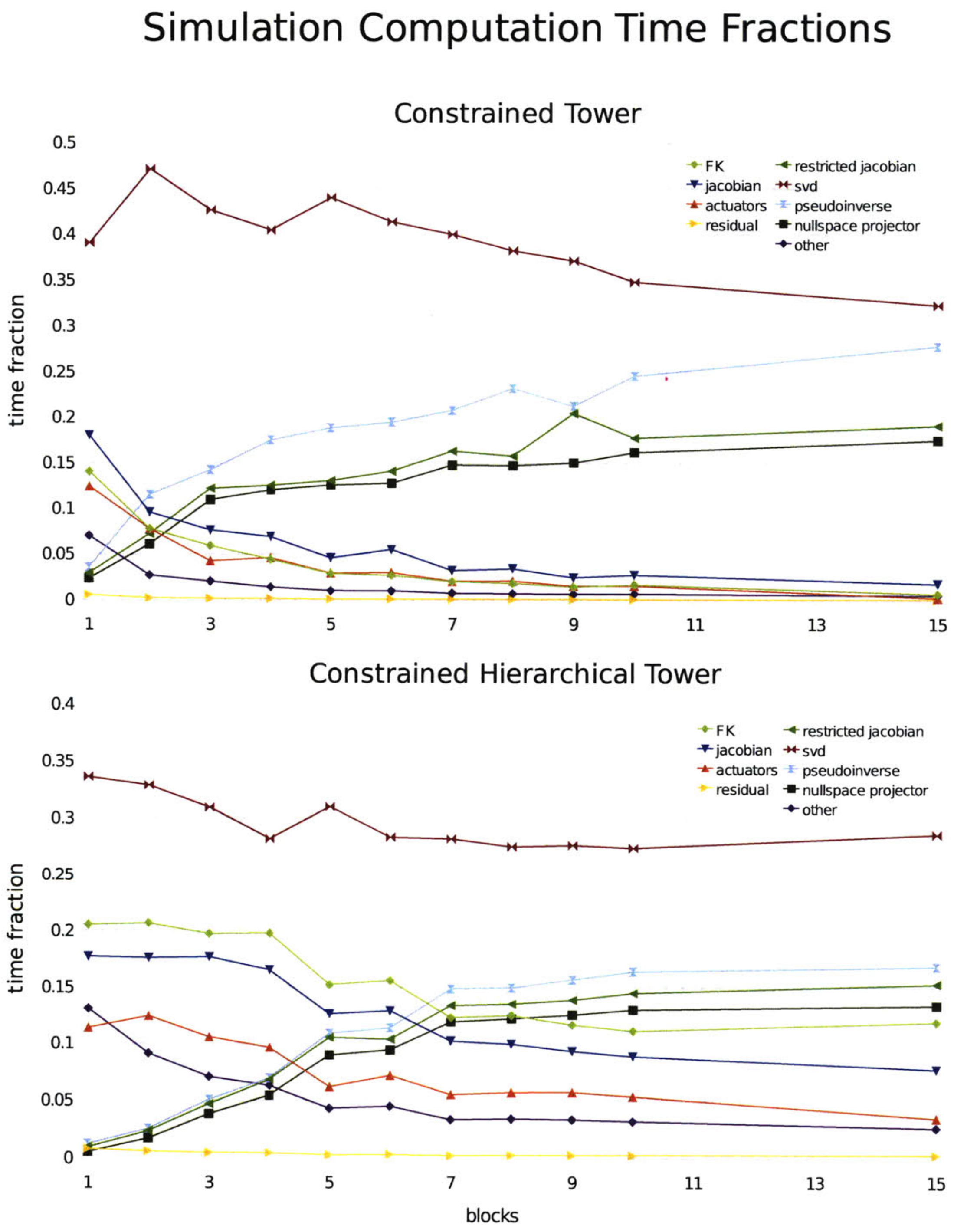


Figure 5-14: Constrained tower relative computation times.
See text for discussion.

framework in which a broad class of constraints can be graphically defined. And, for topologically large systems, the new idea of structure abstraction provides a workable means to break up the problem, easing both the operator's task and the computational burden on the motion computation system.

At least two prior constraint specification approaches are worth noting. One, primarily reported in the fields of graphics and animation, is to allow the operator to introduce geometric constraints—tangency, incidence, parallelism, etc. [123, 163]. The virtual articulation approach was motivated in part by this past work, but shows that, in robotics, where the base model itself is often an articulated system, a homogeneous model of only links and joints is sufficient in some practical applications.

Another idea has been to introduce optimization metrics, such as minimizing energy consumption, avoiding joint limits, or maximizing manipulability [94, 131]. These techniques are useful, but can be relatively blunt instruments—there is typically no direct way for the operator to arbitrarily customize the intended motion. Virtual articulations are one language that does permit such customization: the operator can constrain motion as much or as little as desired, and Kempe's theorem ensures that a large class of motions can be specified even using a compact catalog of virtual joints.

5.5.2 Previous High-Level Methods

Goal-based motion planning, e.g. the classic "piano moving" problem of achieving a target configuration among obstacles, is typically not directly applicable in cases where the operator would also like to specify more detailed or continuous aspects of the motion. Again, something more is needed if the operator needs to arbitrarily customize the motion *on the way* to a primary goal configuration.

Another high-level technique, programming-by-demonstration, allows more specific motion specification, but is hard to apply when the robot topology diverges from preexisting systems and biology. Thus it has been used with some success for humanoids (e.g. [109]), or when mimicking hardware is available, as in our TRACK device. But, short of extending TRACK into a full 36-DoF scale model, how to apply the technique to the whole ATHLETE mechanism? Or for dynamic-topology self-reconfiguring robots? Virtual articulations are not tied to any particular topology. (Also, the experiment in Figure 5-5 shows that that an integration of programing-by-demonstration with virtual articulations can have some of the advantages of both methods.)

5.6 Summary

This chapter has presented a novel class of applications for virtual articulations and kinematic abstractions: operating motion in articulated robots with large numbers (10s to 100s) of DoF. This is a challenging problem because such robots are capable of many different kinds of motion, though often this requires coordination of large numbers of joints. Prior methods exist for specifying motions at both low and high-levels of detail; the new method fills a gap in the middle by allowing the operator to be as detailed as desired in specifying the motion, and is independent of any particular robot or task topology.

The main idea is to use the mixed real/virtual interface to allow the operator to dynamically add and remove virtual joints and links, and inter-connect them with a model of the real robot. Virtual joints can both parametrize task motion and can constrain coordinated motion, by closing kinematic chains. Virtual links can represent task-relevant coordinate frames, and also serve as intermediate connection points for constructions involving multiple virtual joints.

For topologically large robots, structure abstraction can be applied similar to the way traditional algorithmic abstraction is used in the architecture of large software systems. This allows the operator to break the motion specification task into smaller sub-problems, and also enables the system to compute motions more rapidly via hierarchical decomposition.

Detailed applications were demonstrated for two different high-DoF robots:

- ATHLETE is a 36-DoF robot under development by NASA/JPL for use on the Moon. Several new kinds of inspection and manipulation motions were demonstrated for this robot, both on the hardware and in simulation. All of these would have been challenging to operate in more traditional interfaces—the intended motions are easily visualized, but communicating them to the system would have been a bottleneck.

- Multishady is a concept for a new chain-type self-reconfiguring robot. The mixed real/virtual interface was applied to operate deformation motions in an interactive simulation of a large tower construction involving hundreds of joints. Two levels of structure abstraction were used in this example, not only greatly simplifying the operator's job, but also making the difference between a responsive vs. sluggish interactive simulation.

Together, these demonstrate the practicality and scalability of the models and algorithms developed in Chapters 3 and 4, and of their implementation in the mixed real/virtual interface. Operating robots is not the only domain where these ideas are useful—Chapter 6 will present applications in the very different area of modeling and analyzing compliant and proprioceptive locomotion.

Chapter 6

Modeling Compliance and Proprioception

This chapter presents a second class of applications for virtual articulations and kinematic abstractions: modeling and analyzing reliable locomotion strategies that utilize compliance and proprioceptive sensing. This area is significantly different from the high-DoF operations studied in Chapter 5—for example, the ability to model quasi-static effects of joint stiffness and of gravity (Section 4.6) is essential here—but virtual articulations again enable homogeneous models which capture the essence of complex mechanisms. Whereas abstraction was previously used to hide complexity in the spatial structure of a robot, this chapter introduces *sequence abstraction*, in which the detailed temporal evolution of the topology of a mechanism is modeled at a higher level with a virtual stand-in.

This chapter will also introduce two novel robotic systems, the climbing robot *Shady* (Section 6.6 and Appendix J), and *Steppy*, a stair-stepping mini-humanoid (Section 6.7 and Appendix L). Both of these systems incorporate specific compliances and proprioception—sensing of the compliant motion—to achieve high reliability even under significant uncertainty. Shady is experimentally demonstrated to be 99.8% reliable in over 1000 climbing motions, and Steppy is over 90% reliable in 80 step-climbing trials even though the step height is uncertain and can vary from zero up to 10% of the robot's own height.

Using compliance to accommodate uncertainty has a long history, with highlights such as the RCC wrist [46], force-based compliant and hybrid control [100, 35], and passive dynamic/underactuated systems [101, 155]. The field is large: these papers are representatives from broad classes of work which together total hundreds, if not thousands, of publications. Lefebvre et al present a recent survey of active compliance in particular in [93].

How, precisely, does compliance confer uncertainty tolerance? One current viewpoint, which Paul termed *morphological computation* in [120], is that the passive physical dynamics of a robot can be considered a form of computation[1]. But this is an informal idea–consider the following recent observations from Pfeifer, Iida, et al:

> One problem with the concept of morphological computation is that while intuitively plausible, it has to date defied serious quantification efforts: We would like to be able to ask "how much computation is actually being done?" [121]. ... the notion of computation in the context of morphology or dynamics may in fact require fundamental reconceptualization [122]

The true physics of these systems—involving potential and kinetic energy, friction, contact, stiffness, damping, various forms of uncertainty, etc.—is often fairly complex and detailed. Thus, studying a morphological computation such as the interaction of Shady or Steppy with its respective environment (Figure 6-6 and 6-10) at the level of basic physics may be like trying to study a chemical reaction using only the tools of quantum mechanics—technically correct, but hard to see the forest for the trees.

The main idea of the application examples presented in this chapter is to demonstrate that virtual articulations can form the basis for higher-level models which still capture the essence of compliant and proprioceptive mechanisms. In particular, virtual joints can model

- *contacts* between the robot and the environment, where space of low-friction motion corresponds to the mobility of a *virtual contact joint*, and contact state

[1]Actually, this is just one facet of the current study of morphological computation.

evolution (making/breaking contact) corresponds either to topological connection/disconnection of the virtual joint or to hitting its motion limits. For example, the contact between Steppy's heel and the step is modeled using a combination of a Cartesian-2 and a point-slider joint (Figure 6-11).

- *uncertainties* in the relative pose of objects in the environment (or even, potentially, within the robot). When the uncertainty is mainly within some subspace of relative poses, it can be possible to align that subspace with the mobility space of a virtual joint, with virtual links modeling the objects themselves. A prime example is the uncertain height of the step which Steppy climbs, modeled with a virtual prismatic joint.

- *compliances* that may be co-located with the robot's joints or not. Though co-located compliance can be modeled on the primary joint itself, as I do for Steppy, in some cases it is useful to instead insert a virtual joint (in series with the primary joint) for the sole purpose of modeling compliance, as I do for Shady (Figure 6-4). For example, the latter arrangement allows separate expression of the compliance limits vs. the primary kinematic motion limits.

I form such mixed real/virtual models both for Shady and for Steppy, and I show that they can be useful both qualitatively, to help understand and communicate the essence of a morphological computation at a higher level than basic physics, and quantitatively—I instantiate the models in my mixed real/virtual interface system and use its simulation capabilities to predict the maximum uncertainty that can be tolerated in key stages of the locomotion of each robot. In both cases I compare the simulation results to actual experimental data (Figures 6-7 and 6-12).

The next three sections will discuss proprioception and compliance, the idea of sequence abstraction for modeling contact structure evolution, and using mixed real/virtual models to quantitatively predict uncertainty limits. The subsequent two sections document the Shady and Steppy application examples, with Sections 6.6.1 and 6.6.2 giving the details of sequence abstraction and uncertainty limit prediction for Shady, and Sections 6.7.1 and 6.7.2 giving the same for Steppy. Specific related work for structure climbing and stair-stepping is summarized with the description of each system; Section 6.8 gives further context in the broader study of compliance and proprioception.

6.1 Proprioception

When humans or other animals perform a high-uncertainty locomotion task, such as hiking on a rocky trail, it's not hard to be impressed with our actuation capabilities. But also impressive are our sensing abilities, which allow us to estimate the geometry and physical properties of the local terrain, our relation to it and to gravity, to select and land footfalls, and to evaluate their spaces of transmissible force. Of the six traditional senses—sight, smell, hearing, taste, touch, and balance—clearly vision, touch, and balance play important parts. However, the traditional senses represent just one of three sensing modalities [138]: *exterioception*. The other two modalities are *interoception*, which covers sensing of our internal organs, and *proprioception* which is the sense of position and force at our joints[2]. I hypothesize that, in addition to vision, touch, and balance, proprioception is also a key sense enabling reliable performance in high-uncertainty locomotion. Proprioception is the sense by which the morphological computation, performed in the mechanism of an articulated robot, can be "read out" for use in higher-level control.

This hypothesis is supported by the demonstrated reliability of Shady and Steppy, which rely on proprioception (and prior expectation) alone, albeit in fairly controlled settings. In more general natural locomotion tasks, proprioception could be one component of a bi-resolution strategy, where exterioception is used to form a coarse map and plan, and proprioception aids in fine motion.

Compared to exterioception, proprioception is particularly attractive in several ways:

- *Minimality*—proprioception can often use joint sensors which already exist for

[2]Sometimes the sense of balance is considered a proprioceptive, rather than exterioceptive, sense, and joint sensing proper is called *kinesthesia*.

purposes of closed-loop joint control

- *Directness*—proprioception can operate directly in the position domain which is highly relevant to many tasks; since it is based on physical contact, it requires fewer informational transformations than some other sensors (e.g. sonar, lidar, vision, etc.), and is arguably not susceptible to occlusion

- *Noise immunity*—joint sensors are "surrounded" by the engineered system of the robot, which can be easier to control to reduce sensor noise vs. exterioceptors that rely on propagation of a waveform through space, or in the case of tactile sensors, contact against a surface with highly variable properties.

Though proprioception can refer to sensing either force or position, I focus on the latter. Position sensors, such as potentiometers and encoders, are more common and lower-cost than similarly sensitive/reliable/durable force sensors. And the primarily positional quality of locomotion tasks makes position the more direct sense.

6.2 Compliance

The potential usefulness of motion sensing at a joint is coupled directly to how *compliant* the joint is.

Definition 64 The *impedance* [68] of a joint is the dynamic relation of (generalized) force transmitted across it as a function of the velocity of its motion. *Admittance* is the opposite relation: velocity as a function of force. *Stiffness* and *compliance* are the quasi-static correlates to impedance and admittance, respectively: stiffness is the force across a joint as a function of its pose, and compliance is the pose of a joint as a function of the force across it [69].

The more compliant a joint is, the further it can be deflected by collisions with the environment, all else being equal. Said another way, the mechanical state of a robot with zero compliance would be completely determined by its higher-level control commands, and would afford no natural dynamics by which morphological computation could proceed[3].

Conversely, the controllability of a joint—the ability of a robot's controller to impose specific motion on the joint—decreases as the joint becomes more compliant. Sensitive proprioception is thus facilitated by compliant joints, but this is in tension with maximizing control authority [126]. (The tug-of-war is modulated by the additional transmission properties of the surrounding linkage, which can affect the apparent compliance of the joint [68].)

The need to increase compliance to enable morphological computation and proprioception may be an undesirable cost from some perspectives. A large fraction of existing robots have historically been designed with low compliance for maximal control authority in controlled environments [126]; and designing actuators combining both low-impedance capability with other desirable properties such as compact size and high power density is not a totally solved problem. Nevertheless, research in high-performance low- and variable-impedance actuation (e.g. [60, 125]) is encouraging. Some useful systems can be built even with relatively low-performance schemes: the fixed compliance in Shady comes from springs in its drivetrains; for Steppy, compliance is varied on-line by changing the proportional gain constants in its position loop servo controllers.

6.3 Contact Sequence Abstraction

While compliant and proprioceptive mechanisms can be individually interesting from an engineering standpoint, scientifically, a more abstract perspective would be satisfying. This section introduces one technique—*sequence abstraction*—for using virtual articulations to model such mechanisms at a higher level than basic physics. These models are useful qualitatively, to aid understanding and communication of the essence of the mechanism, and they can also be used to make quantitative pre-

[3]In this statement, the contacts between robot and environment are also homogeneously considered to be joints, and may or may not be compliant. The examples in this chapter, and observations by prior authors [99, 19], show that this can be a reasonable viewpoint.

dictions, as detailed in the following section.

Often, key behavior occurs as contact is made and broken between different links. The physical reality of such processes can be complex, potentially involving multiple phases of differing contact configurations, dependence on detailed physical properties (friction, damping, elastic and plastic deformation, etc.) of the contacting bodies, and interplay between morphological and algorithmic (i.e. traditional) computing.

The idea of sequence abstraction is to replace such a temporal evolution by a simpler virtual construction. Sometimes, this replacement can capture the full evolution with a single assembly. The idea is best illustrated with an example: Figure 6-3 shows the actual sequence of contact configurations Shady uses to grip a bar, involving four phases; Figure 6-4 shows an abstraction of this sequence as a single virtual prismatic joint. Even though the contact evolution does not actually involve sliding at any stage, the cumulative effect of the gripping process is nevertheless to adjust the translation of the gripper along the bar.

Sequence abstraction, operating in the time domain, compliments the previously introduced structure abstraction, which operates in space. In each case, the abstractions are created by a human designer, just as a software architect builds traditional algorithmic abstractions. Both types of abstraction can be constructed in the current implementation of the mixed real/virtual interface (the figures in this chapter showing mixed real/virtual models, as in most of this thesis, are snapshots from the actual software), but must be manually added and removed as applicable—automatically deciding when to apply particular abstractions is a potential area of future work.

Because of its temporal nature, addition and removal can be more integral aspects of a sequence abstraction vs. for a structure abstraction. In the Shady example, gripping/ungripping is equivalent to connection/disconnection of the sequence abstraction. In some cases, making and breaking contact can be embedded into the sequence abstraction via the judicious setting of joint limits—the sequence abstractions for each of Steppy's feet, shown in Figure 6-10, use this technique.

6.4 Predicting Uncertainty Limits

Homogeneous models, using virtual articulations (and sequence abstractions) to represent contacts, compliances, and uncertainties, can make quantitative predictions about the capabilities of compliant/proprioceptive mechanisms. Though many questions can be asked, a core interest is often "how much uncertainty can be handled?" I introduce the following general procedure for predicting the limits of tolerable uncertainty:

1. model the relevant parts of the robot and task, including the elasticities of any involved compliances (Section 4.6.1) and the masses of any significant links (Section 4.6.2), in the mixed real/virtual interface

2. add sequence abstractions to model relevant contact interactions

3. parametrize major uncertainties with one or more virtual joints

4. identify key joint DoF limiting the compliant action, and their allowable range of motion

5. *operate* the virtual joints representing the uncertainties to explore the reachable workspace which does not violate the limits established in the previous step

Admittedly, the last two steps could be difficult, especially in more complex situations with multiple compliant DoF. Two broad strategies for identifying the limiting DoF, the penultimate step, are (a) to look for compliant joints that are both limited in their allowed compliance and also relatively high in sensitivity to the uncertainties; and (b) to introduce a joint where motion beyond some threshold corresponds to static instability. The Shady model gives an example of the former, and the Steppy model illustrates the latter.

The final step is a problem of workspace exploration. In the case of a single degree of uncertainty, this can be as simple as a manual scan. This was all that was necessary for Shady and Steppy. For more complex cases, traditional methods of workspace exploration [3] could potentially be applied.

6.5 Assumptions and Limitations

The above procedure for predicting uncertainty limits is not automated. While automatically inferring sequence abstractions is an interesting problem, they are useful even when manually designed. A main goal of this chapter is to contribute a new and general method for expressing and communicating high-level models of compliant motion, and for these purposes, manually developing the model is a reasonable part of the procedure.

Nor do I claim that the methods I develop in this chapter apply universally. In general, they can be expected to apply to compliant motions which (1) do not involve significant kinetic energy (i.e. are statically stable); (2) only involve contacts and compliances which can be reasonably modeled with the available joint catalog (Section 3.3.4); (3) have stiffness and gravity effects that can be modeled—and sufficiently calibrated—using the methods in Section 4.6.1. For example, general curved-surface rolling contact cannot be directly modeled, and non-zero stiffness is supported only on 1-DoF (revolute and prismatic) joints.

Despite these limitations, I have found these techniques useful both for Shady and Steppy. Both are interesting real robots, and the Steppy example in particular shows that these techniques can be applied even to fairly complex humanoid motion. As detailed next, for each robot, sequence abstractions help build a high-level qualitative models that hide low-level details. These models still have reasonable quantitative fidelity—their predictions of uncertainty tolerance compare reasonably well with experimental measurements on the actual hardware.

6.6 Application to Compliant Climbing

This section presents the first of two application studies in this chapter: contact sequence abstraction and uncertainty limit prediction for a novel proprioceptive compliant climbing robot using the mixed/real virtual interface.

The lab where the research in this thesis was conducted incorporates a large wall-window, about 4m tall and 8m wide, which has no shades to block glare from the sun. Instead of installing traditional shades (at significant cost), we designed, constructed, and experimentally tested a new compliant/proprioceptive climbing robot we call *Shady*, shown in Figure 6-2 [41, 158]. Shady locomotes on the aluminum frame of the windows and deploys a 0.6m diameter mylar sun-shade, thus providing active personal shading without significantly decreasing ambient light.

The main research interest here is not sun-shading, but rather the climbing action of the robot, which utilizes a combination of mechanical compliances and a proprioceptive control strategy to achieve high reliability. Shady locomotion proceeds as a series of grip-rotate-grip steps (Figure 6-1).

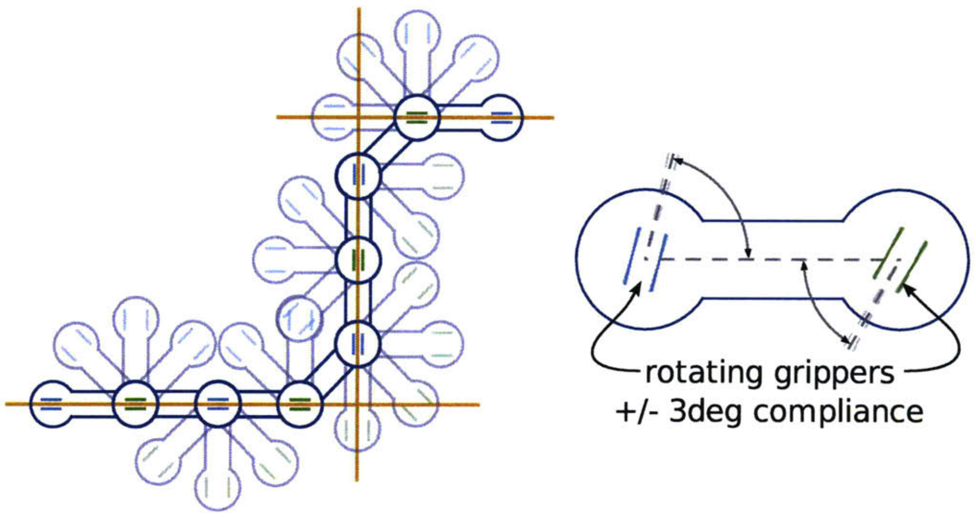


Figure 6-1: Concept of Shady locomotion.
Shady locomotes by alternately gripping and swinging on the members of a a structural framework, using two rotating grippers. The rotary actuators include mechanical compliances that are used in combination with a proprioceptive control strategy.

Shady is specialised to the particular geometry of our lab's window frame; however our group is also currently developing a similar robot with an additional central twist DOF for climbing more general 3D frameworks [173]. Shady's kinematics form the basis for a new self-reconfiguring robot concept we are also developing [41, 42], used in another application example in this thesis (Section 5.4).

We have performed extensive experiments [158] showing that the Shady is over 99.8% successful at completing locomotion primitives. Only two non-dangerous faults occurred out of over 1296 movements comprising several long climbing sequences, exercising all types of motion for the robot. A prior version of the hardware, which used contact and proximity sensors instead of compliance and proprioception, was only about 80% successful at a corresponding set of primitives.

Appendix J gives additional details on these experiments, as well as on the design and research context of the system. The remainder of this section will zoom in on two crucial aspects of Shady's operation: establishing a grip, and making a transition from vertical to horizontal climbing (i.e., executing a 90° turn). First, a sequence abstraction will be presented that models the relatively intricate grip procedure with a single virtual joint. Next, an instantiation of this model in the mixed real/virtual interface is used in simulation to predict the tolerable limits of uncertainty on the actual vs. expected bar position, and this prediction is shown to match well with experiment.

6.6.1 Transition Grip Sequence Abstraction

Since Shady always locomotes on the particular framework in our lab, uses a map to form a coarse motion plan. But there is inevitably some uncertainty remaining in the fine motion, e.g., due to map errors, misalignment of the initial pose, accumulated slippage, unmodeled structural compliances, etc. If no sensing were performed to correct for this then Shady would eventually fail to grip because the bar would be too far from the expected position. While it is possible that the bar could be missed entirely, a misaligned grip can also fail due to a walk-off effect, as shown in Figure J-4.

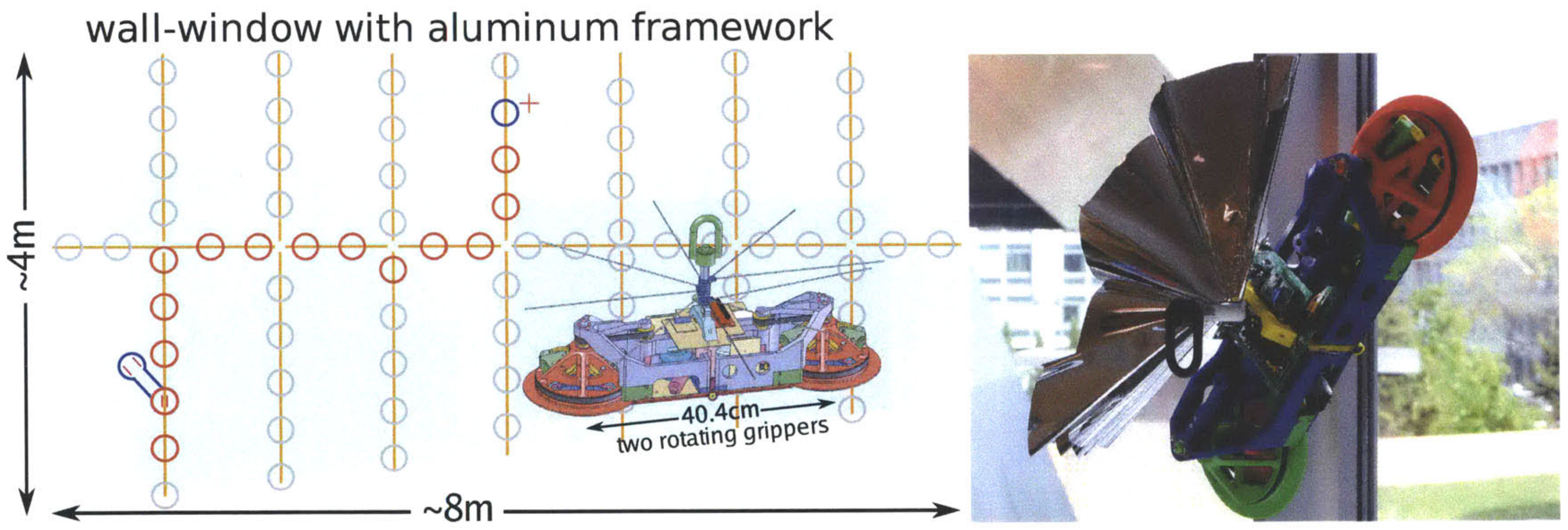


Figure 6-2: The climbing robot *Shady*.
Left: A map of our lab's trapezoidal wall-window with a set of grip points reachable from a starting pose (light circles) and a locomotion path (dark circles) to a commanded target location. Inset: CAD representation of Shady, mylar shade not shown. Right: Photo of the hardware in action.

Shady avoids this eventuality by ensuring that each grip is aligned, using a combination of compliance, proprioception, and a particular four-phase grip sequence, illustrated in Figure 6-3 and given in detail as Algorithm J.1, GRIPREFINEMENT. Essentially, each grip is established twice; the first time to propriocept the precise bar location, and the second time for the final grip.

The actual mechanics of the grip sequence involve several contact situations, friction, material elasticity, and the interaction (via proprioception) of morphological computation and the control algorithm which manages the sequence. However, the ultimate effect is easily stated: the gripper is aligned to the bar and translated along it as necessary, via symmetric rotation in of both grippers.

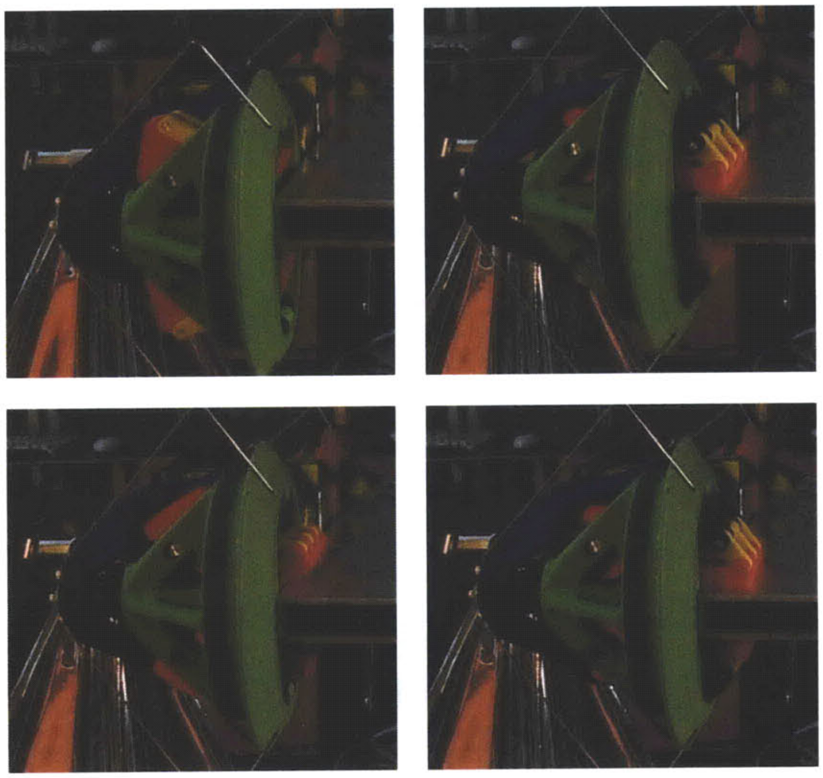


Figure 6-3: GRIPREFINEMENT (Algorithm J.1) running on the Shady hardware. Four key frames are shown from a video of the GRIPREFINEMENT algorithm in action. Top left: the initial un-gripped state; note that the gripper is misaligned slightly downwards from the bar. Top right: the initial grip is established, and the actual bar location is proprioceptively sensed; note the walk-off (see Figure J-4). Bottom left: the initial grip is released and the robot has adjusted to the actual bar location. Bottom right: final grip, without walk-off.

I propose that a reasonable abstraction of the whole sequence, hiding the four phases as well as the lower-level physical details, is a single virtual prismatic joint connected in series between the closing gripper and a virtual link representing the bar, as shown in Figure 6-4. The translational DoF of the joint is aligned with the axis of the bar, representing the effective displacement that GRIPREFINEMENT will produce (even though no actual physical sliding may occur at any step).

If the gripper rotation DoF are also modeled, as shown in the figure, then the mixed real/virtual interface will automatically compute a local assembly (Section 4.1) motion when the virtual joint is connected because it closes a kinematic cycle (assuming that the two bars are represented using a single virtual link, or at least connected links as shown in the figure). This motion will have the same effect as the GRIPREFINEMENT algorithm: the grippers will rotate as necessary to align the closing gripper with the bar.

6.6.2 Transition Uncertainty Limits

Once the contact abstraction is assembled, local control (Section 4.2) can be invoked to operate motion in the resulting closed-chain linkage. If the input for this motion is arranged to be a virtual joint representing an uncertainty of interest, and if the gripper rotation compliances are also modeled, the result is a setup that addresses most of the steps presented above for predicting the limits of uncertainty tolerance. The main remaining questions are: "what is the main uncertainty?" and "what effects limit the compliant action?"

For Shady, the case of locomotion along a straight bar is not particularly difficult. The straightness of the bar supplies a strong prior, and, provided that the robot's own internal joint position controllers are sufficiently accurate, the uncertainty at grip will be low. This holds independent of the absolute orientation of the bar due to the gravity compensation procedures incorporated in Shady's control (Section J.3.1), which automatically pre-load the compliances to avoid sag.

A more interesting case occurs when Shady makes a transition, e.g. from vertical to horizontal climbing. In this case, any linear misalignment (slip) which may have

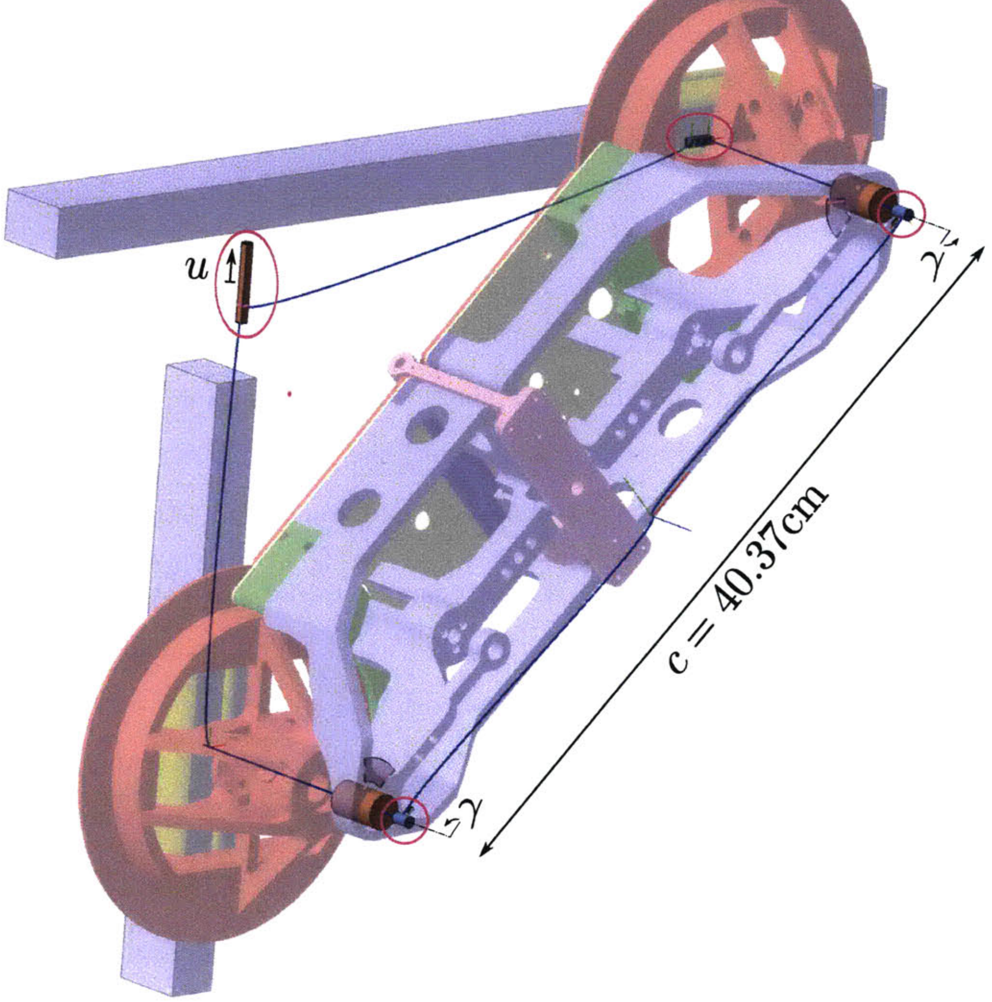


Figure 6-4: Virtual articulation abstraction of the Shady grip sequence. The actual grip sequence, shown in Figure 6-3, involves four separate stages. Further, at each stage, the actual contact mechanics depend on detailed geometry, friction, and material deformation. All of these details are hidden by a simple high-level model of the whole contact process as the addition of a virtual prismatic joint (circled, upper right) modeling the contact. The translational DoF is aligned with the upper bar, corresponding to the effective translation of the gripper due to the GRIPREFINEMENT process. Also shown in this figure are the virtual joints representing the compliant rotations (circled, right and bottom) and the virtual joint representing the uncertain position of the upper bar (circled, upper left). The CAD model of the robot geometry is for visualization only; the mixed real/virtual interface simulation uses only the articulation model. The marked quantities u and γ are used in Eq. 6.1.

accumulated while climbing along the vertical bar is equivalent to uncertainty in the height of the horizontal bar. While such slip is generally low, it is not always exactly zero; thus, the first question is answered: a situation with potentially significant uncertainty occurs in a vertical-to-horizontal climbing transition, and that uncertainty is the deviation of the height of the horizontal bar from its expected position.

For the second question, I hypothesized that the limiting effects in Shady would be the maximum and minimum travels of the compliances, which are relatively small at $\pm 3°$ each. I thus configured a model of Shady, its compliances, the prismatic contact sequence abstraction, and another prismatic joint representing the uncertainty in the mixed real/virtual interface, and I simulated the motion of the uncertain joint while watching the deflections of the compliant joints, as shown in Figure 6-6.

One important detail was to correctly model the pre-load of the compliance in the supporting joint, as shown in Figure 6-5. With that added, I found that the maximum travel u of the uncertain joint was about +15mm and −10mm, with the asymmetry due to the pre-load: the lower bound is reached when the supporting gripper hits the limit of its rotational compliance, which was already reduced due to the pre-load; the upper bound occurs when the other (gripping) joint hits a compliance limit.

The uncertainty tolerance is thus parametrized and quantitatively predicted. To establish the accuracy of this prediction I conducted a corresponding experiment on the actual hardware, the results of which are shown in Figure 6-7. I first configured Shady in the nominal pose for a vertical-to-horizontal transition, with the upper bar at the expected position. Then, I intentionally introduced varying amounts of slip into the position of the supporting gripper, with 3 trials for every 5mm increment of slip in each direction (up and down), starting 2.5mm from nominal. To detect failure of a grip I used both qualitative visual assessment and a quantitative comparison of the ground truth slip vs. a back-computation of the sensed slip from the propriocepted deflections in the gripper rotations after the first (test) grip in the GripRefinement procedure. If γ is the sensed deflection (once the preload is subtracted, γ is theoretically equal for both grippers, and this was supported to within a fraction of a degree in most trials) and u is the slip, then the computation is

$$\begin{aligned}
c/\sqrt{2} + u &= c\cos(\pi/4 + \gamma) \text{ with } c = 40.37\text{cm the distance between rotation axes} \\
&= c(\cos\pi/4\cos\gamma - \sin\pi/4\sin\gamma) = c/\sqrt{2}(\cos\gamma - \sin\gamma) \\
u &= c/\sqrt{2}(\cos\gamma - \sin\gamma - 1) \qquad (6.1) \\
&\text{using small angle approximations } \cos\gamma \approx 1 \text{ and } \sin\gamma \approx \gamma \\
u &\approx c/\sqrt{2}(-\gamma). \qquad (6.2)
\end{aligned}$$

The results show that up to about +18mm and -12mm of slip can be tolerated, in agreement to within 3mm of the predictions from the virtual articulation model. In addition, within those limits, the back-computed slip based on proprioception was generally within a few mm of ground truth.

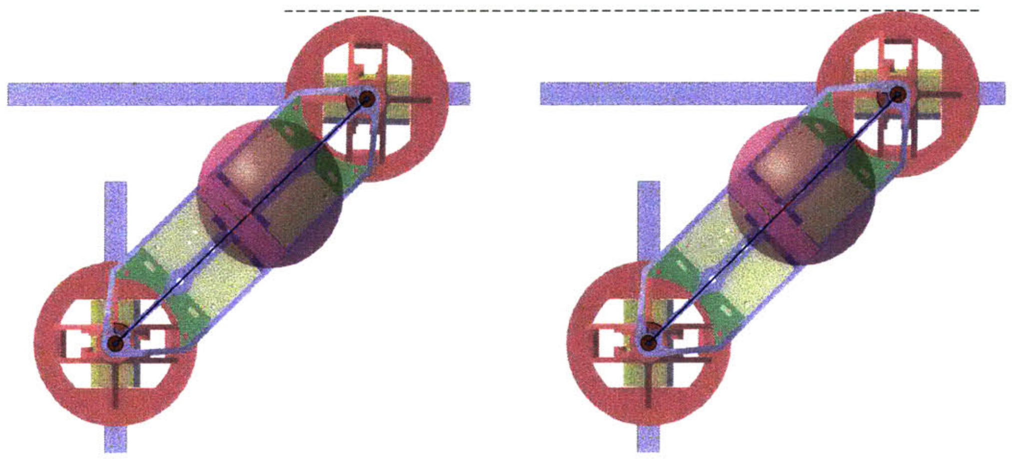


Figure 6-5: Simulating gravity compensation in a Shady transition.
One nuance at a transition from vertical to horizontal climbing is due to the effects gravity compensation (Section J.3.1). The compliance in the supporting (lower left) gripper rotation must be pre-loaded due to the cantilevered mass of the robot body. This decreases the amount of remaining compliance in one direction. The actual amount of compensation is slight, about 1°, but this is fully 1/3 of the available compliance in that direction. Modeling both the mass of the moving part of the robot (represented graphically as a translucent ball) and the actual stiffness of the compliant joint enables the mixed real/virtual interface to simulate the compensating rotation, which has been applied in the right snapshot, but not the left.

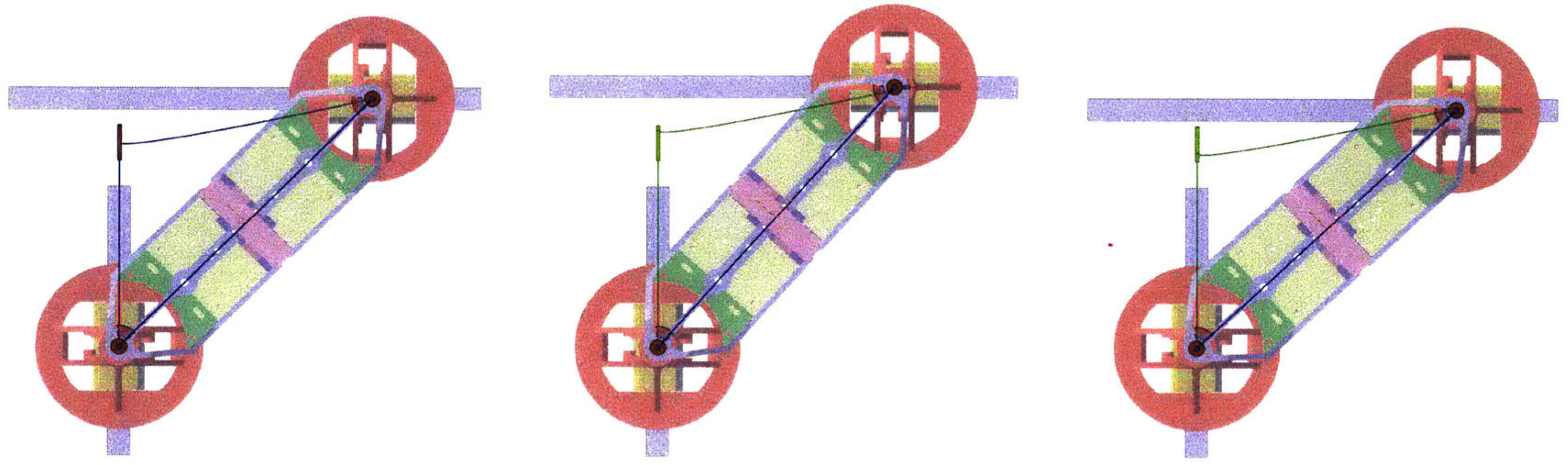


Figure 6-6: Predicting uncertainty limits at a Shady transition.
This figure shows an instantiation of the uncertainty limit prediction process (Section 6.4) for the case of Shady making a transition from vertical to horizontal climbing. The primary uncertainty here is the location of the horizontal bar relative to the grip location on the vertical bar, modeled as a virtual prismatic joint with translation u (Figure 6-4 gives a close-up view with labelled quantities). The joint compliances are modeled with rotation γ, and the grip sequence of the upper bar is abstracted with a virtual prismatic joint as developed in Section 6.6.1. The limits of tolerable uncertainty are found by manually operating the joint representing the uncertainty, starting from $u = 0$, until any limit is hit in either of the compliant joints. For positive motion this occurs at about $u = 15$mm, when the upper compliant joint hits a limit, but negative motion is limited to $u = -10$mm, when the lower compliant joint hits a limit. The asymmetry is due to the preload of the lower compliant joint for gravity compensation (Figure 6-5). These predictions are within 3mm of the actual measured limits (Figure 6-7).

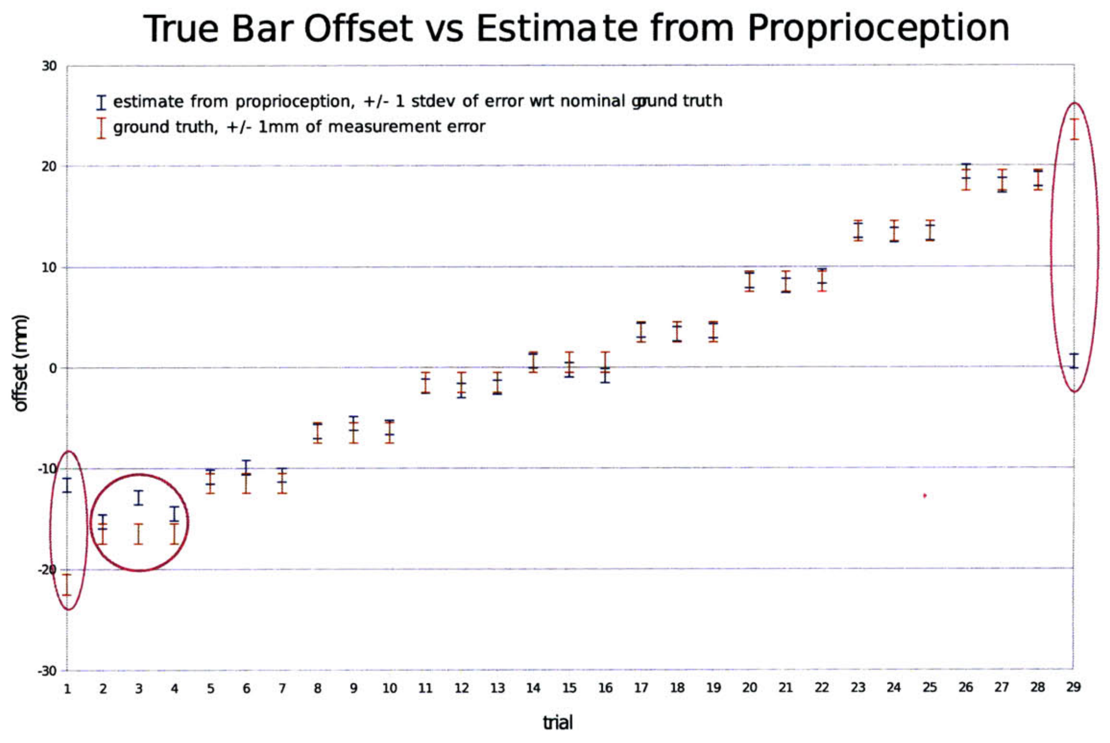


Figure 6-7: Shady transition experiment data.
This plot shows both the ground truth and the back-computation (Eq. 6.2) via proprioception of the position of the upper bar, quantity u in Figure 6-4. The experiment varies u above and below nominal in 11 steps, up to about ± 25mm. The data show two important results: first, the limits of tolerable uncertainty are roughly $+18$mm and -12mm; second, within those limits, proprioceptive sensing is accurate to within a few mm. Circled trials indicate limit-exceeding configurations where proprioception and ground truth diverge due to unmodeled effects, such as bending in structural members of the robot. A corresponding prediction using a homogeneous model in the mixed real/virtual interface (Figure 6-6) predicted uncertainty limits of $+15$mm and -10mm.

6.7 Application to Compliant Stair-Stepping

This section presents a second application of virtual articulations and sequence abstraction in compliant/proprioceptive locomotion, in this case to the task of statically-stable stair-stepping in 3D with an 18-DoF mini-humanoid called *Steppy* (Figure 6-8).

Steppy accommodates significant uncertainty in the height of the step, and, like Shady, uses only compliance and proprioception to achieve high reliability (though not as high as Shady). An experiment with 80 trials indicated about a 90% success rate where the height was only known a-priori to lie in the range 0 to 35mm, which is about 10% of the robot's own height (36cm). In each trial, Steppy starts in double support facing the step on a lower horizontal platform, and ends in double support on an upper horizontal platform, as shown in Figure 6-9. Success is defined as termination of motion with the robot in stable double support on the upper platform.

Details of the apparatus and control strategy are given in Appendix L. Essentially, Steppy assembles pre-defined motion plan fragments on-line based on position feedback from its joints at certain key points where contact is expected with the step. The impedance of some joints is selectively lowered at these points by decreasing the proportional gain in the position servo control loop (Section L.1.1), allowing collision with the step to backdrive the joints and thus provide information about the step height.

Climbing a single step in a laboratory environment, where the only major uncertainty is the step height, is not nearly as challenging as locomotion in natural terrain. Nevertheless, the aim of this experiment was to extend and generalize the compliant/proprioceptive methods employed in Shady to a broader class of articulated robots. Also, to the best of my knowledge, this is the first report of a stair-climbing humanoid that accommodates variation in step height up to nearly 10% of its own height, with experimentally verified reliability. Additional research context for Steppy is also reviewed in Appendix L.

In parallel with the presentation for Shady, the following two sub-sections will detail (1) particular sequence abstractions for the contacts of Steppy's feet with the environment, and (2) a simulation of Steppy in the mixed real/virtual interface, together with the contact sequence abstractions and a virtual joint parametrizing the uncertain step height, used to predict the maximum and minimum permissible step heights. These predictions are again compared with results from a corresponding hardware experiment.

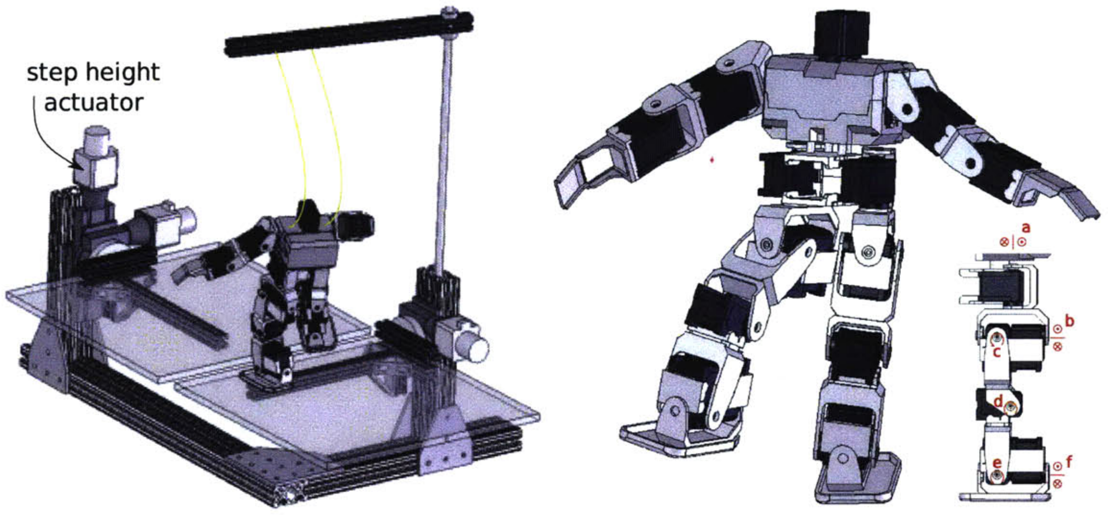


Figure 6-8: Steppy and the variable-height step setup.
Left: Experimental setup for step climbing. Each platform is about 45×45cm, and their relative height is adjustable with the indicated linear actuator. Two additional actuators, which vary the platform angles, are currently used for levelling but may also introduce additional degrees of uncertainty in potential future experiments. The robot can operate fully un-tethered; the yellow bungee cords help prevent damage in a fall. Right: *Steppy*, an 18-DoF 3D mini-humanoid, is based on the low-cost Robotis Bioloid. Steppy stands about 36cm tall and weighs about 2.0kg. We have modified the off-the-shelf hardware in several ways, as described in Appendix L, and added a Bluetooth® remote-brain soft real-time control environment with 10Hz system-level update rate. Leg joint axes a-f are hip yaw/roll/pitch, knee pitch, and foot pitch/roll.

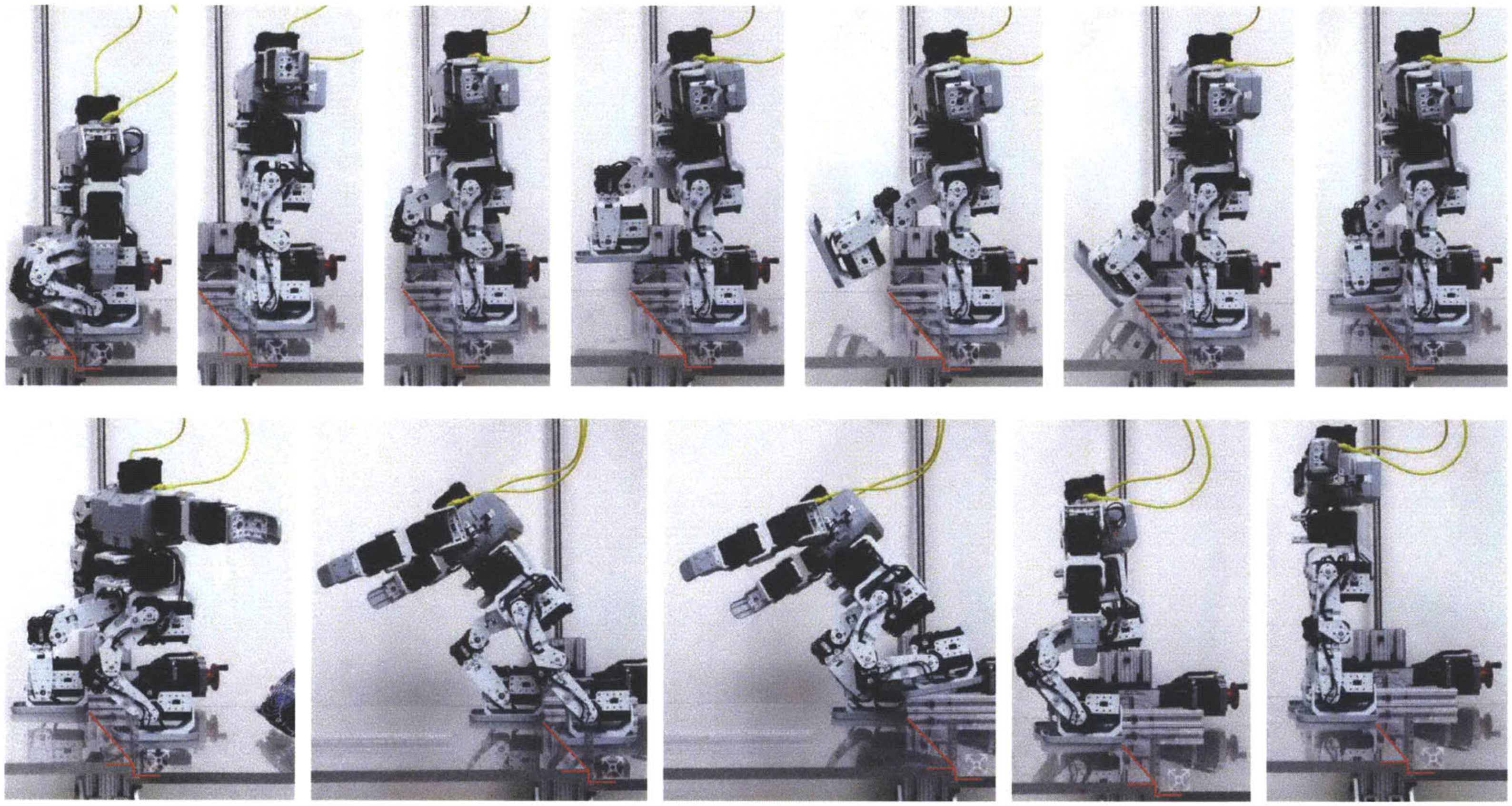


Figure 6-9: Steppy hardware taking a step.
Steppy autonomously and reliably climbs a step with uncertain height in the range 0-35mm, or about 10% of the robot's own height, using a proprioceptive sensing strategy. The step geometry has been manually outlined in red.

6.7.1 Contact Sequence Abstraction

For Steppy, two contacts must be modeled, one for each foot. Figure 6-10 shows my design for each. The left foot, which supports the robot until the upper step is contacted, is modeled to contact the lower plate with a virtual revolute joint aligned with its left edge. This is a significant simplification of the actual contact physics: in reality, the foot is either resting flat on the plate, in which case a slightly sticky silicone rubber coating on the bottom of the foot effectively immobilizes it; or, if the robot has begun to tip, the foot is pivoted up on one of its edges or vertices. Though there are a number of possible behaviors in the latter case—depending on which edge or vertex is in contact, the specific pose of the robot (which will determine the contact forces), and the frictional properties of the foot and plate—observations of the hardware indicated that a common tipping mode is rotation about the left edge of the foot.

The motion limits of this revolute contact joint are set so that the foot is aligned flat with the plate at one extremum of travel. Thus, both contact configurations (flat vs tipping) are abstracted with a single virtual joint; "switching" from one to the other is equivalent to the joint entering or exiting a rotation limit.

A similar idea is applied to the contact of the other foot. Here, the robot's motion during the proprioceptive phase of the step procedure (SteppyStep, Algorithm L.1) ensures that the right foot will be in one of two contact configurations: either it will be dangling above the upper plate, or it will contact the plate at the left heel corner. Both of these possibilities are captured in a single point-slider joint, with the translational DoF perpendicular to the plate, and the spherical rotation DoF centered at the heel corner. The lower limit of translation is set to correspond with the heel corner contacting the plate. A final detail is that, unlike the support foot, observation of the hardware indicated that the contact forces were not typically large enough to prevent sliding in the plane of the plate. An additional Cartesian-2 joint, inserted between the point-slider and the virtual link representing the plate, models this 2D sliding freedom. The contact sequence abstraction for the right foot is the resulting virtual assembly of a point-slider and a Cartesian-2 joint, plus the requisite intervening virtual link.

6.7.2 Step Height Uncertainty Limits

The primary uncertainty for Steppy is, by design, the height of the step. This is naturally parametrized with a virtual prismatic joint which sets the step height (Figure 6-10). One reasonable question is "how low, and how high, can the step be?"

To make this a bit more precise, consider the detailed operation of the original Steppy experiment, presented as Algorithm L.1, STEPPYSTEP. The robot is sequenced through a series of poses which are designed to collide with successively lower steps. Each such pose sets not only the positions of all robot joints, but also their stiffnesses, by changing the proportional gain constants in the position loop servo controls. Thus, a more specific question is "for a given robot pose, what are the minimum and maximum permissible step heights?"

A lower bound on the minimum detectable height is the first height at which contact is made at the right heel, i.e., when the point-slider joint hits the lower limit of its translation. The maximum detectable height is upper-bounded by the lowest height which causes the robot to lose static stability (i.e., to fall). This, in turn, is lower-bounded by the minimum step height which causes the revolute joint representing the left foot's contact to depart its motion limit. I hypothesized that these bounding events were reasonably close to the actual detection limits.

Proceeding with these hypotheses, I conducted simulations using the mixed real/virtual interface to simulate slowly increasing the step height upwards from zero, as shown in Figure 6-11. Three separate poses were tested, the same poses as used in the actual STEPPYSTEP procedure. In each case, the heights were recorded at first contact and at fall. The results are plotted in Figure 6-12, along with measurements from corresponding experiments on the actual hardware. Three trials were performed on the hardware for each pose, and 10 in simulation. As can be seen in the figure, four out of the six main predictions from the simulations match the data collected from hardware experiment to within about 5mm.

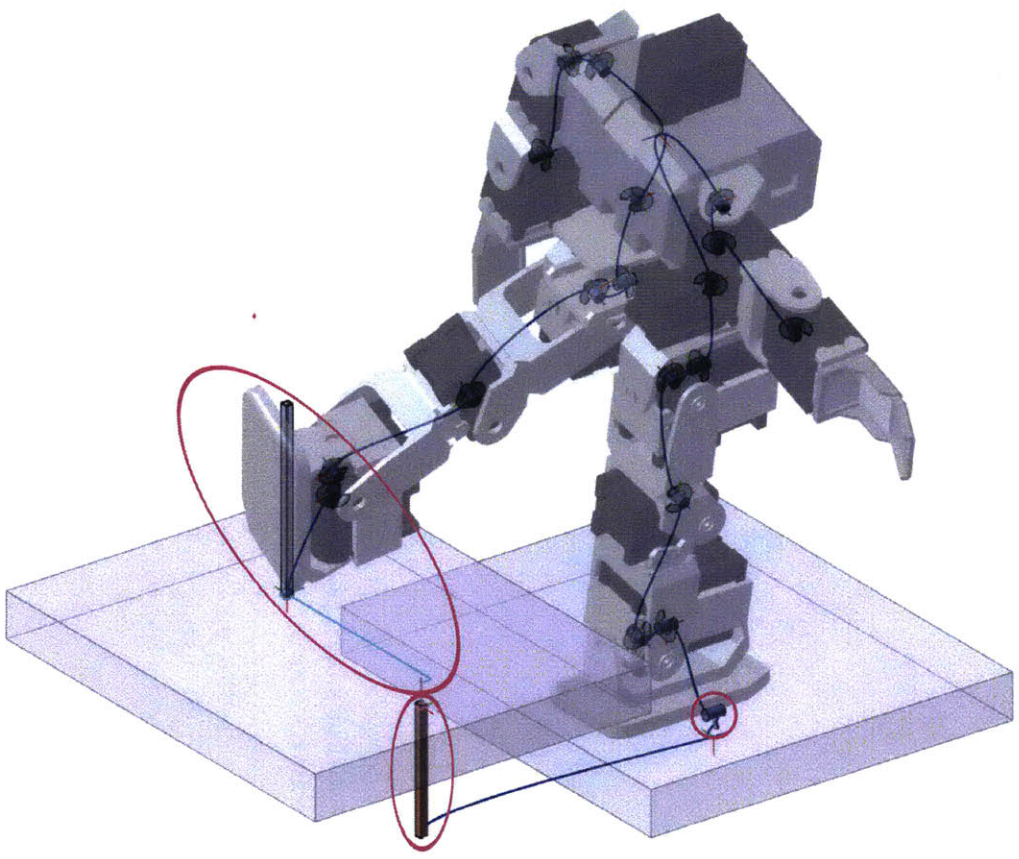


Figure 6-10: Mixed real/virtual model of Steppy.
Each of Steppy's 18 revolute DoF are modeled in a tree topology, rooted at the torso. The contact between the left foot and the lower plate is abstracted with a virtual revolute joint—at one rotation limit the foot is parallel to the plate, and rotation beyond a small threshold corresponds to tipping. The right foot may or may not be in contact with the upper plate depending on the pose of the robot and the step height (see Figure 6-11), and is abstracted with a serial assembly of a Cartesian-2 joint, for sliding in the plane of the upper plate, and a point-slider joint, which models both the rotational freedom of the foot pivoting about its contact point, as well as the making of contact when the translational DoF hits its lower limit. The step height, which is unknown to the robot, is modeled with a virtual prismatic joint.

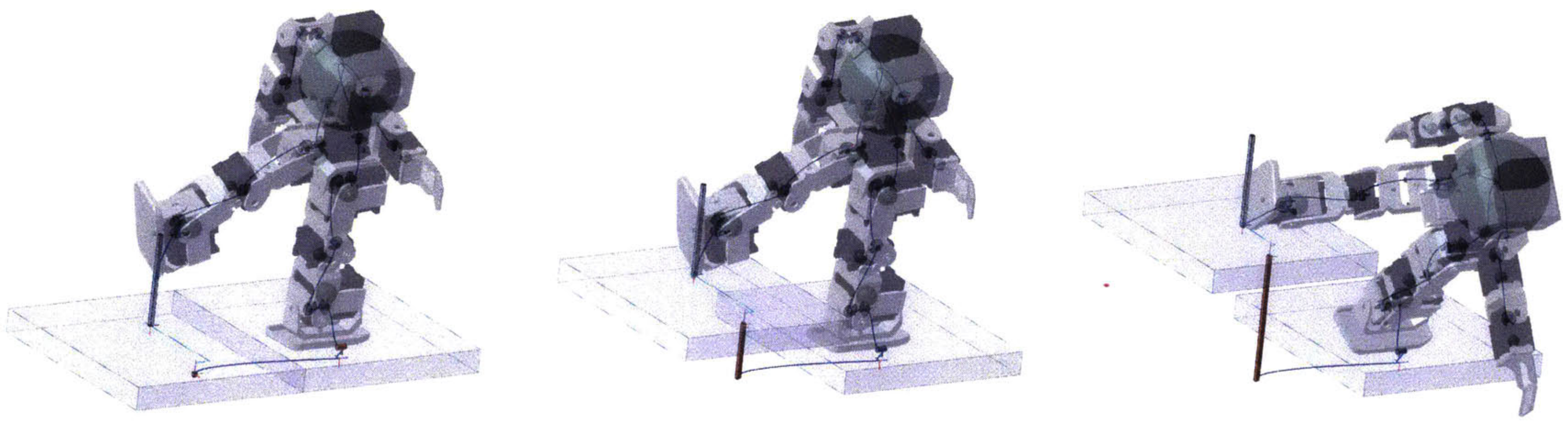


Figure 6-11: Predicting uncertainty limits for a Steppy pose.
Given a specific pose for the robot, which in this case includes both position and stiffness commands for all joints, the minimum detectable step height occurs when the right foot first makes contact, and the maximum height is the lowest height for which the robot becomes unstable. Both can be quantified in terms of the DoF of virtual articulations modeling the contacts (shown in more detail in Figure 6-10). The limits are manually explored by operating the virtual joint representing step height, increasing slowly from 0 (left snapshot) to first contact (middle) and ultimately to fall (right). This procedure was performed for each of three sensing poses involved in the step motion, with the results plotted in Figure 6-12 along with corresponding measurements from a hardware experiment.

The predicted vs. observed fall heights for two poses do not match well. One possible explanation is that the simulated model was not sufficiently calibrated to the actual robot. Another effect seems to be at play, and this also explains why the simulation differed in multiple trials for the same initial pose. I have as yet only implemented an interactive version of the simulator; an off-line batch mode is future work. This introduces some non-determinism because the amount of computation allowed at each step is limited by wall-clock time. Since I use a standard preemptive multitasking operating system, the actual computational cycles afforded to each step of simulation can and do vary from run to run. Thus, the simulated state at the end of each step can vary slightly between runs, and in sensitive cases these small variations can become magnified as the simulation proceeds.

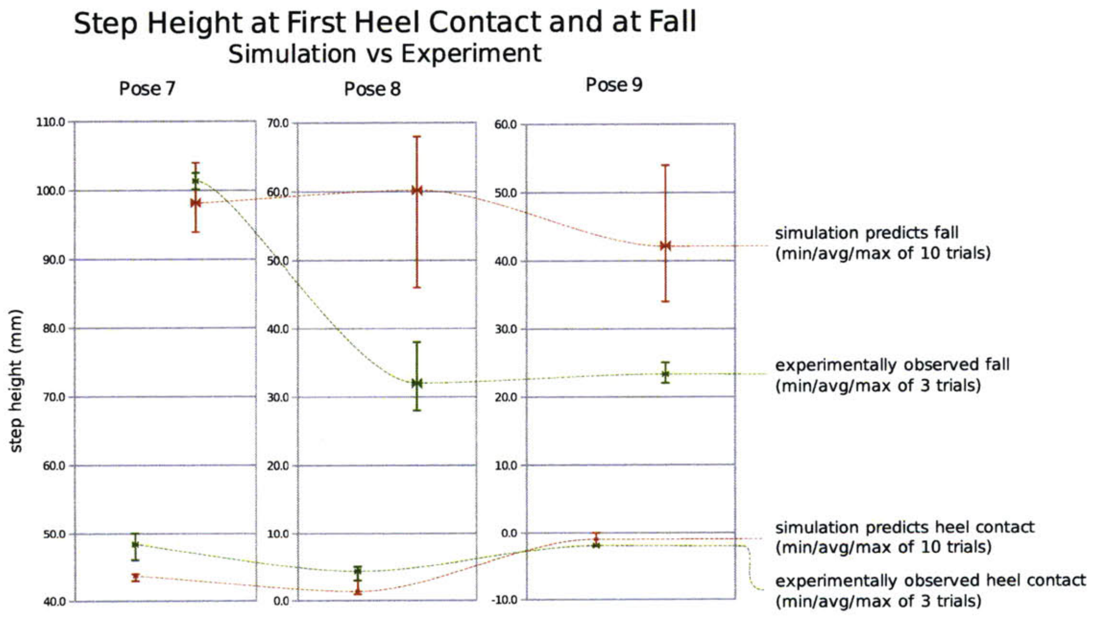


Figure 6-12: Results of Steppy simulation and hardware experiment.
This figure shows results from both simulations in the mixed real/virtual interface and actual trials on the hardware. The procedure, shown in Figure 6-11, was to set the robot pose (position and stiffness of all joints), and then slowly raise the step until contact (lower plots) and fall (upper plots). Three separate poses were tested, corresponding to actual poses used in the STEPPYSTEP sensing sequence (Algorithm L.1). While the simulation matches hardware experiment well in most cases, in a few the error is significant. See text for further discussion.

6.8 Research Context in Compliance and Proprioception

This section considers the contributions of this chapter in the broader context of research in compliance and proprioception. In terms of others who have specifically considered using virtual articulations, the main prior work which stands out is Bruyninckx's thesis [19]. That work also used virtual joints to model contacts, but mainly in the instantaneous case: Bruyninckx's virtual assemblies are valid for a particular contact configuration, but would typically need to be reconfigured after any motion. In contrast, my approach shows that in some cases a virtual articulation sequence abstraction can persist through both motions and changes in contact configuration. Bruyninckx also explored using virtual joints to model uncertainties, but again mainly takes a finer level of magnification than I do, and considers perturbation-type uncertainties in the parameters of his model vs. gross geometric uncertainties (like the step height in Figure 6-10). Finally, Bruyninckx does not seem to have also modeled compliances using the same homogeneous virtual articulation models.

It seems that few prior researchers have used the term "proprioception", though in some cases their work is nevertheless related. And, as covered in the introduction to the chapter, there is a long history of compliance in robotics [46, 93]. Particular areas of interest have been active compliance [100, 35], planning compliant motion [92, 172], avoiding sensing entirely [48], and under-actuated mechanisms involving completely passive DoF [101, 155]. The proprioceptive approach differs from sensorless work in that it actively uses measurements of the compliant motion. Mechanisms like Steppy and Shady also differ from many under-actuated systems in that most joints can achieve significant impedance, but can still participate in proprioception.

While I explore the particular case where both sensing and compliance are co-located with the primary joints of an articulated robot, many other researchers have opted instead to add additional compliant elements and sensors at other parts of the robot structure. For example, Lefebvre, De Schutter, Bruyninckx, and their collaborators have recently presented a series of papers which develop a theory of planning active sensing strategies for autonomous compliant motion [92], including some initial hardware experiments for a manipulation task [102, 92]. Though Lefebvre et al's framework may be conceptually applicable to joint compliance and proprioception, they have mainly reported using rigid-joint robots combined with a passively compliant end-effector and a 6-axis load cell. Also, it appears they have focused mostly on planning exploratory motion and have not yet used the acquired data to adjust subsequent task motion.

However, others have considered adjusting motion plans on-line based on information gained from sensing during compliance; for example, Xiao and Volz introduced a theory of on-line dynamic re-planning of localized *patch-plans* in [172], but did not specifically consider proprioceptive sensing. In [40] Desai and Volz described the use of movability tests, similar to the strategies in both Steppy and Shady, for detecting termination in guarded motions. Spreng [146] extended this idea to an exploration of ambiguous contact with probabilistic movability tests.

The specific relevance of Shady and Steppy with respect to other structure climbing and stair-stepping robots is further detailed in Sections J.1 and L.1.

6.9 Summary

This chapter compliments Chapter 5 by demonstrating applications for virtual articulations and kinematic abstractions in a second area of robotics, in this case reliable compliant/proprioceptive approaches to locomotion in uncertain environments. Both of these applications chapters are built upon the models and algorithms developed in Chapters 3 and 4, and show applications implemented in the mixed real/virtual interface system.

The particular robots studied in this chapter are *Shady*, a novel structure climbing robot, and *Steppy*, a mini-humanoid that climbs a step of uncertain height. Both systems have been implemented in hardware and have experimentally demonstrated reliability: Shady was 99.8% reliable in over 1000 climbing motions, and Steppy was about 90% reliable in 80 stair-stepping trials. This reliability is conferred in each case by a combination of specifically designed compliances and control strategies based on proprioception, the only sensing used in either system.

Virtual articulations can help study compliant locomotion by enabling homogeneous models where uncertainties, compliances, and contacts are all incorporated in a single kinematic model with the robot and task. Unlike the high-DoF operations applications studied in Chapter 5, quasi-static modeling of elastic and gravitational potential energy is usually important in the complaint locomotion domain (the systems I consider are statically stable; faster-moving dynamic locomotion would likely also require modeling of kinetic energy). Also, whereas the main abstraction of interest for high-DoF operations was in the spatial domain, here I introduce *sequence abstraction*, which applies in the time domain, and enables simple virtual models that capture the essence of potentially detailed physical contact interactions.

I also introduce the perspective that proprioception can be considered a way for control algorithms to "read out" the results of *morphological computation* performed in the mechanism of the robot itself. While many particular compliant (and to a lesser extent, proprioceptive) mechanisms have been reported in the literature, it is still an open problem to build a scientific understanding of such systems that is deep, quantitative, and generally applicable. While basic physics is technically sufficient to model these systems, even laboratory examples can be very intricate at this level. Virtual articulations and kinematic abstractions can help us see the forest over the trees.

Chapter 7

Conclusions and Future Work

This thesis has explored the possibilities enabled by virtually relaxing the usual assumption that the kinematic topology of an articulated robot—the connectivity of its links and joints—is fixed. The ability to add, remove, and reconfigure *virtual articulations* on-line enables expressing motion constraints and parametrizing tasks in the same language as the joints and links of the robot itself. This idea led to the following thesis statement:

By homogeneously combining a robot with its task, virtual articulations enable models that are both qualitatively high-level and quantitatively functional.

The algorithms and data structures presented in Chapters 3 and 4 helped support this statement by forming a general purpose framework for modeling and real-time kinostatic simulation of *mixed real/virtual* articulated systems. While some earlier works have suggested virtual articulations in specific cases, this framework is the first to specifically address the unique challenges of general real/virtual modeling and simulation: it is unusually generic, supports a complete set of on-line topological mutations, incorporates a novel hierarchical subdivision, and scales to topologically large models. Allowing both real and virtual articulations also enables *kinematic abstractions* in the spatial and temporal domains. Abstractions are well known for managing complexity in algorithmic systems, but previously there was no corresponding formalism in kinematics.

The framework was fully implemented as the *mixed real/virtual interface* (Figure 1-4, and most other figures of 3D simulations in the thesis), and used as the basis for supporting experiments with four different robots in two application domains of current interest: operating high-DoF robots, and analyzing reliable compliant/proprioceptive locomotion. In both areas virtual articulations homogeneously model the robot and its task environment, and abstractions structure complex models. For high-DoF operations, the operator attaches virtual joints as a novel interface metaphor to define task motion and to constrain coordinated motion. For compliant locomotion, virtual articulations model relevant compliances and uncertainties, and temporal abstractions model contact.

This next sections summarize the advances and limitations of the framework and the approaches to high-DoF operations and compliant motion modeling it enables. The chapter concludes with an overview of future work.

7.1 Advances

The re-usable framework for mixed real/virtual models in Chapters 3 and 4 is unusually generic in two ways: it includes a large catalog of joints, with all lower pairs except helical; and it supports arbitrary topology models including both closed- and open-chain kinematics and both under- and over-constraint. The joint catalog is enabled by a proven-complete partition of a novel parametrization of the 3D special Euclidean group (Sections 3.3.2, 3.3.4, and Theorem 1). Topological independence is enabled by a reduced-coordinate modeling approach based on a kinematic graph with an identified spanning tree, root joints, and closure joints (Section 3.2); and also by a prioritized damped least-squares approach to motion computation (Chapter 4) which allows both over- and under-constraint. Such genericity is particularly relevant because part of the usefulness of virtual articulations is that they are not subject to the practical engineering constraints of actual physical mechanisms.

On-line topological mutation is not commonly supported in kinematic frameworks, but is important here for the addition and removal of virtual articulations. Section 3.6

and Appendix D provide a comprehensive set of algorithms for topology changes.

Scaling to topologically large structures, with 100s of joints and many closed chains, is also important: a main use of virtual articulations is to constrain motion in such high-DoF cases, and virtual joints can themselves quickly add up and increase overall model size. Two key decomposition algorithms work together to help enable this scaling: the coupling decomposition (Section 4.10) and a novel *hierarchical decomposition* (Section 4.10.1). These can help support interactive kinostatic simulation of local assembly (Section 4.1) and differential control (Section 4.2) motions by breaking break the model up into independently solvable sub-components.

The novel interface for operating high-DoF robots in Chapter 5 used the implementation of this framework as a graphical tool to operate robots with large numbers (10s to 100s) of joints, filling a gap between prior low and high-level interfaces. Virtual articulations parametrized task motion and also constrained and coordinated whole-robot motion, and *structure abstractions* hid internal operation of sub-mechanisms. Four new classes of coordinated motion were experimentally demonstrated for NASA/JPL's 36-DoF ATHLETE, all of which would have been difficult using prior methods. Constraint prioritization also enabled combining a novel direct manipulation device with virtual motion constraints (Figure 5-5). A second example, a simulation of a modular tower involving nearly 300 joints (Section 5.4), showed the scaling potential for the system. Structure abstraction and hierarchical decomposition helped manage complexity both for the operator and for the system.

The novel approach to concisely model compliant interaction in Chapter 6 used virtual articulations to homogeneously model compliance, contact, and uncertainty. *Sequence abstractions* hid details of contact state evolution. It has long been known that compliant mechanisms can perform reliably even under significant uncertainty, but little work has been done to enable modeling and understanding this relationship at a higher level than basic physics. Virtual articulations and kinematic abstractions are qualitatively useful here—they can help see the forest for the trees.

When sufficiently calibrated, kinostatic models can also enable quantitative prediction of tolerable uncertainty, using the procedure in Section 6.4.

Two hardware experiments are studied: *Shady*, a novel scratch-built compliant/proprioceptive vertical climbing robot, and *Steppy*, a mini-humanoid that climbs a step of uncertain height. Both systems have experimentally demonstrated reliability: Shady was 99.8% reliable in over 1000 climbing motions, and Steppy was about 90% reliable in 80 stair-stepping trials. In both cases mixed real/virtual models predict the limits of tolerable uncertainty and capture the essence of the coupling between compliance and uncertainty tolerance.

7.2 Limitations

Sections 5.2 and 6.5 called out the limitations of the framework as it applied to each application domain. Overall:

- The constructable constraints depend on available virtual joints; for example, the current joints cannot implement helical motion, rolling contact, or arbitrary trajectory path following.
- The decomposition algorithms may not reduce asymptotic time complexity, but by breaking the problem up they can lower the constant factors. Also, they can only decompose certain topological structures.
- Like abstractions in software, kinematic abstractions are manually designed, and depend on the ingenuity of the user. In general, it can be hard to prove that a structure abstraction is proper (Theorem 2), though the ability to handle over-constrained systems can make even improper abstractions usable.
- The framework includes link mass and stiffness for individual joint DoF. Non-statically stable robots will also require dynamics.
- I take a mainly local approach (Section 3.4.3), and do not consider collision detection or avoidance.

- I do not address specific control of velocity.

7.3 Future Work

Human Subject Testing and Usability Features: A clear next step for the high-DoF operations work is to perform usability experiments measuring both the operator learning time for the mixed real/virtual interface vs. existing systems and also the time required to design a complex motion in each. For these experiments to be meaningful, a few additional critical usability features should first be implemented, including snap-dragging [14], undo, and a more self-documenting drag-and-drop UI for constructing virtual articulations and kinematic abstractions.

Automatically Generated Virtual Articulations and Kinematic Abstractions: It would clearly be interesting to explore algorithms to automate the process of adding virtual articulations and kinematic abstractions. For the operations interface domain this could, for example, provide a bridge between a specification of geometric constraints (parallelism, perpendicularity, tangency, incidence, etc) and a set of virtual articulations that implement them. For the compliant motion domain, one idea would be to automatically infer sequence abstractions from the geometric models of the contacting bodies and a coarse description of (or possibly experimental data from) their relative motion in the task.

Prototype Model Libraries: Larger models often involve repeated sub-structures—consider ATHLETE's six identical legs, or the symmetries in the modular tower example. It could be useful to enable saving such sub-structures independently in a library, from which they could be repeatedly instantiated to build new models.

Collision Detection/Avoidance: It would often be desirable to compute motions that detect or avoid collision among the geometries associated with links (including terrain surface models). One form of collision avoidance can be implemented by simulating repulsive force fields, as described by Connelly, Demaine, et. al. in [21].

The differentiability of this approach would make it a natural addition to the local linear framework presented in this thesis (e.g., it collision avoidance could be inserted as another priority level), though other approaches could be considered as well.

Optimality Criteria: It would also be possible to insert other differentiable optimality criteria, such as manipulability maximization or joint limit avoidance, as new priority levels. Pryor [131] gives a thorough development of a number of such criteria.

Solver Improvements: Though the prioritized damped least squares solver (Chapter 4 and Appendix F) is already specifically designed to support a broad class of kinematic topologies, it still involves some tuning parameters. Automatic heuristics for setting these have been explored in the literature, and I implement some of these (Section 4.8.1), but there is still room for improvement.

Force Control: Several of the experiments with ATHLETE, for example trenching and bi-manual manipulation (Figures 5-6 and 5-8), suggested integrating some force control with the motion trajectories computed in the mixed real/virtual interface.

Full Dynamics: It should be possible to extend the present kinostatic framework into a full kinodynamic one. There is already some precedent for mixed real/virtual dynamic models [128, 127] in the literature, and also for on-line structure changes in dynamic articulated systems [108, 91].

7.4 Summary

This chapter began with a review of the thesis statement, and summarized how this statement has been supported by the presented algorithms, data structures, and experiments. The advances and the limitations of the mixed real/virtual framework, and its applications to high-DoF operations and compliant motion modeling, were then reviewed. Finally, the potential for future work was suggested in a number of different directions.

Appendix A

The Lower Pair Joints

In the late 19th century, Reuleaux proposed an important classification scheme for mechanical joints [133]. He noted that in practical machinery, where joint constraint must physically be implemented by contacting material components, it is often advantageous to maintain a two-dimensional surface of sliding contact. Reuleaux called such joints *lower pairs.* Remarkably there are only six classes of joint mobility that satisfy this criteria: revolute, prismatic, cylindrical, spherical, planar, and helical (figure A-1). Reuleaux himself apparently did not prove the completeness of this set, though for many years it was assumed complete—e.g. Denavit and Hartenberg state the completeness of the six lower pairs as fact in their seminal 1955 paper [38]. The first proof is usually attributed either to Waldron [160, 161] or to Hervé [63]. O'Connor and Srinivasan also include the completeness of the lower pairs as a corollary of their more general completeness proof for the classes of connected Lie subgroups of the rigid motion group [114].

Joints which maintain only curve or point—as opposed to full surface—contact were termed *higher pairs.* These include, for example, cam-followers. And there are certainly joints which are neither lower nor higher pairs: for example, a discrete joint which takes on only a finite set of poses, or a joint that maintains non-slip rolling contact or gear mesh.

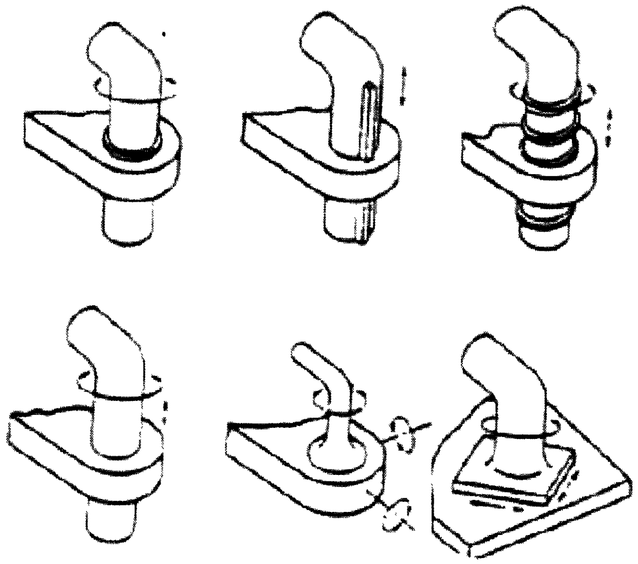


Figure A-1: The six classes of lower pair joint.
Top row, left to right: revolute, prismatic, helical. Bottom row: cylindrical, spherical, planar. Figure adapted from [139].

Appendix B

Orientation Vectors

It is well known that there are various ways to parametrize the space of 3D rotations—rotation matrices, unit quaternions, Euler parameters, axis-angle, etc. (see e.g. [98]). For reasons discussed in Section 3.3.3 I use the *orientation vector* approach [58] in this thesis, combined with a differentiable *exponential map* from an orientation vector $\boldsymbol{\theta}$ to a unit quaternion

$$\mathring{\boldsymbol{q}} = \exp(\boldsymbol{\theta}), \tag{B.1}$$

the inverse *logarithmic map*

$$\boldsymbol{\theta} = \log(\mathring{\boldsymbol{q}}), \tag{B.2}$$

and a method of *dynamic reparametrization* recently introduced by Grassia [55].

Orientation vectors are a particular instantiation of the axis-angle approach to orientation parametrization, with the amount of rotation encoded as the length and the rotation axis the direction of a 3D vector. The full details are given in Definition 65.

Definition 65 The direction $\hat{\boldsymbol{\theta}} = \boldsymbol{\theta}/\|\boldsymbol{\theta}\|$ of a non-degenerate *orientation vector* $\boldsymbol{\theta} \in \mathbb{R}^3 \setminus \mathbf{0}$ defines an axis of rotation, and the length $\|\boldsymbol{\theta}\|$ defines an amount of rotation, or *twist*, according to the right hand rule, in radians, as illustrated in Figure 3-5. The degenerate orientation vector $\boldsymbol{\theta} = \mathbf{0}$ corresponds to the identity (i.e. no) rotation. In either case, the *effective twist* θ of $\boldsymbol{\theta}$ is

$$\theta = \|\boldsymbol{\theta}\| \bmod 2\pi. \tag{B.3}$$

Euler's rotation theorem [98] establishes that such an axis and twist are sufficient to represent any 3D rotation.

The orientation vector representation is a multiple-cover of the space of 3D rotations, repeating on successive 2π-wide half-closed spherical annuli. Such an annulus $U_\imath$, $i \in \mathbb{Z}^+$, is precisely defined as

$$U_\imath = B^3(2i\pi) \setminus B^3(2(i-1)\pi), \tag{B.4}$$

with $B^d(r)$ the open Euclidean ball of radius r centered at the origin in $\mathbb{R}^d$.

Orientation vectors on the (included) inner boundary of any annulus $U_\imath$ all have effective twist

$$\theta = 2(i-1)\pi \bmod 2\pi = 0 \tag{B.5}$$

and thus all correspond to the identity rotation. Except for the innermost annulus U_1 there is generally a continuum of such inner-boundary vectors, a fact which will be important below. The remaining vectors in $U_\imath$ cover the space of all rotations because (1) all axis directions are reachable and (2) all effective twists $0 \leq \theta < 2\pi$ are included.

In fact, each $U_\imath$ is itself a double-cover of the space of all 3D rotations. Consider the *inner closed half-annulus*

$$\bar{U}_\imath^\nabla = \bar{B}^3((2i-1)\pi) \setminus B^3(2(i-1)\pi), \tag{B.6}$$

with $\bar{B}^d(r)$ the closure of $B^d(r)$.

$\bar{U}_\imath^\nabla$ is itself sufficient to cover the space of all 3D rotations. It includes all effective twists in the range $0 \leq \theta \leq \pi$, and the remaining rotation vectors in the *outer open half-annulus*

$$U_\imath^\Delta = B^3(2i\pi) \setminus \bar{B}^3((2i-1)\pi) \tag{B.7}$$

with effective twists $\pi < \theta < 2\pi$ are equivalent to rotations about the opposing axis with effective twist $\theta' = 2\pi - \theta$. This fact will also be important below.

B.1 Computations via Unit Quaternions

An orientation vector $\boldsymbol{\theta}$ can be considered from both denotational and operational viewpoints. Denotationally, $\boldsymbol{\theta}$ is current orientation of a 3D object in some ambient coordinate frame; operationally $\boldsymbol{\theta}$ is a rotation *action* taking any 3D object from its current orientation to a new one. The operational viewpoint is crucial, as there will often be a need to compute (1) inverse rotations, (2) rotation compositions, and (3) the application of a rotation to a set of points defining an object.

The computation to invert an orientation vector is trivial:

$$\boldsymbol{\theta}^{-1} = -\boldsymbol{\theta}. \tag{B.8}$$

However, there seems to be no more direct way to compute rotation composition and point rotation than to first convert to another representation [55]. For this purpose I use unit quaternions, with an exponential map (Eq. B.1) taking $\boldsymbol{\theta}$ to an equivalent unit quaternion $\mathring{\boldsymbol{q}}$, and a logarithmic map (Eq. B.2) converting back. The essential formulas for unit quaternion composition operations are reviewed in Appendix C.

B.2 The Exponential and Logarithmic Maps

The terms "exponential" and "logarithmic" derive from the fact that the space of unit quaternions is a Lie group for which the orientation vectors can be considered a Lie algebra. In general, the mapping from any Lie algebra to its Lie group is called an exponential map, and the opposite mapping is a logarithmic map.

Definition 66 The particular *exponential map* from orientation vectors to unit quaternions that I use is computed as [55]

$$
\begin{aligned}
\exp : \mathbb{R}^3 \rightarrow S^3 \subset \mathbb{R}^4 \quad \boldsymbol{\theta} \rightarrow \exp(\boldsymbol{\theta}) &= (q_w, \boldsymbol{q}) = \mathring{\boldsymbol{q}} \\
&= \left(\cos \frac{\|\boldsymbol{\theta}\|}{2}, \frac{\boldsymbol{\theta}}{\|\boldsymbol{\theta}\|} \sin \frac{\|\boldsymbol{\theta}\|}{2} \right)
\end{aligned}
\tag{B.9}
$$

with q_w the scalar and $\boldsymbol{q} = (q_x, q_y, q_q)$ the vector part of $\mathring{\boldsymbol{q}}$

and S^d the unit sphere centered at the origin in $\mathbb{R}^d$.

I compute the *logarithmic map* from unit quaternions to orientation vectors as

$$
\begin{aligned}
\log : S^3 \subset \mathbb{R}^4 \rightarrow \mathbb{R}^3 \quad \mathring{\boldsymbol{q}} \rightarrow \log(\mathring{\boldsymbol{q}}) &= \log((q_w, \boldsymbol{q})) = \boldsymbol{\theta} \\
&= \begin{cases} 2\,\mathrm{atan2}(\|\boldsymbol{q}\|, q_w) \frac{\boldsymbol{q}}{\|\boldsymbol{q}\|} & \text{if } \|\boldsymbol{q}\| > 0 \\ \mathbf{0} & \text{otherwise.} \end{cases}
\end{aligned}
\tag{B.10}
$$

Figure B-1 illustrates these mappings graphically.

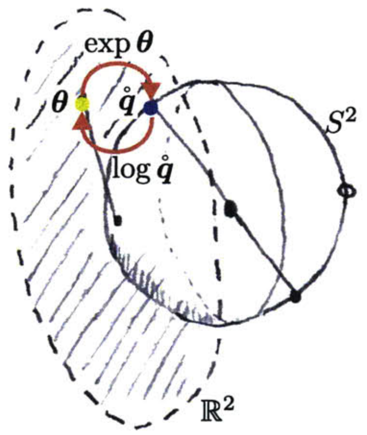

exp $\boldsymbol{\theta}$
$\boldsymbol{\theta}$
$\mathring{\boldsymbol{q}}$
S^2
log $\mathring{\boldsymbol{q}}$
$\mathbb{R}^2$

Figure B-1: $\exp(\boldsymbol{\theta})$ and $\log(\mathring{\boldsymbol{q}})$.
The exponential map takes orientation vectors to unit quaternions. The actual mapping is from $\mathbb{R}^3$ to the 4D sphere S^3, which is difficult to draw, but motivated by analogy using $\mathbb{R}^2$ and the 3D sphere S^2.

B.3 Singularities

Despite the appearance of a singularity at $\|\boldsymbol{\theta}\| = 0$ in Eq. B.9, in fact

$$\lim_{\|\boldsymbol{\theta}\| \to 0} \frac{\sin \|\boldsymbol{\theta}\|/2}{\|\boldsymbol{\theta}\|} = 1/2. \tag{B.11}$$

Similarly, Eq. B.10 is differentiable at $\|\boldsymbol{q}\| = 0$ (i.e. when $\boldsymbol{\theta} = \mathbf{0}$) because the unit-magnitude constraint on $\mathring{\boldsymbol{q}}$ implies that $q_w = \sqrt{1 - \|\boldsymbol{q}\|^2}$ and

$$\lim_{\|\boldsymbol{q}\| \to 0} \operatorname{atan2}(\|\boldsymbol{q}\|, \sqrt{1 - \|\boldsymbol{q}\|^2}) = 0. \tag{B.12}$$

Thus, both the exponential and the logarithmic maps are differentiable in the neighborhood of $\boldsymbol{\theta} = \mathbf{0}$, which is the degenerate inner boundary of the spherical annulus U_1. Numerically stable computations for the derivatives will be given in the next section, even for $\boldsymbol{\theta} \to \mathbf{0}$.

The singularities at the inner boundary of the other annuli are not so easily avoided—the continua of orientation vectors on these boundaries all map to the single quaternion $(-1, 0, 0, 0)$. Fortunately, it suffices for our purposes to stay within U_1, which we can do with a technique of dynamic reparametrization, as described below in Section B.5.

B.4 Derivatives

The algorithm presented in Chapter 4 to compute linkage motion relies on the computation of the derivatives of $\exp(\boldsymbol{\theta})$ and $\log(\mathring{\boldsymbol{q}})$. These take the form of $[4 \times 3]$ and $[3 \times 4]$ matrices, respectively:

$$\text{with } \boldsymbol{\theta} = (\theta_x, \theta_y, \theta_z) \text{ and } \exp \boldsymbol{\theta} = \mathring{\boldsymbol{q}} = (q_w, q_x, q_y, q_z)$$

$$\frac{\partial \mathring{\boldsymbol{q}}}{\partial \boldsymbol{\theta}} = \frac{\partial \exp \boldsymbol{\theta}}{\partial \boldsymbol{\theta}} = \underbrace{\begin{bmatrix} \frac{dq_w}{d\theta_x} & \cdots & \frac{dq_w}{d\theta_z} \\ \vdots & \ddots & \vdots \\ \frac{dq_z}{d\theta_x} & \cdots & \frac{dq_z}{d\theta_z} \end{bmatrix}}_{[4\times3]}, \tag{B.13}$$

$$\text{with } \mathring{\boldsymbol{q}} = (q_w, q_x, q_y, q_z) \text{ and } \log \mathring{\boldsymbol{q}} = \boldsymbol{\theta} = (\theta_x, \theta_y, \theta_z)$$

$$\frac{\partial \boldsymbol{\theta}}{\partial \mathring{\boldsymbol{q}}} = \frac{\partial \log \mathring{\boldsymbol{q}}}{\partial \mathring{\boldsymbol{q}}} = \underset{[3\times4]}{\begin{bmatrix} \frac{d\theta_x}{dq_w} & \cdots & \frac{d\theta_x}{dq_z} \\ \vdots & \ddots & \vdots \\ \frac{d\theta_z}{dq_w} & \cdots & \frac{d\theta_z}{dq_z} \end{bmatrix}}. \tag{B.14}$$

In [55], Grassia showed that the expressions for the entries in Eq. B.13 reduce to only three forms:

$$\text{with } \theta = \|\boldsymbol{\theta}\|,\ s_2 = \sin(\theta/2),\ c_2 = \cos(\theta/2) \text{ and } \alpha, \beta \in \{x, y, z\}$$

$$\begin{aligned} \frac{dq_\alpha}{d\theta_\alpha} &= \frac{s_2}{\theta} + \frac{\theta_\alpha^2}{\theta^2}\left(\frac{c_2}{2} - \frac{s_2}{\theta}\right) \\ \frac{dq_\alpha}{d\theta_\beta} &= \frac{\theta_\alpha \theta_\beta}{\theta^2}\left(\frac{c_2}{2} - \frac{s_2}{\theta}\right) \quad \alpha \neq \beta \\ \frac{dq_w}{d\theta_\alpha} &= -\frac{\theta_\alpha}{2}\frac{s_2}{\theta}. \end{aligned} \tag{B.15}$$

Grassia pointed out that a reasonable approach in the region $\theta \longrightarrow 0$ is to switch to approximations for the rational forms s_2/θ and $1/\theta^2\,(c_2/2 - s_2/\theta)$:

$$\begin{aligned} &\text{when } \theta \leq \sqrt[4]{\text{machine precision}} \\ &\frac{s_2}{\theta} \approx \frac{1}{2} - \frac{\theta^2}{48} \\ \frac{1}{\theta^2}&\left(\frac{c_2}{2} - \frac{s_2}{\theta}\right) \approx \frac{1}{24}\left(\frac{\theta^2}{40} - 1\right). \end{aligned} \tag{B.16}$$

This technique aids stability of the numerical computations when implemented in limited precision (including standard floating-point) by avoiding division by small numbers.

Grassia did not also present expressions for the entries in the derivative of the logarithmic map (Eq. B.14); I derive them as follows:

with $s_2 = \|\boldsymbol{q}\|,\ c_2 = q_w,\ \theta = 2\,\mathrm{atan2}(s_2, c_2),$ and $\alpha, \beta \in \{x, y, z\}$

$$\begin{aligned} \frac{d\theta_\alpha}{dq_\alpha} &= \frac{\theta}{s_2} + \frac{q_\alpha^2}{s_2^2}\left(2c_2 - \frac{\theta}{s_2}\right) \\ \frac{d\theta_\alpha}{dq_\beta} &= \frac{q_\alpha q_\beta}{s_2^2}\left(2c_2 - \frac{\theta}{s_2}\right) \quad \alpha \neq \beta \\ \frac{d\theta_\alpha}{dq_w} &= -2q_\alpha. \end{aligned} \tag{B.17}$$

In this case, for $\theta \to 0$ replace the rational forms θ/s_2 and $1/s_2^2(2c_2 - \theta/s_2)$ with corresponding approximations:

$$\begin{aligned} &\text{when } \theta \leq \sqrt[4]{\text{machine precision}} \\ \frac{\theta}{s_2} &\approx \frac{48}{24 - \theta^2} \\ \frac{1}{s_2^2}\left(2c_2 - \frac{\theta}{s_2}\right) &\approx \frac{32}{\theta^2 - 24}. \end{aligned} \tag{B.18}$$

B.5 Dynamic Reparametrization

One of Grassia's main contributions in [55] is that, for incremental computations, a restriction of $\boldsymbol{\theta}$ to the domain

$$\boldsymbol{\theta} \in \bar{U}_1^\nabla = \bar{B}^3(\pi) \tag{B.19}$$

gives sufficient coverage of the space of 3D rotations while avoiding all the problematic singularity surfaces. The iterative Solve algorithm in Chapter 4 is precisely such an incremental computation context—after each iteration all orientation vectors are *dynamically reparametrized*, returning any $\boldsymbol{\theta}$ that the prior incremental update pushed outside $\bar{B}^3(\pi)$ to an equivalent $\boldsymbol{\theta}' \in \bar{B}^3(\pi)$ as follows:

$$\text{assuming } \|\boldsymbol{\theta}\| < 2\pi, \text{ if not then first set } \boldsymbol{\theta} = (\|\boldsymbol{\theta}\| \bmod 2\pi)\frac{\boldsymbol{\theta}}{\|\boldsymbol{\theta}\|}$$

$$\boldsymbol{\theta}' = \left(1 - \frac{2\pi}{\|\boldsymbol{\theta}\|}\right)\boldsymbol{\theta} \text{ when } \pi < \|\boldsymbol{\theta}\| < 2\pi. \tag{B.20}$$

Thus, the incremental SOLVE algorithm can evolve the rotation state of any joint without worry of "falling off the edge" of the parametrization domain given by Eq. B.19.

B.6 Taking Differences of Orientation Vectors

The term "orientation vector" is partly a misnomer because the space of 3D rotations is not a Euclidean vector space. Normal vector-algebraic manipulations, such as addition, do not necessarily correspond to meaningful operations on orientation vectors.

That said, there are two instances in which algebraic manipulations on orientation vectors are useful. One is finding the inverse of an orientation vector by negating it (Eq. B.8). Second, I represent the *difference* $\boldsymbol{e}$ between two orientations algebraically, though this difference is not itself an orientation vector:

$$\boldsymbol{e} = \boldsymbol{\theta}_1 - \boldsymbol{\theta}_0. \tag{B.21}$$

When taking such a difference, care must be taken to handle the fact that, due to the multiple-cover of rotation space, there is a countable infinity of orientation vectors which represent the same rotation. Typically it makes sense to select nearest aliases of $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_0$ which minimize $\boldsymbol{e}$. I take this approach when computing residuals for the *Solve* algorithm, Section 4.9.1 gives the details.

Appendix C

Unit Quaternions

Unit quaternions are a powerful and convenient way to represent 3D rotations. They gained popularity in graphics and robotics the 1980s when Shoemake [141] introduced a practical quaternion scheme for correctly interpolating 3D rotations. Horn gives a concise summary of quaternion computations in [71].

Here I review the quaternion manipulations and formulae that are used in this thesis. Unit quaternions used alone have some drawbacks for our purposes, as discussed in Section 3.3.3. I use them in conjunction with the orientation vector rotation representation (Appendix B).

The 3D rotation represented by a unit quaternion $\mathring{\boldsymbol{q}} = (q_w, \boldsymbol{q})$ has a geometric interpretation that is similar to an orientation vector: $\boldsymbol{q}$ is parallel to the axis of rotation and has magnitude $\sin(\theta/2)$, where θ is the right-hand-rule rotation about the axis in radians, and q_w is $\cos(\theta/2)$.

The inverse of the rotation represented by a unit quaternion $\mathring{\boldsymbol{q}}$ is given by the *conjugate* $\mathring{\boldsymbol{q}}^*$:

$$\text{when } \|\mathring{\boldsymbol{q}}\| = 1$$
$$\mathring{\boldsymbol{q}}^{-1} = \mathring{\boldsymbol{q}}^* = (q_w, \boldsymbol{q})^* = (q_w, -\boldsymbol{q}). \tag{C.1}$$

Composition of rotations and point rotation are both computed using quaternion multiplication:

$$\mathring{\boldsymbol{r}} = \mathring{\boldsymbol{p}}\mathring{\boldsymbol{q}}. \tag{C.2}$$

I.e. $\mathring{\boldsymbol{r}}$ identifies the rotation that results by first applying the rotation $\mathring{\boldsymbol{q}}$ followed by the rotation $\mathring{\boldsymbol{p}}$. For the application of a rotation $\mathring{\boldsymbol{r}}$ to a point $\boldsymbol{v} = (v_x, v_y, v_z) \in \mathbb{R}^3$

$$\boldsymbol{v}' = \mathrm{vp}(\mathring{\boldsymbol{r}}\mathring{\boldsymbol{v}}\mathring{\boldsymbol{r}}^*) \tag{C.3}$$

$$\text{with } \mathring{\boldsymbol{r}} = (0, \boldsymbol{r}) \text{ and } \mathrm{vp}((r_w, \boldsymbol{r})) = \boldsymbol{r}, \tag{C.4}$$

where $\boldsymbol{v}'$ is the result of rotating $\boldsymbol{v}$ by $\mathring{\boldsymbol{r}}$.

These expressions employ quaternion multiplication but do not define how to compute it. Horn gives a practical method based on simple functions—which I call QMM and QMM*—that produce $[4 \times 4]$ matrices from a quaternion, so that quaternion multiplication can be cast into the multiplication of a matrix and a vector. There are two variations depending on whether the first or the second factor is converted into a matrix:

$$\begin{aligned} \mathring{\boldsymbol{r}} &= \mathring{\boldsymbol{p}}\mathring{\boldsymbol{q}} & \mathring{\boldsymbol{r}} &= \mathring{\boldsymbol{p}}\mathring{\boldsymbol{q}} \\ &= \mathrm{QMM}(\mathring{\boldsymbol{p}})(q_w, q_x, q_y, q_z)^{\mathrm{T}} & &= \mathrm{QMM}^*(\mathring{\boldsymbol{q}})(p_w, p_x, p_y, p_z)^{\mathrm{T}} \\ \mathrm{QMM}(\mathring{\boldsymbol{p}}) &= \begin{bmatrix} p_w & -p_x & -p_y & -p_z \\ p_x & p_w & -p_z & p_y \\ p_y & p_z & p_w & -p_x \\ p_z & -p_y & p_x & p_w \end{bmatrix} & \mathrm{QMM}^*(\mathring{\boldsymbol{q}}) &= \begin{bmatrix} q_w & -q_x & -q_y & -q_z \\ q_x & q_w & q_z & -q_y \\ q_y & -q_z & q_w & q_x \\ q_z & -q_y & -q_x & q_w \end{bmatrix}. \end{aligned} \tag{C.5}$$

Eq. H.9 uses QMM and QMM*.

Finally, there is a well-known (e.g. see [2]) function, which I call ROT, that produces a $[3 \times 3]$ orthogonal rotation matrix corresponding to a quaternion:

$$\mathrm{ROT}(\mathring{\boldsymbol{q}}) = \mathrm{ROT}((q_w, q_x, q_y, q_z)) = \begin{bmatrix} 1-2q_z^2-2q_y^2 & -2q_wq_q+2q_yq_x & 2q_wq_y+2q_zq_x \\ 2q_wq_q & 1-2q_z^2-2q_x^2 & -2q_wq_x+2q_zq_y \\ -2q_wq_y+2q_xq_z & 2q_wq_x+wq_yq_z & 1-2q_y^2-2q_x^2 \end{bmatrix} \tag{C.6}$$

Eq. H.9 also uses ROT and its partial derivatives $\partial\mathrm{ROT}(\mathring{\boldsymbol{q}})/\partial\mathring{\boldsymbol{q}}$. These can be represented as a $[[3 \times 3] \times 4]$ tensor, i.e. as a stack of four $[3 \times 3]$ matrices, each of which is the partial derivative of ROT with respect to one of the four components of its quaternion argument:

$$\underset{[[3\times3]\times4]}{\frac{\partial \text{ROT}(\mathring{\boldsymbol{q}})}{\partial \mathring{\boldsymbol{q}}}} = \begin{bmatrix} \underset{[3\times3]}{\frac{\partial \text{ROT}(\mathring{\boldsymbol{q}})}{\partial q_w}} & \underset{[3\times3]}{\frac{\partial \text{ROT}(\mathring{\boldsymbol{q}})}{\partial q_x}} & \underset{[3\times3]}{\frac{\partial \text{ROT}(\mathring{\boldsymbol{q}})}{\partial q_y}} & \underset{[3\times3]}{\frac{\partial \text{ROT}(\mathring{\boldsymbol{q}})}{\partial q_z}} \end{bmatrix} \tag{C.7}$$

with
$$\frac{\partial \text{ROT}(\mathring{\boldsymbol{q}})}{\partial q_w} = \begin{bmatrix} 0 & -2q_z & 2q_y \\ 2q_z & 0 & -2q_x \\ -2q_y & 2q_x & 0 \end{bmatrix} \quad \frac{\partial \text{ROT}(\mathring{\boldsymbol{q}})}{\partial q_x} = \begin{bmatrix} 0 & 2q_y & 2q_z \\ 2q_y & -4q_x & -2q_w \\ 2q_z & 2q_w & -4q_x \end{bmatrix}$$
$$\frac{\partial \text{ROT}(\mathring{\boldsymbol{q}})}{\partial q_y} = \begin{bmatrix} -4q_y & 2q_x & 2q_w \\ 2q_x & 0 & 2q_z \\ -2q_w & 2q_z & -4q_y \end{bmatrix} \quad \frac{\partial \text{ROT}(\mathring{\boldsymbol{q}})}{\partial q_z} = \begin{bmatrix} -4q_z & -2q_w & 2q_x \\ 2q_w & -4q_z & 2q_y \\ 2q_x & 2q_y & 0 \end{bmatrix}.$$

The derivatives of the inverse, $\partial(\text{ROT}(\mathring{\boldsymbol{q}}))^{-1}/\partial\mathring{\boldsymbol{q}}$, are calculated similarly except each plane in Eq. C.7 is individually transposed.

Appendix D

Additional Structure Mutation Primitives

This appendix collects all the remaining structure mutation primitives and helper functions that were omitted from Chapter 3.

D.1 Adding Links and Joints

AddLink and SetLinkName, Algorithms D.1 and D.2, add a new link to the linkage or change the name of an existing link, respectively. AddLink always produces a new uniquely named root link co-located with the ground frame—the connectivity, pose, and name of the new link may all be evolved by the application of further mutation primitives. A unique associated root joint is also automatically created for each new link. A name n is valid iff it is a string of alphanumeric characters, dash, and underscore. Both AddLink and SetLinkName have $O(1)$ time complexity (see assumptions in Section 3.6.1).

AddJoint and SetJointName, Algorithms D.3 and D.4, are the parallel operations for adding and re-naming a joint. Newly added joints are always initially chain closures; they may be promoted to tree disposition by MakeTree, Algorithm 3.1 on page 87. Assuming that CMTs have been pre-computed, AddJoint and SetJoint-Name are also $O(1)$. If up-to-date CMTs are not available, an extra cost of $O(h)$ is incurred in AddJoint, where h is the maximum spanning tree height of the parent and child links of the new joint.

D.2 Removing Links and Joints

Removing a closure joint is straightforward, as closure joints play no critical function in the linkage topology. The general strategy for RemoveJoint, Algorithm D.5, is thus to first ensure the joint is a closure, returning the child link of a tree joint to its root, and then to unlink the closure. Its run time is dominated by the call to MakeClosure.

Removing a link is slightly more involved, as all the adjacent joints will be left dangling. In this implementation of RemoveLink, Algorithm D.6, these joints are also removed. In the process, any tree descendants of the link will be made root links. RemoveLink is $O(ah)$, where a is the number of joints adjacent to the link and h is either 1 if updated CMTs are available or the maximum spanning tree height of any adjacent link.

D.3 Changing Joint Type and Limits

SetType and SetLimits, Algorithms D.7 and D.8, mutate the type and limits of a joint in-place. For tree joints, the mobility transform is also clamped to the new mobility space. One particular use for these operations is for joint space restriction and generalization (Section 3.4.1). Joint limits I are valid iff they satisfy Eqs. 3.33 and 3.34. $I = \emptyset$ is a shorthand for all-infinite limits. Both SetType and SetLimits are $O(1)$.

D.4 Repositioning Links, Restructuring Joints

The joint mobility positioning transforms P and C, taking the child link frame to the child mobility frame and the parent mobility frame to the parent link frame, respec-

```
Algorithm D.1: AddLink($L$)
  Input: linkage $L$
  Output: a new link $k$, which has been added to $L$ along with its root joint
  let $n$ be a unique link name
  let $O = (M_r, Y_r, I_r, \phi_r, P_r, C_r) \leftarrow (\mathbf{0}, \mathbf{G}, \emptyset, +1, \mathbf{0}, \mathbf{0})$ ▷Eq. 3.11
  let $r = (p_r, c_r, O_r, \Upsilon_r, n_r) \leftarrow (g, \emptyset, O, \emptyset, \emptyset)$ ▷Eq. 3.3
  let $k = (p_k, r_k, n_k) \leftarrow (r, r, n)$ ▷Eq. 3.4
  $c_r \leftarrow k$, $n_r \leftarrow$ RootName($k$)
  add $k$ to $K_L$ and $r$ to $J_L$
  return $k$
```

```
Algorithm D.2: SetLinkName($k, n$)
  Input: link $k$, valid link name $n$
  Output: $k$ renamed to $n$
  $n_k \leftarrow n$, $n_{r_k} \leftarrow$ RootName($k$)
  return $k$
```

```
Algorithm D.3: AddJoint($L, Y, p, c$)
  Input: linkage $L$, joint type $Y \in \mathcal{Y}$, parent and child links $p, c \in K_L$
  Output: a new closure joint $j$, which has been added to $L$
  let $n$ be a unique joint name
  let $C \leftarrow \text{cmt}(p)^{-1}\text{cmt}(c)$ ▷child-to-mobility transform
  let $O = (M_j, Y_j, I_j, \phi_j, P_j, C_j) \leftarrow (\emptyset, Y, \emptyset, +1, \mathbf{0}, C)$ ▷Eq. 3.11
  let $j = (p_j, c_j, O_j, \Upsilon_j, n_j) \leftarrow (p, c, O, \emptyset, n)$ ▷Eq. 3.3
  add $j$ to $J$
  return $j$
```

```
Algorithm D.4: SetJointName($j, n$)
  Input: non-root joint $j$, valid joint name $n$
  Output: $j$ renamed to $n$
  if $n$ not unique in $p_j$ then error name not unique
  $n_j \leftarrow n$
  return $j$
```

```
Algorithm D.5: RemoveJoint($j$)
  Input: non-root joint $j$ in a linkage $L$ (i.e. $j \in J_L$)
  Output: $L$ with $j$ removed
  MakeClosure($j$)
  remove $j$ from $J_L$
  return $L$
```

```
Algorithm D.6: RemoveLink($k$)
  Input: non-ground link $k$ in a linkage $L$ (i.e. $k \in K_l$)
  Output: $L$ with $k$ removed
  foreach joint $j$ s.t. $p_j = k$ do RemoveJoint($j$)
  foreach joint $j$ s.t. $c_j = k$ and closure?($j$) do
    if ¬RootJoint?($j$) then RemoveJoint($j$)
  $K_L \leftarrow K_L \setminus \{k\}$, $J_L \leftarrow J_L \setminus (\{r_k\} \cup \{p_k\})$
  return $L$
```

tively, provide an important level of flexibility: they can be manipulated to give any desired relative positioning of a joint's mobility space with respect to each adjacent link frame. I provide two mutation primitives, RepositionLink and Reposition-Joint, Algorithms D.9 and D.10, which serve to move a link or joint while holding the adjacent joints or links pinned in their current global poses. These repositioning operations do not actually change the mobility state of the joint. For example, RepositionJoint could be used to change the orientation of the rotation axis of a revolute joint with respect to the adjacent links, but would not actually change the joint's current amount of rotation. RepositionLink is $O(ch)$ where c is the number of child joints and h is 1 if updated CMTs are available, or the spanning tree height of the link otherwise. RepositionJoint is $O(h)$ where h is 1 if updated CMTs are available, else the spanning tree height of the parent link.

Sometimes it is necessary to independently move the parent and child links of

```
Algorithm D.7: SetType($j, Y$)
  Input: non-root joint $j$, joint type $Y \in \mathcal{Y}$
  Output: $j$, now of type $Y$
  $Y_j \leftarrow Y$
  if tree?($j$) then $M_j \leftarrow$ ClampX($M_j, Y_j, I_j$)
  return $j$
```

```
Algorithm D.8: SETLIMITS($j, I$)
    Input: non-root joint $j$, valid joint limits $I$
    Output: $j$, now with limits $I$
    $I_j \leftarrow I$
    if TREE?($j$) then $M_j \leftarrow$ CLAMPX($M_j, Y_j, I_j$)
    return $j$
```

```
Algorithm D.9: REPOSITIONLINK($k, X_{g_0 \leftarrow k}$)
    Input: non-ground link $k$, composite model transform $X_{g_0 \leftarrow k}$
    Output: $k$, now repositioned
    let $X \leftarrow \text{CMT}(k)^{-1} X_{g_0 \leftarrow k}$
    if ROOTJOINT?($p_k$) then $M_{p_k} \leftarrow M_{p_k} X$ else $C_{p_k} \leftarrow C_{p_k} X$
    foreach joint $j$ s.t. $p_j = k$ do $P_j \leftarrow X^{-1} P_j$
    return $k$
```

a joint relative to the mobility space, i.e. to individually control the positioning transforms. RESTRUCTURE, Algorithm D.11, performs this operation in $O(1)$ time.

D.5 Splitting Links and Merging Joints

The final two mutation primitives, SPLIT and MERGE, provide insertion and removal semantics for links and joints that are more convenient in some cases than ADDJOINT, ADDLINK, REMOVEJOINT, and REMOVELINK. They are strictly sugar—each could be implemented via calls to those more basic primitives (though the time complexity would increase due to the cyclicity error checking in SETPARENT and SETCHILD).

SPLIT, Algorithm D.12, inserts a combination of a new link and a new tree joint behind an existing link k.The new link is initially co-located with k. Except for the case of splitting the ground link, SPLIT is $O(1)$ expected if updated CMTs are available, else $O(h)$ expected with h the spanning tree height of the link to be split.

```
Algorithm D.10: REPOSITIONJOINT($j, X_{g_0 \leftarrow pm_j}$)
    Input: non-root joint $j$, composite model transform $X_{g_0 \leftarrow pm_j}$
    Output: $j$, now repositioned
    let $X \leftarrow \text{CMT}(pm_j)^{-1} X_{g_0 \leftarrow pm_j}$
    $P_j \leftarrow P_j X$, $C_j \leftarrow X^{-1} C_j$
    return $j$
```

Algorithm D.11: RESTRUCTURE(j, P, C)

```
Input: non-root joint $j \in J$, transforms $P$ and $C$
Output: $j$, now restructured
if $P \neq \emptyset$ then $P_j \leftarrow P$, if $C \neq \emptyset$ then $C_j \leftarrow C$
return $j$
```

Algorithm D.12: SPLIT(k)

```
Input: link $k$ in linkage $L$ (i.e. $k \in K_L$)
Output: a new link $k'$ inserted with a new tree joint behind $k$
let $k' \leftarrow$ ADDLINK$(L)$
if $k = g_L$ then MAKEGROUND$(k')$ else
    $M_{p'_k} \leftarrow$ CMT$(k)$
    if $\neg$ROOTLINK?$(k)$ then $c_{p_k} \leftarrow k'$, $p_{k'} \leftarrow p_k$
    let $j \leftarrow$ ADDJOINT$(\mathbf{G}, k', k)$
    $p_k \leftarrow j$, $M_j \leftarrow \mathbf{0}$
return $k'$
```

SPLITting the ground link has the same cost as the root link case of MAKEGROUND.

MERGE, Algorithm D.13, performs the reciprocal operation to split: it deletes a tree joint j and its child link c_j, re-attaching all other neighboring joints of c_j to the parent p_j of j. MERGE is $O(|N| + |O| + h)$ where N is the set of child joints of the

Algorithm D.13: MERGE(j)

```
Input: non-root joint $j$
Output: the previous parent link $p_j$ of $j$
let $D \leftarrow \{\text{joint } i \mid p_i = c_j\}$, $N \leftarrow \{\text{joint } i \mid c_i = c_j \text{ and CLOSURE?}(i)\}$
foreach $i$ in $D$ do if $n_i$ not unique in $p_j$ then error name not unique
let $X_j \leftarrow \text{CMT}(p_j)^{-1}\text{CMT}(c_j)$
foreach $i$ in $D$ do $P_i \leftarrow X_j P_i$, $p_i \leftarrow p_j$
foreach $i$ in $N$ if $\neg$ROOTJOINT?$(i)$ do $C_i \leftarrow C_i X_j^{-1}$, $c_i \leftarrow p_j$
REMOVELINK$(c_j)$, REMOVEJOINT$(j)$
return $p_j$
```

child link of the joint to be merged, O is the set of closure joints whose child is that link, and h is 1 if updated CMTs are available or the maximum spanning tree height of the parent and child links of the joint to be merged otherwise.

D.6 Helper Functions

Some helper functions make the definitions of the main algorithms more concise. With the exception of ClampV and ClampX, these do not perform any mutations.

The first two helpers, tree? and closure?, Algorithms D.14 and D.15, are conveniences to test the disposition of a joint. The time complexity of each is $O(1)$.

RootLink? and RootJoint?, Algorithms D.16 and D.17, are convenience predicates to test for a root link (a link parented directly to ground through its root joint), or whether a joint is a root joint. The time complexity of each is $O(1)$.

RootName, Algorithm D.18, generates a unique name for a root joint.

The clamping helper functions are the only ones which perform any mutation. ClampV, Algorithm D.22, clamps either a $\boldsymbol{t}$ or $\boldsymbol{\theta}$ 3-vector to a specified axis-aligned subspace and limits, and ClampX, Algorithm D.23, builds on this to clamp a full $(\boldsymbol{t}, \boldsymbol{\theta})$ transform, implementing Eq. 3.36. Both have time complexity $O(1)$. While not explicitly handled in the pseudocode, $I = \emptyset$ is a shorthand for all-infinite limits.

The final helper function, cmt, Algorithm D.24, computes the *composite model transform* taking coordinates in any model frame to the link frame of the ground link g_0 of the top-most sub-linkage, effectively implementing Eq. 3.1. It is expressed here as a self-contained recursive function that can compute the CMT for any coordinate frame, even those associated with closure joints, without directly requiring the mobility transform M_j or joint transform X_j for any closure joint (recall from Section 3.3.2 that these quantities are implicit).

cmt has time complexity $O(h)$ where h is the length of the spanning tree path from the indicated frame to the ground frame. However, the results may be cached, reducing query time but adding some time cost for the bookkeeping—cached CMTs may need to be invalidated after structure or state changes. Algorithm G.2 in Appendix G gives one method for such a precomputation. The current implementation periodically re-computes all CMTs at about 15 Hz, as they are needed for rendering at that rate anyway. When more frequent CMT updates are required, e.g. to compute more than one numeric solve iteration between renders, they are explicitly initiated.

Algorithm D.14: TREE?(j)

```
Input: joint $j$
Output: true iff $j$ is a tree joint
return $(p_{c_j} = j)$
```

Algorithm D.15: CLOSURE?(j)

```
Input: joint $j$
Output: true iff $j$ is a closure joint
return $(p_{c_j} \neq j)$
```

Algorithm D.16: ROOTLINK?(k)

```
Input: link $k$
Output: true iff $k$ is a root link
return $(p_k = r_k)$
```

Algorithm D.17: ROOTJOINT?(j)

```
Input: joint $j$
Output: true iff $j$ is a root joint
return $(r_{c_j} = j)$
```

Algorithm D.18: ROOTNAME(k)

```
Input: link $k$
Output: a generated name for the root joint of $k$
let $\chi \leftarrow$ a unique numeric id ▷avoid name collision
return $n_k$"-root-"$\chi$
```

Algorithm D.19: OUTCROSSING?(j)

```
Input: joint $j$ in linkage $L$ (i.e. $j \in J_L$)
Output: true iff $j$ is outcrossing
return $(c_j \in K_L$ and $p_j \in K_{p_L})$
```

Algorithm D.20: INCROSSING?(j)

```
Input: joint $j$ in linkage $L$ (i.e. $j \in J_L$)
Output: true iff $j$ is incrossing
return $(p_j \in K_L$ and $c_j \in K_{p_L})$
```

Algorithm D.21: CROSSING?(j)

```
Input: joint $j$
Output: true iff $j$ is crossing
return (OUTCROSSING?$(j)$ or INCROSSING?$(j)$)
```

Algorithm D.22: $\textsc{ClampV}(\boldsymbol{x}, \boldsymbol{a}, I)$

```
Input: vector $\boldsymbol{x} \in \mathbb{R}^3$, subspace index $\boldsymbol{a} \in \{0,1\}^3$, limits $I = (\boldsymbol{l} \in \mathbb{R}^3, \boldsymbol{u} \in \mathbb{R}^3)$
Output: $\boldsymbol{x}$ projected to $\mathcal{A}(\boldsymbol{a})$ and restricted to $I$
for $0 \le i < 3$ do
    if $a_i = 0$ then $x_i \leftarrow 0$
    else if $x_i < l_i$ then $x_i \leftarrow l_i$ else if $x_i > u_i$ then $x_i \leftarrow u_i$
end
return $\boldsymbol{x}$
```

Algorithm D.23: $\textsc{ClampX}(X, Y, I)$

```
Input: transform $X = (\boldsymbol{t}, \boldsymbol{\theta})$, joint type $Y = \mathcal{A}(\boldsymbol{a})$, limits $I = ((\boldsymbol{l}_t, \boldsymbol{l}_\theta), (\boldsymbol{u}_t, \boldsymbol{u}_\theta))$
Output: $X$ clamped to $Y|_I$
with $\boldsymbol{a} = (\boldsymbol{a}_t, \boldsymbol{a}_\theta),\ \boldsymbol{a}_t, \boldsymbol{a}_\theta \in \{0,1\}^3$
$\boldsymbol{t} \leftarrow \textsc{ClampV}(\boldsymbol{t}, \boldsymbol{a}_t, \boldsymbol{l}_t, \boldsymbol{u}_t)$
$\triangleright$ check both aliases of $\boldsymbol{\theta}$ within $B^3(2\pi)$
if $\|\boldsymbol{\theta}\| > 0$ then let $\boldsymbol{\theta}' \leftarrow (1 - 2\pi/\|\boldsymbol{\theta}\|)\boldsymbol{\theta}$ else let $\boldsymbol{\theta}' \leftarrow \mathbf{0}$
let $\boldsymbol{\omega} \leftarrow \textsc{ClampV}(\boldsymbol{\theta}, \boldsymbol{a}_\theta, (\boldsymbol{l}_\theta, \boldsymbol{u}_\theta))$, $\boldsymbol{\omega}' \leftarrow \textsc{ClampV}(\boldsymbol{\theta}', \boldsymbol{a}_\theta, (\boldsymbol{l}_\theta, \boldsymbol{u}_\theta))$
if $\|\boldsymbol{\omega} - \boldsymbol{\theta}\| < \|\boldsymbol{\omega}' - \boldsymbol{\theta}'\|$ then $\boldsymbol{\theta} \leftarrow \boldsymbol{\omega}$ else $\boldsymbol{\theta} \leftarrow \boldsymbol{\omega}'$
if $\|\boldsymbol{\theta}\| \ge \pi$ then $\boldsymbol{\theta} \leftarrow (1 - 2\pi/\|\boldsymbol{\theta}\|)\boldsymbol{\theta}$
return $X$
```

Algorithm D.24: $\textsc{cmt}(f)$

```
Input: identifier $f$ for coordinate frame $F_f$
Output: composite model transform $X_{g_0 \leftarrow f}$ from frame $F_f$ to $F_{g_0}$
if $f = g_0$ then return $\mathbf{0}$ $\triangleright$ identity transform (base case)
else if $f$ is a link then return $\textsc{cmt}(p_{p_f})X_{p_f}$ $\triangleright X_{p_f}$ from Eq. 3.9 (link CMT)
else if $f = pm_j$, $j$ a joint then return $\textsc{cmt}(p_j)P_j$ $\triangleright$ (parent mobility CMT)
else return $\textsc{cmt}(c_j)C_j^{-1}$ $\triangleright$ (child mobility CMT)
```

Appendix E

Computing the Pseudoinverse

There are a number of ways to compute the pseudoinverse M^+ of an arbitrary matrix M; I use the singular value decomposition, which is regarded to be among the most accurate and numerically stable practical implementations [20]. If M is $[m \times n]$, the SVD is a decomposition consisting of three matrices

$$\underset{[m\times k]}{U}\ \underset{[k\times k]}{\Sigma}\ \underset{[k\times n]}{V^T} = \textsc{svd}(M) \text{ with } k = \min(m, n), \tag{E.1}$$

where U and V are orthogonal and Σ is a diagonal matrix of *singular values*

$$\Sigma = \begin{bmatrix} \sigma_0 & & \\ & \ddots & \\ & & \sigma_{k-1} \end{bmatrix} \tag{E.2}$$

ordered s.t. $\sigma_0 \geq \sigma_1 \geq \cdots \sigma_{k-1} \geq 0$. Since U and V are orthogonal, their inverses are simply their transposes. And the inverse of Σ is just

$$\Sigma^{-1} = \begin{bmatrix} 1/\sigma_0 & & \\ & \ddots & \\ & & 1/\sigma_{k-1} \end{bmatrix}, \tag{E.3}$$

though taking $1/0 = \infty$ would result in a matrix with infinite entries if any of the singular values are 0. If we instead define Σ^+ to be the same computation but with $1/0$ taken to be zero, we can compute the pseudoinverse of M as

$$M^+ = V\Sigma^+U^T. \tag{E.4}$$

E.1 Damping

In theory the SVD pseudoinverse works even for M that are singular or near-singular. But in practical implementations using limited precision—including floating point—arithmetic there is a numeric issue for matrices that are near-singular. In this case even the best limited-precision implementations can result in M^+ with inaccurate large entries, which causes instability in the iteration. Essentially, for a near-singular matrix, one or more of the singular values will be very small but not exactly zero, and will not be accurately represented in limited precision arithmetic. When the reciprocal of these entries is taken in the computation of Σ^+, the small inaccuracies will be magnified into large ones.

One well-known way to address this is to introduce a *damping* factor $\lambda > 0$, and to replace Σ^+ with

$$\Sigma^{+\lambda} = \begin{bmatrix} \frac{\sigma_0}{\sigma_0^2+\lambda^2} & & \\ & \ddots & \\ & & \frac{\sigma_{k-1}}{(\sigma_{k-1})^2+\lambda^2} \end{bmatrix}. \tag{E.5}$$

Definition 67 The *λ-damped pseudoinverse* of M may then be computed as

$$M^{+\lambda} = V\Sigma^{+\lambda}U^T. \tag{E.6}$$

E.2 Computing the SVD

I use the DGESDD algorithm from LAPACK [7] to compute the SVD. This algorithm has been ported to various languages; since my system is mainly implemented in Java, I use the netlib-java library [111], which provides a bridge to optimized implementations where available without sacrificing portability. Like other typical SVD implementations, DGESDD has asymptotic time complexity $O(lk^2)$ where l is the larger matrix dimension and k the smaller [23, 12].

I also measured the actual DGESDD runtime in the implementation, as shown in table E.1. This informal test was performed on a modern mid-range laptop under normal working load, and is informative as it shows the scale of the SVD sizes we can expect to compute in an interactive application under current real-world conditions. For example, to maintain a minimum response time to the operator of say 100ms, up to about 100 10x10 SVD, 10 100x100, or 4 500x100 could be computed in the available time. In actuality, of course, there will be less time available because there are other things to compute. In practice, such as in the examples detailed in Section 5.4, the SVD often does account for a significant fraction (up to 30–40%) of the total simulation computation time.

rows	cols	average runtime
10	10	1ms
100	100	10ms
500	100	25ms
500	500	1400ms

Table E.1: netlib-java DGESDD timings on a current mid-range laptop.

Appendix F

Prioritized Damped Least Squares

Section 4.2.1 introduced the two-level *task priority* approach, with residual compensation and nullspace projection, to combine the solution of two damped least squares iterations. Siciliano and Slotine extended this to an arbitrary number of cascaded priority levels in [142], and recently some further improvements were reported by Baerlocher and Boulic in [11]. I call their method *prioritized damped least squares* (PDLS).

One of the main contributions in [11] is that the formulas for residual compensation and nullspace projection in Eq. 4.21 extend directly to a relatively efficient multi-priority formulation, which takes as input a set of residual vectors $\boldsymbol{e}_l$ paired with corresponding Jacobians J_l, one pair for each priority level l. Per-level damping factors λ_l and clamping thresholds γ_l are also inputs. Cast into the terminology of my framework, their *priority loop* begins by initializing the change in tree state $\Delta\boldsymbol{x} \leftarrow \mathbf{0}$ and an $[f \times f]$ nullspace projector matrix $N \leftarrow \mathrm{I}_f$ (the $[f \times f]$ identity matrix). Then, for each priority level from highest to lowest, compute

1. the *restricted Jacobian* $J_r = J_l N$
2. the *compensated residual* $\boldsymbol{e}_c = \textsc{clamp}(\boldsymbol{e}_l, \gamma_l) + J_l \Delta\boldsymbol{x}$
3. $\Delta\boldsymbol{x} \leftarrow \Delta\boldsymbol{x} - J_r^{+\lambda_l}\boldsymbol{e}_c$ and $N \leftarrow N - J_r^+ J_r$.

(The time complexity and convergence properties of PDLS are covered with the Solve algorithm in Section 4.8.)

F.1 Posture Variation

After iteration of the priority loop for any level, N is an aggregate projector onto the combined nullspace of all higher priority levels. Thus, when the loop terminates after processing the lowest-priority level, N can be used to project one final tree state differential vector $\Delta\boldsymbol{x}_p$; this time onto the nullspace of all priority levels. In some cases this nullspace will be empty, but if not, the effect is to bias the tree pose of the linkage in the direction of $\Delta\boldsymbol{x}_p$, which was called a *posture variation* vector in [11].

F.2 Limits on Tree Joint DoF

A final feature presented in [11] is that the priority loop is wrapped in a surrounding *limiting loop* which checks the final computed $\Delta\boldsymbol{x}$, after handling all priority levels and also the posture variation, to verify that when added to the current tree state $\boldsymbol{x}$, the resulting new tree state $\boldsymbol{x} + \Delta\boldsymbol{x}$ keeps all tree joint DoF within limits. Each DoF that does exceed its limits is *pinned.* The priority loop is re-executed as long as newly-pinned DoF are found, each time excluding all pinned DoF from consideration by changing the initial entry on the diagonal of the nullspace projection matrix N from 1 to 0 for pinned DoF. The process continues until a priority loop completes without pinning any DoF, and then the final $\Delta\boldsymbol{x}$ is adjusted so that $\boldsymbol{x} + \Delta\boldsymbol{x}$ brings each pinned DoF to the limit (upper or lower) that was hit.

Appendix G

Computing the Forward Kinematic Mapping

In Section 4.8, the current state vector $\boldsymbol{y}_s$ for the set of closure joints C_s processed by SOLVE is needed for computing both residuals and Jacobians. The main idea for computing $\boldsymbol{y}_s$ is to (1) update the appropriate tree joint DoF from $\boldsymbol{x}_s$, (2) update all affected composite model transforms up to and including those for the parent and child mobility frames of all closures in C_s, and then (3) use those CMTs to back-compute the necessary closure joint mobility transforms. Technically only the CMTs for the mobility frames in the closure joints are required, but the other CMTs along the support chains are also computed in order to get them.

FK, Algorithm G.3 below, performs this computation for an entire (flattened, Def. 40) linkage in a depth-first traversal of the spanning tree from the ground link. This effectively updates the full closure state $\boldsymbol{y}$ for the linkage; since a particular call to SOLVE may only be for a part of a linkage, I also present an optimized FK$_s$ algorithm which updates only the necessary CMTs for computing the restricted closure state $\boldsymbol{y}_s$. Both algorithms are written in terms of corresponding CMT updates: UPDATEALLCMTS and UPDATERESTRICTEDCMTS, Algorithms G.2 and G.4, respectively. The resulting cached CMTs are useful not only for computing $\boldsymbol{y}$ but also for other purposes, including rendering and the Jacobian computation algorithm given in Section 4.9.2.

The CMT update algorithms each ensure that the CMTs for a given node are computed only after CMTs are computed for all spanning tree ancestors, which makes the update for the CMTs within each node a local computation depending only on the CMTs of frames in directly adjacent nodes. A shared sub-routine, UPDATENODECMTS (Algorithm G.1), performs this local computation in $O(1)$, making the full UPDATEALLCMTS $O(|J| + |K|)$ where J is the set of joints and K the set of links in a linkage. Since $C \subset J$, the time complexity of FK is dominated by the call to UPDATEALLCMTS.

We now move to the restricted versions of these algorithms. UPDATERESTRICTEDCMTS has two modes of operation: on the first call after any structure change it updates a cached ordered set R_s of the set of links and joints whose CMTs must be recomputed in later calls. Since many convergence iterations are often performed between structure changes, the latter mode is most common, and UPDATERESTRICTEDCMTS is $O(|R_s| + |C_s|)$ in this case, with $R_s \subset (K \cup J)$. However, when R_s needs to be recomputed the worst-case running time becomes

$$\begin{gathered} O(u \log u) \text{ expected} \\ \text{with } u = |J \setminus C_s||C_s| \end{gathered} \tag{G.1}$$

due to (a) the depth sort, (b) duplicate removal (e.g.) using a hash, and (c) tracing the full support cycle of each closure joint back to the least common ancestor of its parent and child links (Def. 17) (assume that the support chains $S\downarrow$ and $S\uparrow$ forming these cycles are known; they are computed as a side-effect of the ANALYZE algorithm). This upper time bound is reachable in a pathological case where (1) every closure joint is attached in parallel between the same pair of links and (2) the only tree joints in the linkage are on the support chains for these links. Closure joint supports do not typically overlap in such a massive way, and even when they do, this computation is only a log factor slower than ANALYZE, which also runs in this context.

Algorithm G.1: UPDATENODECMTS(i)

```
Input: link or joint $i$
Output: updated CMTs for all coordinate frames in $i$
if $i$ is a link then $X_{g_0 \leftarrow i} \leftarrow X_{g_0 \leftarrow p_i} X_{p_i}$
else if $i$ is a tree joint then $X_{g_0 \leftarrow pm_i} \leftarrow X_{g_0 \leftarrow p_i} P_i$, $X_{g_0 \leftarrow cm_i} \leftarrow X_{g_0 \leftarrow pm_i} M_i$
else $X_{g_0 \leftarrow pm_i} \leftarrow X_{g_0 \leftarrow p_i} P_i$, $X_{g_0 \leftarrow cm_i} \leftarrow X_{g_0 \leftarrow c_i} C_i^{-1}$ $\triangleright$ i is a closure
```

Algorithm G.2: UPDATEALLCMTS(L)

```
Input: linkage $L$
Output: updated CMTs for all links and joints in the flattening of $L$
foreach link or joint $i$ in a pre-order spanning tree DFS from $g_0$ do
    UPDATENODECMTS($i$)
```

Algorithm G.3: FK($L, C, \boldsymbol{x}$)

```
Input: flattened linkage $L$ with joints $J$, closure joints $C$, tree state $\boldsymbol{x}$
Output: closure state $\boldsymbol{y}$ corresponding to $C$
foreach unlocked tree joint $j$ do set DoF of $j$ from $\boldsymbol{x}$
UPDATEALLCMTS($L$)
foreach $j \in C$ do set corresponding elements of $\boldsymbol{y}$ to $M_j = X_{g_0 \leftarrow cm_j}^{-1} X_{g_0 \leftarrow pm_j}$
return $\boldsymbol{y}$
```

Algorithm G.4: UPDATERESTRICTEDCMTS(L, C_s)

```
Input: flattened linkage $L$, any subset of closure joints $C_s$
Output: updated CMTs for all moving links and joints in the support of $C_s$
if first call since any structure change then
    let $R_s$ be an ordered set of links and joints, initially empty
    foreach $j \in C_s$ do
        foreach joint $i \in S{\downarrow}_j$ do $R_s \leftarrow R_s \cup \{i, c_i\}$
        foreach joint $i \in S{\uparrow}_j$ do $R_s \leftarrow R_s \cup \{i, c_i\}$
    end
    sort $R_s$ in order of increasing tree depth (i.e. topological distance from $g_0$)
end
foreach link or joint $i$ in $R_s$ in order do UPDATENODECMTS($i$)
foreach $j \in C_s$ do UPDATENODECMTS($j$)
```

Algorithm G.5: FK$_s$($L, C_s, T_s, \boldsymbol{x}_s$)

```
Input: flattened linkage $L$; $C_s$, $T_s$, and $\boldsymbol{x}_s$ as for SOLVE
Output: closure state $\boldsymbol{y}_s$ corresponding to $C_s$
foreach unlocked tree joint $j \in T_s$ do set DoF of $j$ from $\boldsymbol{x}_s$
UPDATERESTRICTEDCMTS($L, C_s$)
foreach $j \in C_s$ do set corresponding elements of $\boldsymbol{y}_s$ to $M_j = X_{g_0 \leftarrow cm_j}^{-1} X_{g_0 \leftarrow pm_j}$
return $\boldsymbol{y}_s$
```

Appendix H

Computing the Sub-Jacobian $J_{c \leftarrow t}$

In Section 4.9.2, each $[6 \times 6]$ block $J_{c \leftarrow t}$ of J gives the partial derivatives of the mobility transform $M_c = (\boldsymbol{t}_c, \boldsymbol{\theta}_c)$ for a particular closure joint c with respect to the mobility transform $M_t = (\boldsymbol{t}_t, \boldsymbol{\theta}_t)$ of a particular tree joint t:

$$\underset{[6\times6]}{J_{c \leftarrow t}} = \frac{\partial M_c}{\partial M_t} = \frac{\partial(\boldsymbol{t}_c, \boldsymbol{\theta}_c)}{\partial(\boldsymbol{t}_t, \boldsymbol{\theta}_t)} = \begin{bmatrix} \underset{[3\times3]}{\frac{\partial \boldsymbol{t}_c}{\partial \boldsymbol{t}_t}} & \underset{[3\times3]}{\frac{\partial \boldsymbol{t}_c}{\partial \boldsymbol{\theta}_t}} \\ \underset{[3\times3]}{\frac{\partial \boldsymbol{\theta}_c}{\partial \boldsymbol{t}_t}} & \underset{[3\times3]}{\frac{\partial \boldsymbol{\theta}_c}{\partial \boldsymbol{\theta}_t}} \end{bmatrix}. \tag{H.1}$$

If t is not in the support of c then $J_{c \leftarrow t} = \mathbf{0}$. Otherwise its computation depends on the topological relationship of c and t, for which there are 8 combinatorial cases depending on whether (1) t is in the support downchain or the upchain of c (Def. 17); (2) c is inverted (i.e. $\phi_c = -1$); or (3) t is inverted. Fortunately these can be boiled down to just two, depending on whether the forward sense of the mobility action of t is parallel or anti-parallel to the forward sense of the action of c. UPDATEJACOBIAN computes rigid transforms Q and R from the cached CMTs such that

$$M_c = RM_tQ \text{ when } t \parallel c \tag{H.2}$$

$$M_c = RM_t^{-1}Q \text{ when } t \nparallel c. \tag{H.3}$$

I.e., Q is the transform from the start frame for M_c to the topologically nearest mobility frame of t, and R is the transform from the other mobility frame of t to the destination frame of M_c. Figure H-1 illustrates these relationships.

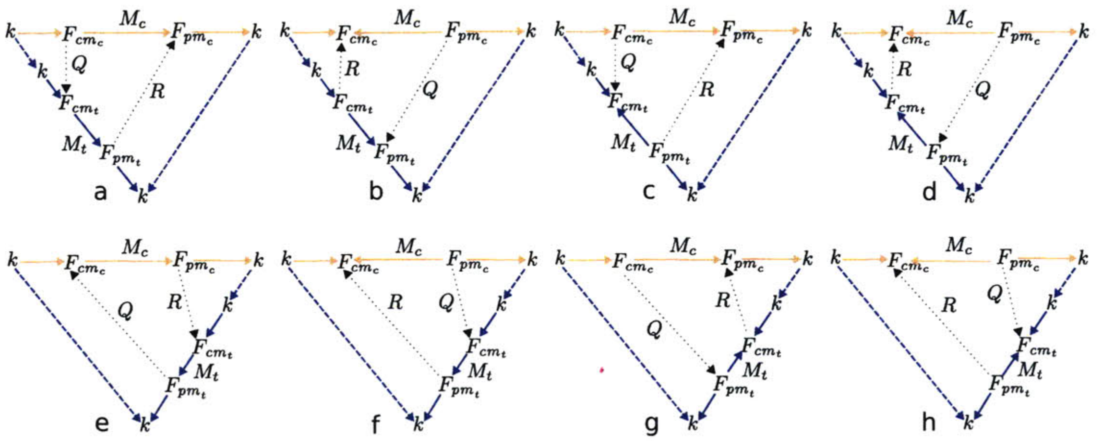


Figure H-1: Topological cases for $J_{c \leftarrow t}$.
There are 8 possible combinatorial cases depending on whether (1) tree joint t is in the support upchain (top row) or downchain (bottom row) of closure joint c; (2) c has inverted mobility—cases (b), (d), (f), (h); (3) t has inverted mobility—cases (c), (d), (g), (h). Cases (a), (d), (f), and (g) have $t \parallel c$; the others have $t \nparallel c$. In this figure links are labelled k; the closure joint c is shown at the top of each diagram, broken down into three sub-transforms (child-to-mobility, mobility M_c, and mobility-to-parent, also see Fig. 3-4); and the tree joint t is similarly broken down. Dashed arrows represent topological connectivity in the linkage graph. The thinner dotted Q and R arrows indicate the corresponding transforms.

As explained in Appendix B, transforms in the $(\boldsymbol{t}, \boldsymbol{\theta})$ representation do not compose directly, but can be converted to the unit quaternion representation $(\boldsymbol{t}, \exp(\boldsymbol{\theta}))$, composed, and then converted back using the log map:

$$\begin{aligned}
M_c &= R M_t Q \text{ when } t \parallel c \\
(\boldsymbol{t}_c, \boldsymbol{\theta}_c) &= (\boldsymbol{t}_r, \boldsymbol{\theta}_r)(\boldsymbol{t}_t, \boldsymbol{\theta}_t)(\boldsymbol{t}_q, \boldsymbol{\theta}_q) \\
(\boldsymbol{t}_c, \exp(\boldsymbol{\theta}_c)) &= (\boldsymbol{t}_r, \exp(\boldsymbol{\theta}_r))(\boldsymbol{t}_t, \exp(\boldsymbol{\theta}_t))(\boldsymbol{t}_q, \exp(\boldsymbol{\theta}_q)) \\
(\boldsymbol{t}_c, \mathring{\boldsymbol{q}}_c) &= (\boldsymbol{t}_r, \mathring{\boldsymbol{q}}_r)(\boldsymbol{t}_t, \mathring{\boldsymbol{q}}_t)(\boldsymbol{t}_q, \mathring{\boldsymbol{q}}_q) \\
\boldsymbol{t}_c &= \boldsymbol{t}_r + \mathring{\boldsymbol{q}}_r \boldsymbol{t}_t + (\mathring{\boldsymbol{q}}_r \mathring{\boldsymbol{q}}_t)\boldsymbol{t}_q && \text{applying Eq. 3.15} \quad \text{(H.4)} \\
\boldsymbol{\theta}_c &= \log(\mathring{\boldsymbol{q}}_c) = \log(\mathring{\boldsymbol{q}}_r \mathring{\boldsymbol{q}}_t \mathring{\boldsymbol{q}}_q) && \text{applying Eq. 3.15} \quad \text{(H.5)}
\end{aligned}$$

and similarly for the anti-parallel case

$$
\begin{aligned}
M_c &= RM_t^{-1}Q \text{ when } t \nparallel c \\
(\boldsymbol{t}_c, \mathring{\boldsymbol{q}}_c) &= (\boldsymbol{t}_r, \mathring{\boldsymbol{q}}_r)(\boldsymbol{t}_t, \mathring{\boldsymbol{q}}_t)^{-1}(\boldsymbol{t}_q, \mathring{\boldsymbol{q}}_q) \\
\boldsymbol{t}_c &= \boldsymbol{t}_r + (\mathring{\boldsymbol{q}}_r \mathring{\boldsymbol{q}}_t^*)(\boldsymbol{t}_q - \boldsymbol{t}_t) && \text{applying Eqs. 3.15 and 3.16} && \text{(H.6)} \\
\boldsymbol{\theta}_c &= \log(\mathring{\boldsymbol{q}}_c) = \log(\mathring{\boldsymbol{q}}_r \mathring{\boldsymbol{q}}_t^* \mathring{\boldsymbol{q}}_q) && \text{applying Eqs. 3.15 and 3.16.} && \text{(H.7)}
\end{aligned}
$$

Computing $J_{c\leftarrow t}$ thus reduces to computing partial derivatives of Eqs. H.4–H.7. These can be collected in a general partitioned matrix form:

$$
\begin{aligned}
\underset{[6\times6]}{J_{c\leftarrow t}} = \frac{\partial(\boldsymbol{t}_c, \boldsymbol{\theta}_c)}{\partial(\boldsymbol{t}_t, \boldsymbol{\theta}_t)} = \begin{bmatrix} \frac{\partial \boldsymbol{t}_c}{\partial \boldsymbol{t}_t} & \frac{\partial \boldsymbol{t}_c}{\partial \boldsymbol{\theta}_t} \\ \frac{\partial \boldsymbol{\theta}_c}{\partial \boldsymbol{t}_t} & \frac{\partial \boldsymbol{\theta}_c}{\partial \boldsymbol{\theta}_t} \end{bmatrix} &= \underset{[6\times7]}{\frac{\partial(\boldsymbol{t}_c,\boldsymbol{\theta}_c)}{\partial(\boldsymbol{t}_c,\mathring{\boldsymbol{q}}_c)}} \underset{[7\times7]}{\frac{\partial(\boldsymbol{t}_c,\mathring{\boldsymbol{q}}_c)}{\partial(\boldsymbol{t}_t,\mathring{\boldsymbol{q}}_t)}} \underset{[7\times6]}{\frac{\partial(\boldsymbol{t}_t,\mathring{\boldsymbol{q}}_t)}{\partial(\boldsymbol{t}_t,\boldsymbol{\theta}_t)}} \\
&= \begin{bmatrix} \frac{\partial \boldsymbol{t}_c}{\partial \boldsymbol{t}_c} & \frac{\partial \boldsymbol{t}_c}{\partial \mathring{\boldsymbol{q}}_c} \\ \frac{\partial \boldsymbol{\theta}_c}{\partial \boldsymbol{t}_c} & \frac{\partial \boldsymbol{\theta}_c}{\partial \mathring{\boldsymbol{q}}_c} \end{bmatrix} \begin{bmatrix} \frac{\partial \boldsymbol{t}_c}{\partial \boldsymbol{t}_t} & \frac{\partial \boldsymbol{t}_c}{\partial \mathring{\boldsymbol{q}}_t} \\ \frac{\partial \mathring{\boldsymbol{q}}_c}{\partial \boldsymbol{t}_t} & \frac{\partial \mathring{\boldsymbol{q}}_c}{\partial \mathring{\boldsymbol{q}}_t} \end{bmatrix} \begin{bmatrix} \frac{\partial \boldsymbol{t}_t}{\partial \boldsymbol{t}_t} & \frac{\partial \boldsymbol{t}_t}{\partial \boldsymbol{\theta}_t} \\ \frac{\partial \mathring{\boldsymbol{q}}_t}{\partial \boldsymbol{t}_t} & \frac{\partial \mathring{\boldsymbol{q}}_t}{\partial \boldsymbol{\theta}_t} \end{bmatrix} \\
J_{c\leftarrow t} &= \begin{bmatrix} \mathrm{I}_{3\times3} & \mathbf{0}_{3\times4} \\ \mathbf{0}_{3\times3} & \frac{\partial \boldsymbol{\theta}_c}{\partial \mathring{\boldsymbol{q}}_c} \end{bmatrix} \begin{bmatrix} \frac{\partial \boldsymbol{t}_c}{\partial \boldsymbol{t}_t} & \frac{\partial \boldsymbol{t}_c}{\partial \mathring{\boldsymbol{q}}_t} \\ \mathbf{0}_{4\times3} & \frac{\partial \mathring{\boldsymbol{q}}_c}{\partial \mathring{\boldsymbol{q}}_t} \end{bmatrix} \begin{bmatrix} \mathrm{I}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{4\times3} & \frac{\partial \mathring{\boldsymbol{q}}_t}{\partial \boldsymbol{\theta}_t} \end{bmatrix}. \qquad \text{(H.8)}
\end{aligned}
$$

There are five non-trivial sub-matrices to compute in Eq. H.8. The components of the $[3 \times 4]$ sub-matrix $\partial\boldsymbol{\theta}_c/\partial\mathring{\boldsymbol{q}}_c$ are the partials of the logarithmic map, given in Eq. B.17 in Section B.4. Similarly, the components of the $[4 \times 3]$ sub-matrix $\partial\mathring{\boldsymbol{q}}_t/\partial\boldsymbol{\theta}_t$ are the partials of the exponential map, given in Eq. B.15 in Section B.5. The three remaining sub-matrices have different formulas depending on the parallelism of c and t (quantities derived from Q and R are constants with respect to these derivatives):

$$
\begin{array}{ll}
\underline{\text{when } t \parallel c} & \underline{\text{when } t \nparallel c} \\
\dfrac{\partial \boldsymbol{t}_c}{\partial \boldsymbol{t}_t} = \mathrm{ROT}(\mathring{\boldsymbol{q}}_r) & \dfrac{\partial \boldsymbol{t}_c}{\partial \boldsymbol{t}_t} = -\mathrm{ROT}(\mathring{\boldsymbol{q}}_r)\mathrm{ROT}(\mathring{\boldsymbol{q}}_t^*) \\
\dfrac{\partial \boldsymbol{t}_c}{\partial \mathring{\boldsymbol{q}}_t} = \mathrm{ROT}(\mathring{\boldsymbol{q}}_r)\dfrac{\partial \mathrm{ROT}(\mathring{\boldsymbol{q}}_t)}{\partial \mathring{\boldsymbol{q}}_t}\boldsymbol{t}_q & \dfrac{\partial \boldsymbol{t}_c}{\partial \mathring{\boldsymbol{q}}_t} = \mathrm{ROT}(\mathring{\boldsymbol{q}}_r)\dfrac{\partial (\mathrm{ROT}(\mathring{\boldsymbol{q}}_t))^{-1}}{\partial \mathring{\boldsymbol{q}}_t}(\boldsymbol{t}_q - \boldsymbol{t}_t) \\
\dfrac{\partial \mathring{\boldsymbol{q}}_c}{\partial \mathring{\boldsymbol{q}}_t} = \mathrm{QMM}(\mathring{\boldsymbol{q}}_r)\mathrm{QMM}^*(\mathring{\boldsymbol{q}}_q) & \dfrac{\partial \mathring{\boldsymbol{q}}_c}{\partial \mathring{\boldsymbol{q}}_t} = \mathrm{QMM}(\mathring{\boldsymbol{q}}_r)\mathrm{QMM}^*(\mathring{\boldsymbol{q}}_q)\left[\begin{smallmatrix}1 & 0 \\ 0 & -\mathrm{I}_{3\times3}\end{smallmatrix}\right]
\end{array} \qquad \text{(H.9)}
$$

QMM and QMM* refer to functions described by Horn in [71] that produce $[4 \times 4]$

multiplication matrices from a quaternion, transforming quaternion multiplication into the multiplication of a matrix and a vector. Appendix C gives their definition (Eq. C.5).

ROT is a well-known function (e.g. see [2]) that produces the $[3 \times 3]$ orthogonal rotation matrix corresponding to a quaternion. ROT and the partial derivatives $\partial \text{ROT}(\mathring{\boldsymbol{q}})/\partial \mathring{\boldsymbol{q}}$ and $\partial (\text{ROT}(\mathring{\boldsymbol{q}}))^{-1}/\partial \mathring{\boldsymbol{q}}$, which are represented as $[[3 \times 3] \times 4]$ tensors, are also given in Appendix C.

Appendix I

The *TRACK* Interface Device

As an initial contribution to ATHLETE operations (Sec. 5.3) I designed and fabricated a hardware operator interface device called the Tele-Robotic ATHLETE Controller for Kinematics (TRACK), Figure I-1. TRACK is a sensed but un-actuated 1:8 mechanical scale model of one ATHLETE limb. On-board sensors and electronics continually read-out the model pose to a laptop or workstation via a standard Universal Serial Bus (USB) connection. TRACK provides ATHLETE operators a physical representation of the limb which can be directly manipulated by hand into desired poses and motions.

Previous input devices used for ATHLETE, including the PHANTOM® haptic device, do not mimic the limb structure and allow only for specifying the end-effector (EE) configuration. In some contexts this does not uniquely identify a pose for the rest of the leg. For example, in many cases, some rotation about the wheel axis is permissible, so specifying a full spatial pose for the wheel still allows a continuous space of leg postures. Even if such rotation is not allowed there is still generally a discrete set of possible postures that realize a given EE spatial pose. TRACK is useful in cases where the whole-leg posture must be unambiguously commanded, for example, to maximize rigidity, available range of motion, or to avoid nearby obstacles. Also, because TRACK mimics the link lengths and geometry of ATHLETE as well as the joint angle limits, its kinematic workspace is representative of the actual ATHLETE limb workspace.

Two TRACK units were fabricated. One is retained at MIT. The other was delivered to JPL, incorporated with JPL's Ensemble operations software suite, and used successfully in over 100 ATHLETE motions in a field test at Moses Lake, Washington (Figure I-1, upper right) [103]. TRACK has also been integrated with a version of the mixed real/virtual interface system presented in this thesis, and used in an experiment combined with virtual articulations (Fig. 5-5) [159].

Master-slave teleoperation is a well-established practice in robotics, so TRACK is not intended to be a significant new scientific result. Rather it was conceived as, and has been demonstrated to be, a useful practical tool to aid ATHLETE operations.

I.1 Features and Implementation

TRACK features

- rotary sensors (potentiometers) for each kinematic DoF in the ATHLETE limb, and an extra rotary sensor for the wheel rotation
- spring-loaded friction bearings to hold joint pose against gravity
- joint mobility, link bounding volumes, and hence reachable workspace similar to ATHLETE
- tri-color R/G/B LEDs for user feedback at each joint and the wheel
- two momentary-contact tactile pushbuttons for user input at each joint and the wheel
- USB (1.1 and 2.0 compatible) communications and power
- on-board firmware with calibration storage and ASCII monitor
- host-side Java interface library compatible with GNU/Linux, Mac OS X, and Windows.

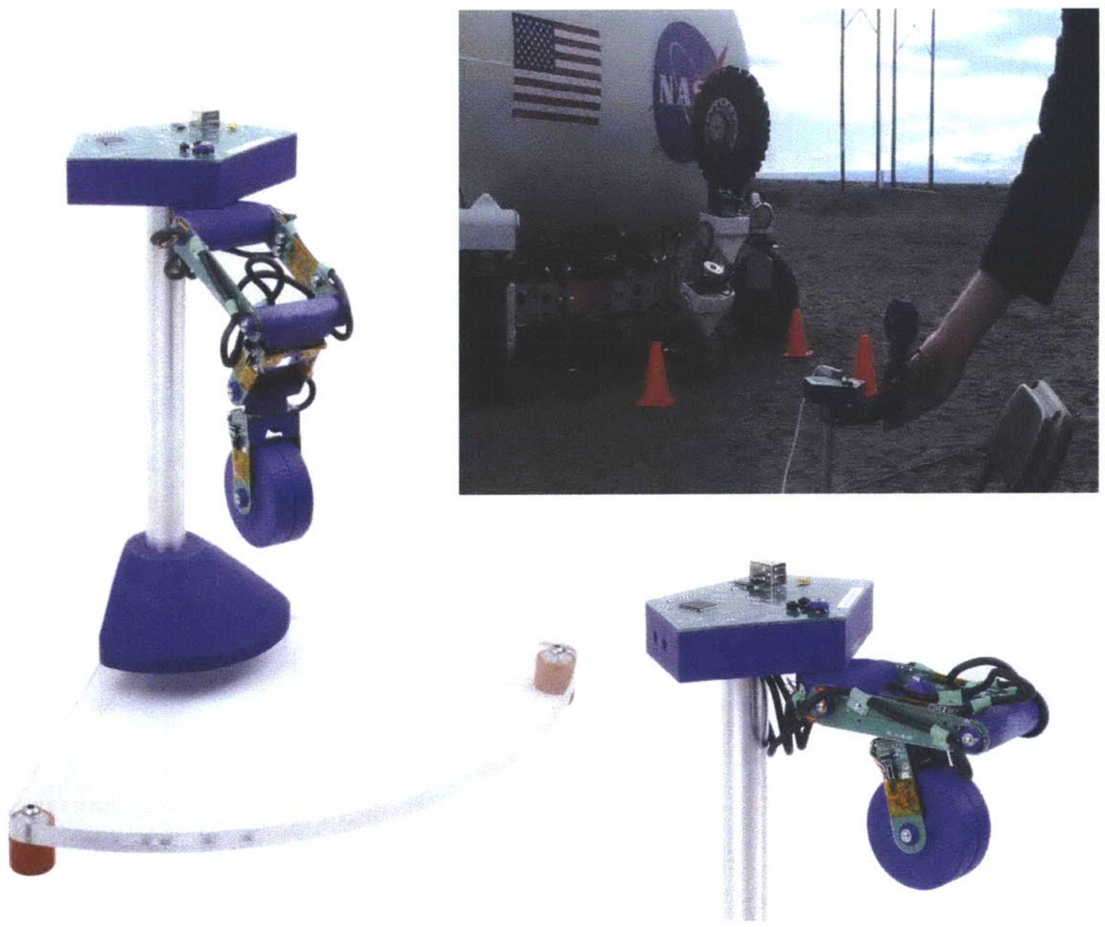


Figure I-1: Tele-Robotic ATHLETE Controller for Kinematics (TRACK). TRACK is a sensed but un-actuated table-top scale model of one ATHLETE limb which I built as an additional aid for human operators of ATHLETE. TRACK can be directly manipulated a means of specifying a pose or a motion for a selected limb, and was incorporated with JPL's Ensemble operations software suite and used successfully in over 100 ATHLETE motions in a field test at Moses Lake, Washington [103]. It has also been integrated with a version of the mixed real/virtual interface system presented in this thesis, and used in an experiment combined with virtual articulations (Fig. 5-5) [159].

Effort was made in the design of TRACK to match the ATHLETE joint ranges of motion, link geometry, and overall reachable workspace. For example, the TRACK knee can tuck into the thigh, as in ATHLETE (Figure I-1 lower right). Still, a few differences do exist. For example, the TRACK hardware may not be able to reach every configuration that the actual ATHLETE hardware can reach due to cable wrapping and to collision with the TRACK support shaft.

TRACK is powered via USB with a maximum current draw of approximately 75mA (the limit for a passive device like TRACK, which does not explicitly negotiate for more current, is 100mA). A self-resetting fuse is included to protect the host USB hardware under any unforeseen power conditions.

Communications is also performed over USB (compatible with USB 1.1 and 2.0). The device will appear as a generic USB serial port when attached to a host, e.g. `/dev/ttyUSB0` on GNU/Linux. The firmware communicates at 115.2kbps with 8 data bits, no parity, one stop bit (8N1), and no flow control. Normally a special-purpose Java host library is used to communicate with the firmware over this port, though it is also possible to exercise the various firmware functions with a generic terminal program.

The USB-to-serial chip in TRACK is in the same family as chips commonly used in USB-to-serial dongles, and may require non-standard drivers on some hosts. The chip manufacturer provides these drivers if needed. No driver installation should be required on modern GNU/Linux systems.

I.1.1 Joint Sensing and Mobility

TRACK joint angles are sensed with ALPS RDC506002A potentiometers designed for low cost miniature robotics applications. They have no hardstops, 320° of electrical range, and ±2% linearity (and are not mapped or otherwise calibrated for linearity in TRACK). In the worst case this could mean that the joint angles are reported with up to about ±6° of error; in practice the actual error appears to be much smaller. The pot readings are digitized with 10 bits of precision, giving about 0.3° resolution.

The TRACK firmware samples all pots constantly at a configurable rate and then applies a moving-window averaging low-pass filter with configurable window size. The default sample period is 10ms and the default window size is 16, which yields about a 5Hz cutoff frequency. Jitter of about ± 1 count is still to be expected in consecutive steady-state filtered pot readings (i.e. as reported by the firmware to the host) as in practice the pot will never physically be exactly at one count. The host polls the firmware for mechanism state updates and is thus in control of the highest-level update period; 100ms is a reasonable minimum host-to-firmware polling period as it will satisfy Nyquist with respect to the firmware 5Hz low-pass filter. Longer host-to-firmware polling periods are also reasonable since in practice with TRACK highly dynamic motions will generally be transient anyway.

The firmware also stores, for each pot,

- An additive offset O which gives the raw counts when the corresponding ATHLETE joint is at zero rotation. These offsets constitute the zero calibration and are stored in nonvolatile memory on the hardware.

- A sign bias $S = \pm 1$ that gives the correspondence between positive pot counts and positive rotation of the associated joint. These bias values are intrinsic to the hardware but are reported by the firmware to allow future design revisions.

Let R be the raw reading for a given pot. Then the corresponding ATHLETE joint angle is computed as

$$\text{joint angle}^\circ = S(R - O)(333.3^\circ)/(2^{10} - 1)$$

Methods are provided in the Java host library to perform this computation. The result is un-ambiguous for physical configurations of the joint that are within the corresponding ATHLETE softstop ranges, since the those ranges are never self-overlapping. In some cases the TRACK hardware may be physically moved outside the corresponding softstop range; as long as the pot is still within its 320° electrical range, the joint angle will be reported with the same sign as the nearest softstop.

I.1.2 LEDs and Pushbuttons

Each TRACK joint, including the wheel, has a co-located constellation of two pushbuttons and one tri-color LED. The pushbuttons are rectangular with a definite orientation: the longitudinal button at each joint is aligned parallel to the length of the leg, and the transverse button is perpendicular to it. On/off control is provided for each color of each LED.

The TRACK firmware and Java host library provide access to the pushbutton and LED state, but the actual functions of the LEDs and pushbuttons are determined by higher-level software. For example, in the Ensemble implementation, the red LEDs are flashed to indicate that a TRACK joint is at a different angle than the corresponding ATHLETE joint. Pressing any button brings a simulation of ATHLETE into the same pose as TRACK, and from there the actual hardware can be commanded.

Appendix J

The Climbing Robot *Shady*

Shady, introduced in Section 6.6, is a bilaterally-symmetric compliant/proprioceptive structure climbing robot with two rotating grippers and a central circular deployable sun-shade (Figures 6-1 and 6-2). This appendix summarizes the research context of Shady with respect to other climbing robots, and gives additional details of the robot's design, control, and testing.

J.1 Research Context

Many large terrestrial structures—towers, bridges, construction scaffolds—are sparse assemblies of rigid bars connected together at structural nodes. This is also true of many in-space structures such as antennae, solar panel supports, and space-station members. I use the term *truss* or *framework* to refer to this type of structure in general, and a long-term application of *truss climbing robots* is automated assembly, repair, and inspection of trusses: one or more climbing robots could grip the bars and locomote about the truss, conveying sensors, tools, or construction materials. The robot could then either carry out the desired task on its own or cooperate with a human [74, 110].

Truss climbing is a special case of structure climbing, with some particular challenges. Many previous structure climbing robots, e.g. as in Pack et al [118], and others referenced therein, are intended to climb on assemblages of 2D planar surfaces. Only a few structure climbing robots, such as Nechba et al's "SM2" [110] and Staritz et al's "Skyworker" [148], are also designed for climbing on truss-like structures where the members are more nearly 1D links. Pole-climbing robots are also related [134, 6, 4]. The reliability of the climbing action is not quantified in these prior works, nor do any of them incorporate intentional compliance. Shady is a new mechanism with a unique compliant/proprioceptive control strategy and experimentally confirmed reliability.

The penalties for uncertainty are potentially higher for truss climbing than for climbing on planar surfaces. Consider foot placement. On a large 2D surface, foot placement can be resilient to significant parallel-plane misalignment, usually does not require strong certainty of the perpendicular distance to the surface (as the foot can often be extended until it hits the surface), and is similarly tolerant of orientation uncertainties. However, the comparable task in truss climbing—gripping a thin structural member starting from a nearby but uncertain spatial pose—can be much more sensitive: even small translation and orientation misalignments can result in a weak or missed grip.

J.2 Design

Shady's grippers (Figure J-1) are symmetric 6-bar linkage mechanisms situated in rotating "barrels". Actuated through central 50:1 worm gears, the grippers open to over 7cm in about 5s and close on the 2.5cm window bar in about 15s[1]. Each 6-bar is actually two coupled 4-bars: 0-1-2-5 and 2-3-4-5 (link 5 is the barrel), and these are both in singularity when the gripper is closed on a window bar, resulting in very large mechanical advantage and effectively zero backdriveablity at closure. Silicone rubber grip pads develop over 46N measured compression force against the window frame with very high stiction and no measurable slip when closed. When fully opened, the gripper pads retract behind the bounding plane of the barrel, allowing the barrel to brush past the window frame without collision.

[1]Closing takes longer due to the grip refinement algorithm, described later.

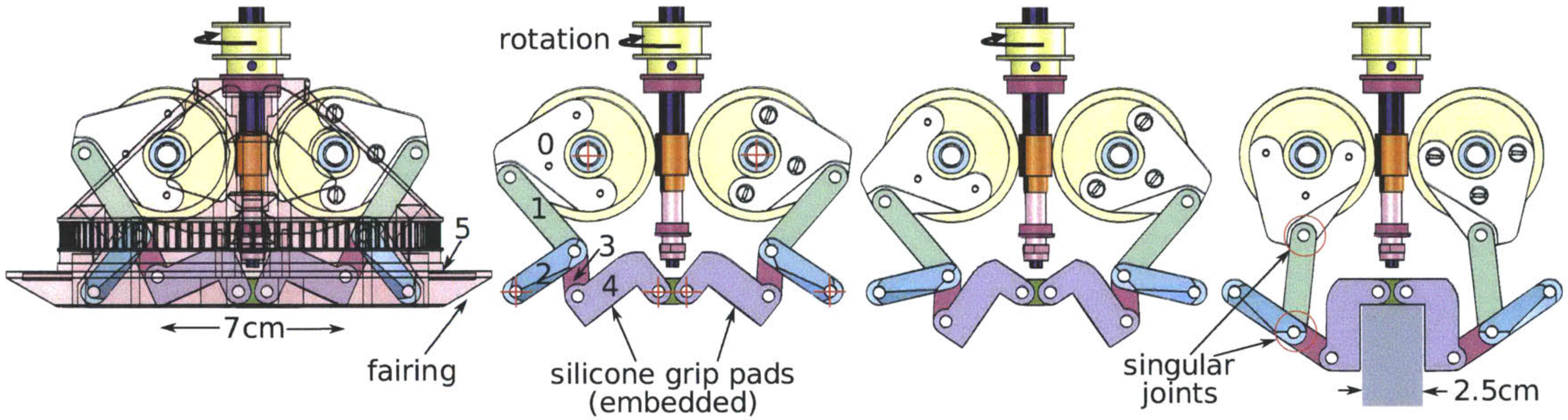


Figure J-1: Operation of the Shady grip mechanism.
Motion sequence (left to right) showing the symmetric 6-bar gripper linkages (actually coupled 4-bar pairs) closing on a window frame member, actuated by rotation of a central worm gear. Link 5, the barrel, is shown only in the initial step; red crosses in the second picture indicate the locations of pins fixing links 0, 2, and 4 to the barrel. When closed, the two four bar linkages which make up the 6 bar are both in singularity (circled joints in-line).

Shady measures 40.4cm between barrel centers (59.4cm end-to-end), a scale selected to match the geometry of the window frame on which Shady climbs. Barrel rotations are effected at about 10°/s by a series-elastic belt drive actuator incorporating a non-backdriveable worm gear (Figure J-3), leading to a maximum locomotion speed of

$$\frac{\text{one center-to-center body length}}{\text{ungrip time} + 180^\circ \cdot 1\text{s}/10^\circ + \text{grip time}} = \frac{40.4\text{cm}}{5\text{s} + 18\text{s} + 15\text{s}} \approx 1\frac{\text{cm}}{\text{s}}, \qquad (J.1)$$

which is acceptable in practice for this application—Shady can climb from the bottom to the top of the window in under seven minutes, and the apparent position of the sun moves much more slowly[2]. Shady extends about 15cm (24cm including shade mechanism and belay hook) outwards from the window frame, and weighs 3.50kg. Most of the mechanical components are made of ABS plastic formed on a rapid-prototype machine, and the largest of these are hollow to reduce mass. Shady incorporates four 3.7A-H Li-Po batteries sufficient for over 6 hours of continuous un-tethered climbing on a single charge, five in-house motor control boards which run low-level feedback control loops, a top-level real-time processor, and a miniature Bluetooth® wireless modem for communication with a command and control workstation. The barrel rotation and gripper actuators are based on 6V Maxon A-Max 22 brushed DC motors with integral 19:1 planetary gearboxes, and the fan is actuated by a small Sanyo 12GN-NA4S DC gearmotor.

J.2.1 Mechanical Compliances

Shady contains three intentional mechanical compliances (springs): a central hinge with about $+6^\circ,-1^\circ$ maximum travel (Figure J-2) that biases the grippers towards the window frame, and two actuator torsion mounts composed of antagonistic pairs of compression springs which enable about $\pm 3^\circ$ deflection on each barrel rotation (Figure J-3). Potentiometers proprioceptively measure the compliant motion in the two barrel rotations, but the hinge motion is not sensed.

[2]Also, the primary design goal for Shady is reliability, not speed.

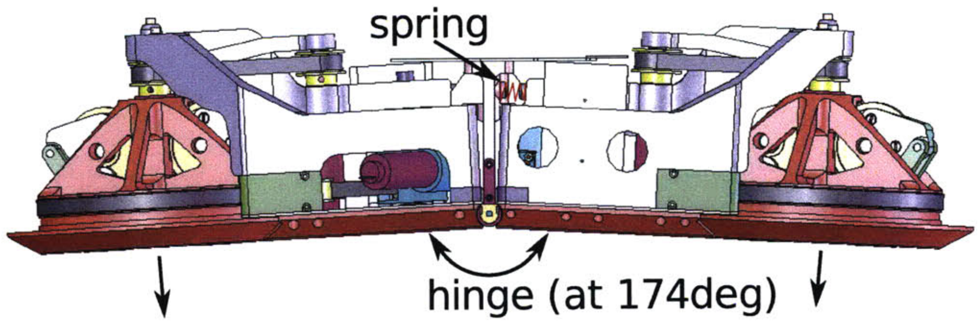


Figure J-2: Shady central hinge.
Shown extended 3° downwards on each side (the maximum travel), the passively sprung central hinge biases the grippers towards the window frame (shade mechanism not shown).

The antagonistic pair of springs on the barrel rotation motor, which is otherwise free to rotate about the same axis as the timing pinion that it drives through a worm gear, forms a series-elastic actuator [125]: the compression of the springs as measured by the potentiometer is directly proportional to the actuator's applied torque, and can be used in a feedback loop to control that torque. Commanding *zero* torque enables us to selectively turn the normally non-backdriveable barrel rotation actuator into a freely backdriveable mechanism (up to the saturation limits of the motor), useful in handling/configuring the robot, and a critical component of the grip refinement algorithm. We also utilize this torque control when preloading the mechanism to avoid sag due to gravity, as described below.

J.3 Control

A major concern is grip failure due to uncertainty, i.e., falling off the framework. For a planar climber like Shady, this can occur when in-plane pose uncertainty of the connecting gripper is beyond the in-plane misalignment (Figure J-4, left) tolerance of the grip mechanism, or alternately it could be due to a cumulative process where successive grips "walk" the robot normal to the plane of the framework. This walk-off effect may in some cases be mechanically avoidable by designing the gripper to

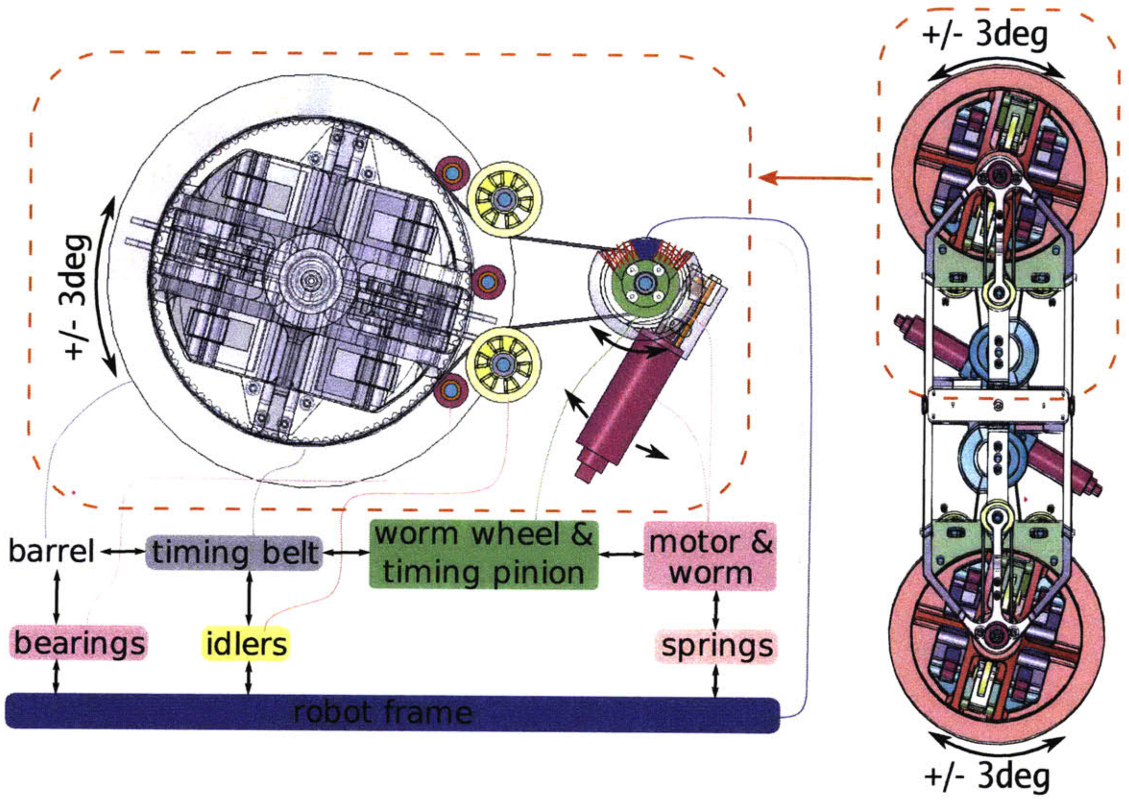


Figure J-3: Shady rotation actuator.
Series-elastic [125] barrel rotation actuator with torsion-mounted motor. Block diagram indicates force/torque transmission among the various elements. Maximum deflection is about $\pm 3^\circ$ of barrel rotation.

partially or fully envelop the cross-section of a framework member, for a form-closed grip. In practice, however, some frameworks effectively present a fully convex cross-section which prevents an enveloping grasp (Figure J-4), and our laboratory window is such a case. We have also noted that there can be some cross-talk between these two mechanisms: when the gripper is not well-centered to start, the rubber grip pads can catch on the vertex of the window bar which is encountered first, preventing the gripper from fully sliding down onto the bar (Figure J-4, right).

Algorithm J.1, GripRefinement, leverages the barrel rotation compliances to minimize these effects, and is executed each time a gripper is closed. Figure 6-3 gives key frames from a video of GripRefinement running on the hardware. This algorithm is proprioceptive, as it uses the barrel rotation springs not only to permit

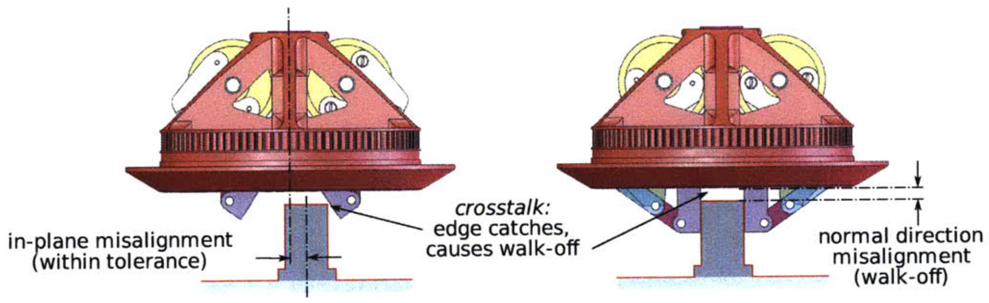


Figure J-4: Shady gripper tests.
In-plane gripper misalignment, normal direction misalignment (walk-off), and the crosstalk effect. Convex cross-section formed by window frame and glass, highlighted, prevents a mechanically enveloping grip.

```
Algorithm J.1: GRIPREFINEMENT($g$)
  Input: left or right gripper $g$
  Let $o$ be the other gripper, and measure its barrel's initial torque $T_o$
  close $g$ completely ▷initial sensing grip
  command $o$'s barrel to torque $T_o$ and $g$'s barrel to zero torque
   ▷the low-level motor controllers can servo torque, position, or power
  command barrel rotation actuators to hold current positions
  re-open $g$ to 50% of the fully open state
  re-close $g$ at 20% speed ▷final grip
  command $g$ to rotate at 50% power for 0.5s ▷validation test
  if $g$ actually rotated more than 1.5° then error validation failed
  command both barrels to zero torque
```

deflection but also to measure that deflection and the torque causing it. At the start of the process, gripper g is currently open and has been commanded to close; the other gripper is already closed, and is initially supporting the robot. The main idea is to perform two separate grips. The first one is essentially a sensing operation, and may be walked-off. The robot then adjusts based on the newly acquired fine position data and re-grips. A final step at the end is to validate the grip by commanding a small torque on the barrel of the just-attached gripper—if it were to actually rotate, it must have closed on thin air.

GRIPREFINEMENT is crucial to reliable operation of the robot—a large proportion of the hundreds of grips we have observed were initially offset in the normal direction (i.e. walked-off) by up to about 8mm. In virtually all cases, GRIPREFINEMENT reduces this walk-off to less than about 2mm, and usually to 0mm.

J.3.1 Avoiding Sag Under Gravity

The plane of our window frame is vertical (as with most windows), so gravity is an important design consideration. The effect is minimal when in double support (both grippers attached) but requires special consideration in single support—unless the robot body is pointing either straight up or straight down, gravity will induce a static torque on the connected barrel rotation with a magnitude dependent on the angle of the body. If unaccounted, this torque would cause Shady to sag due to deflection in the rotation actuator springs. We use the following procedure to preload the springs and avoid this sag:

- Shady is always "launched" (i.e. initialized) in double support on a vertical bar, and the most-recent single support torque T_s is initialized to 0.

- Whenever the robot leaves double support (opens a gripper), the barrel b of the remaining connected gripper is commanded to a preload torque T_p where $T_p = T_s$ if b was the prior single-support barrel, and $T_p = -T_s$ otherwise. Once the gripper is opened, b is commanded to hold position rather than torque, as torque will vary if b is commanded to rotate.

- Whenever the robot enters double support (closes a gripper), the actual torque on the barrel containing the already-closed gripper (i.e. the current single-support barrel) is measured and saved in T_s.

This particular method of accounting T_s does not require explicit knowledge of any absolute orientation relative to the gravity vector, and will thus be immune to uncertainty in measurement or estimation thereof. Under this preload method, virtually all the movements we have observed show no perceptible sag upon gripper opening, independent of orientation.

J.3.2 Path Planning

Since Shady must always be supporting itself with at least one gripper, the reachable shade locations are limited to an offset-band about the window framework. Shady, and its fan, have been scaled appropriately so that this band covers most of the area of the window.

We have developed a path-planning algorithm to determine a short locomotion sequence to any target location in the reachable band. The user interacts with the planner and can specify a target location for Shady by clicking on the screen.

The algorithm first performs a breadth-first search of reachable gripper locations on the framework based on the kinematic structure of the robot, the geometry of the framework, and the robot's starting point. This set of reachable grip points is discrete, but may be infinite even in finite environments due to "spiral" motion sequences about structural nodes which can return the robot to its original grip location plus an arbitrarily small delta. Thus, we put a maximum bound on the search—grip points close to already-found points are pruned. The final set of discovered grip points induces a graph on which Dijkstra's algorithm is run, finding the shortest grip sequence from Shady's current location to the grip point nearest (by Euclidean distance) the desired shade location. Since the shade location may be anywhere in the reachable band, the final step is to rotate Shady's body out over the window to put the shade as close as possible to the requested shade point.

J.4 Experiments and Results

We have performed three types of experiments on the Shady hardware. First, we tested GRIPREFINEMENT with successively greater in-plane angular and linear offsets to determine the maximum tolerable misalignment (Sections J.4.1 and 6.6.2). Second, we placed Shady on the window frame and commanded a cyclic climbing trajectory which exercised the grip, ungrip, and rotate motion primitives in all possible orientations (Section J.4.2). Finally we commanded locomotion sequences out-and-back across the window, similar to that depicted in the lower right of Figure 6-2

(Section J.4.3).

In total, over 1296 individual grip, ungrip, and rotate motions were executed, with only two failures (other than the grips which intentionally failed in the first test set), a reliability rate of over 99.8%. The two faults which did occur were not dangerous: the robot simply stopped and informed the operator that an unexpected state was encountered. In both cases the most likely cause was an intermittent fault in one encoder interface circuit. The command/control software permitted us to investigate this problem over the RF link to the robot, to remotely re-initialize that encoder circuit, and then to resume normal operations, all without requiring physical access to the robot.

J.4.1 Gripper Misalignment Tests

Using the GripRefinement algorithm, we conducted three experiments to determine the maximum misalignments that can be tolerated before a grip fails. In this context, the *connected* barrel is gripped on the window frame, and the *distal* barrel is initially ungripped. Two experiments involved changing the initial orientation of the connected resp. distal barrels prior to gripping, and are described in this section. A third experiment, described in the main text in Section 6.6.2, successively slid the connected barrel along the supporting framework member.

In the first rotation test, we kept the distal gripper aligned to the robot body and we rotated the connected barrel in $1°$ increments, commanding the distal barrel to grip at each step. The maximum tolerated misalignment in this situation is about $3°$, which corresponds to about 2.1cm horizontal displacement at the distal barrel: $(40.4\text{cm})(\sin 3°) \approx 2.1\text{cm}$, giving a healthy 4.2cm lateral uncertainty tolerance band for gripping. The range of compliance is related to the maximum opening width of the gripper, about 7cm, as compared with the 2.5cm window bar: $(7\text{cm} - 2.5\text{cm})/2 = 2.25\text{cm}$.

In the second rotation test, we held the connected barrel fixed and incrementally rotated the distal barrel. Up to $13°$ misalignment is tolerated here, giving a significant $26°$ tolerance band.

These rotation tests were performed with the robot in a vertical configuration; we expect the performance to be similar for other orientations.

J.4.2 Grip and Rotate Primitive Tests (Cyclic Climbing)

We designed a simple cyclic locomotion sequence which takes the robot one full cycle around a structural node of the window frame in 48 grip/ungrip/rotate motions, leaving it where it started. This sequence is designed to test each motion in all of the orientations that would be encountered on our window frame. We ran it for almost 5 full loops around the node, a total of 984 primitive motions over nearly 5 hours. The experiment was interrupted by human intervention only twice due to the two faults described above. No significant normal-direction misalignment (walk-off) was observed at the end of the run, nor was any significant in-plane shift apparent.

J.4.3 Out-and-Back Locomotion Sequences

As a final test, we commanded the robot to repeatedly climb the window from a starting position near the bottom to a point near the top, deploy the fan, retract it, and then to return to the starting position. The overall sequence was similar to that depicted at the lower right in figure 6-2. We ran this experiment over four full cycles, a total of 312 grip/ungrip/rotate primitive motions, with no locomotion faults and no human intervention. Again, there was no significant walk-off after the experiment, however the robot slipped about 4mm in-plane.

Appendix K

The Self-Reconfiguring Robot *Multishady*

Multishady, introduced in Section 5.4, is a self-reconfiguring robot concept based on the Shady climbing robot (Section 6.6 and Appendix J). This appendix gives some additional research context and details for Multishady.

K.1 Background and Motivation

Self-reconfiguring modular robots are systems that are physically connected and capable of actively making different geometric structures [136, 135, 104]. Most research in this field has been focused on homogeneous systems in which all the modules are identical (e.g. [105, 28, 174, 90, 16]). Multishady explores the concept of a *bi-partite* self-assembling robot system consisting of passive structural modules plus active robotic modules. Bi-partite systems are less common in the literature; one prior work is Unsal, Kiliccote, and Khosla's "I-Cubes" system [156].

The passive modules in Multishady are structural bars which may either be fixed in the world or free to move individually. The mobile active modules have the same kinematics as the Shady climbing robot (Figure 6-1), and use their rotating grippers to pick up or climb on the passive modules, organize and hold them in a desired shape, or actively move them for self-assembly, self-reconfiguration, or self-repair purposes.

The passive modules can be passed around by the active modules and coordinated to form the skeleton of a large class of structure and linkage geometries. For example, Figure K-1 show a simulation of the self-assembly of a tower, and Figure 5-11 shows the tower actively deforming.

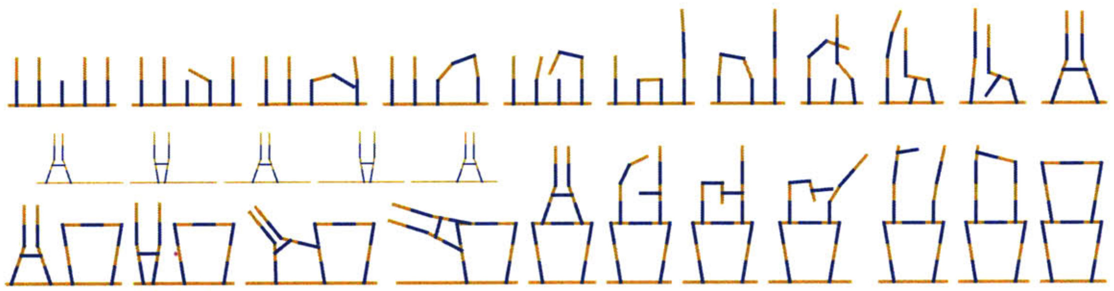


Figure K-1: Self-assembly of a tower with Multishady.
Snapshots from a manually-planned simulation showing the construction of a two-legged walking structure (top row, rightmost) starting from a "packed" configuration of active and passive modules (top row, leftmost); a walker locomoting on a truss segment (middle row); and a walker performing concave and convex transitions, walking up a tower, and reconfiguring into a new structural block of the tower (bottom row).

A long-term application of self-reconfiguring systems like Multishady is in-space structure construction. The modules will pack tightly in a spacecraft, yet they will be able to self-assemble, self-reconfigure, self-repair, and adapt their collective morphology to perform a variety of tasks—some known in advance (pre-launch) and some dynamic (post-launch). And the system can act either as a tool to assemble/repair/service other space structures, or as an active orbiting structure itself. Other applications include terrestrial construction of increasingly more capable structures such as dynamic scaffolds and movable towers for construction tasks.

K.2 Practical Considerations

So far, Multishady has mainly been explored in concept and in kinematic simulation. Though the idea is based on our Shady climbing robot, that hardware is scaled for climbing on a particular structure, and is likely not appropriate for use directly in a system like Multishady. We expect that hardware with the same kinematics could be designed and built in a way that is applicable to such cooperative applications. For a system operating under terrestrial gravity, necessary changes include making the module significantly smaller, increasing its power-to-mass ratio, changing the gripper design, and possibly designing the system to distribute power through the structure. Our research group has already begun developing hardware that addresses some of these issues [42].

The Multishady concept presented in this thesis is a 2D system, but it extends directly to a 3D version by adding a rotational DoF in the middle of the module [175].

K.3 Research Context

Multishady is related to prior work in not only in self-reconfiguring robots, but also hyper-redundant robots and variable-geometry trusses.

K.3.1 Self-Reconfiguring Robots

Of all the self-reconfiguring modular robots which have been previously reported, Multishady is most closely allied with systems based on rotary DOF and mechanical connection mechanisms, for example: Murata, Kurokawa, et al's "3D Fracta" [106]; Kotay and Rus' "Molecule" [88, 87, 86]; Unsal, Kiliccote, and Khosla's bi-partite "I-Cubes" [156] system; Duff, Yim, et al's PolyBot [47]; and Lund, Beck, Dalgaard, Støy et al's ATRON [96, 149].

A major difference is that in Multishady only some modules contain active DOF—the rest serve as passive structural elements. In contrast, all of the above referenced systems are either homogeneous (all modules identical and actuated) or are heterogeneous but still require actuation in all modules.

K.3.2 Hyper-Redundant Robots

Research in the field of hyper-redundant robots has mainly explored non-reconfiguring systems with high DOF and fixed kinematic topology, typically open chains. Both planar systems—e.g. Burdick and Chirikjian's "snakey" (which is also a variable geometry truss, see below) [29, 30]; Greenfield, Rizzi, Choset et al's modular snake [57]—and full spatial mechanisms, e.g. Suthakorn and Chirikjian's binary-actuation manipulator [151]; Wolf, Choset, et al's "Schmoopie" [170]—have been explored. The planar systems typically have one (effective) kinematic DOF per link, and the spatial systems may have two or more. Sometimes the links are internally parallel mechanisms, an arrangement which has been called "hybrid serial-parallel" [61, 153, 151].

K.3.3 Variable Geometry Truss and Truss Climbing Robots

Variable geometry trusses (VGTs), can be viewed as a generalization of the serial-chain hyper-redundant systems to more general kinematic topologies. Both fixed-topology systems like the NASA/DOE "SERS DM" [169] and manually-reconfigurable systems—notably Hamlin, Sanderson, et al's TETROBOT [61]—have been considered. Also related are robotic systems which assemble static trusses, for example, Everest, Shen, et al's SOLAR [49], and Howe and Gibson's "Trigon" system [72]. Such self-assembling and self-reconfiguring truss systems are a promising direction for robotic assembly of large structures in space—for example, see Doggett's overview of automatic structural assembly for NASA [45].

Appendix L

The Stair-Stepping Robot *Steppy*

Steppy, introduced in Section 6.7, is an experiment in compliant/proprioceptive stair-stepping with 18-DoF mini-humanoid (Figure 6-8). This appendix summarizes the research context of Steppy with respect to other work in humanoid stair-stepping, and gives additional details of the robot's hardware, software, control, and testing.

L.1 Research Context

Starting with the Honda Asimo [65], stair climbing has been reported in several recent humanoids, both in simulation and physical experiment [178, 140, 24]. Most of these systems do not use joint proprioception and are based instead on contact, inertial, and limb strain sensors (load cells). Reliability and tolerance to uncertainty are not typically quantified, though Gutmann et all report implementing climbing steps with apparently significant uncertainty using vision on a mini-humanoid in [59].

Both Zheng and Shen [180] and Chew, Pratt, and Pratt [25, 129] used joint compliance and joint sensing to enable bipeds to navigate terrain of uncertain slope. Those works both augmented the joint sensing with foot-sole contact sensors, and considered gentle continuous slopes vs. a discrete step.

L.1.1 Hardware and Software

Steppy (Figure 6-8) is based on the low-cost (~$1k) Robotis Bioloid system, is about 36cm tall, and weighs about 2.0kg. A few modifications were made to the stock hardware:

- some components were added and re-arranged to stiffen the hip yaw rotation
- the soles (but not the edges) of the feet were coated with several layers of a spray-on silicone conformal coating to increase stiction on the climbing platforms
- a Bluetooth® module was added.

External fans were also added which cool the robot at all times, reducing the maximum temperature of the servos by about 30°C. Without these fans (and careful temperature monitoring) several servos reached temperatures above 90°C and were destroyed. The servos do not offer much torque or power margin relative to the loads that can sometimes be imposed during locomotion, which can complicate experiments by limiting the actuatable motions and poses to a subset of even those that are statically stable.

Steppy can run un-tethered, though the experiments reported below were performed with a power tether since the batteries last under an hour.

I implemented custom firmware for the on-board controller of the Bioloid to exchange afferent sensor data and efferent commands for all DoF to/from a remote-brain workstation in soft real time at about 10Hz. The remote brain software has an interactive GUI for manipulating the robot and for editing motion sequences, and can also autonomously execute non-linear sequences as described below in Section L.2.

The Bioloid hardware was chosen primarily because, compared to other low-cost mini humanoids currently available, the servos in the Bioloid system support a much richer real-time command and control interface. All 18 servos are Robotis model AX-12, and accept a range of data including position commands, slew rate limits, and position loop proportional gains over a 1Mbps serial bus. The servos can also report measurements including present position, speed, temperature, and a "load" value, though it is undocumented how this last measurement is derived. The actual update rates for the various data items are also not specified, though it appears they are significantly faster than the 10Hz remote brain update rate.

Controlling Compliance

The Robotis documentation implies that the internal servo control loop contains proportional and derivative, but not integral, terms. I verified this, and calibrated the proportional gain command value to natural units of static rotational stiffness at the servo output, in an initial experiment on an isolated servo (Figure L-1). For this test the servo was loaded with a known mass along a measured lever arm. Various P-gain values (which Robotis inverts and calls "compliance slope") were then commanded, and the resulting deflection was measured vs. the commanded servo position. This experiment also revealed that the compliance slope register value can change the effective static stiffness of the servo by a factor of about 8.

I manually fit the following approximate model to the data, to predict the compliance as a function of the firmware command:

$$\textit{stiffness} = (2 - 0.55(\log(\textit{command})))^3 \left[\frac{\mathrm{kg}\cdot\mathrm{cm}}{\mathrm{degree}}\right]. \tag{L.1}$$

This is used in the quasistatic simulation in Section 6.7.2.

L.1.2 Adjustable Step Setup

The experimental setup, shown in Figure 6-8, includes two 45×45cm acrylic platforms supported in cantilever by an aluminum structure, and is rigid enough to only deflect a few millimeters under the weight of the robot. The foreground platform in the figure is mounted to a rotary actuator which can change its pitch, and the background platform is mounted to a series stack of two actuators, one controlling roll and the other the step height. In the experiments reported here the rotary actuators were used only to level the platforms, though angle variations may be incorporated in future stepping experiments. All three actuators are highly rigid (they were designed

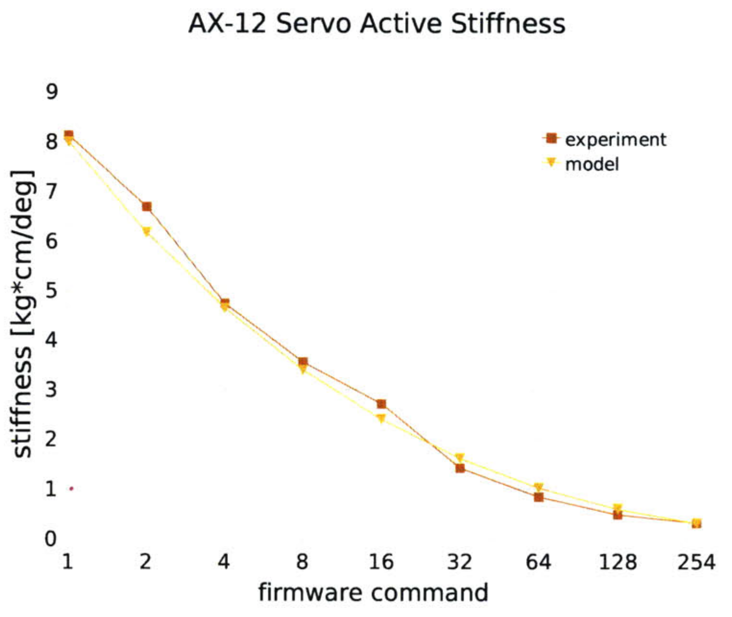


Figure L-1: Testing the compliance of the AX-12 servo.
This plot shows experimentally measured static rotational stiffness at the AX-12 servo output as a function of the "compliance slope" firmware command, which corresponds to the inverse of the position loop proportional gain. Also shown is a model (Eq. L.1) that was manually fit to the data.

for machine tool axes) and may be operated either manually or by stepper motors. The complete apparatus can be assembled in a day, costs under $2k, and requires only off-the-shelf parts and minimal machining.

L.2 Experiments and Results

In general, the Steppy experiment was designed to minimize cost and complexity while still achieving an interesting level of uncertainty tolerance using proprioception in humanoid step climbing. Overall, the experimental procedure was:

1. manually design statically stable *canonical motion sequences* (Section L.2.1) for climbing 3, 2, and 1cm steps; the prefixes of these sequences are all identical until first contact with the upper platform

2. design a *sensing sequence* (Section L.2.2), starting from the pose at the end of

the common prefix of the canonical sequences, which progressively lowers the right heel while monitoring the deflection of the right hip pitch servo, whose position loop proportional gain has temporarily been set to minimum

3. abort the sensing sequence once the step height has been determined closest to 3, 2, or 1cm, and continue with the suffix of the corresponding canonical sequence.

The 1, 2, and 3cm levels correspond to roughly 1/4, 1/2, and 3/4 the total 0-35mm range of step height capability. Without loss of generality absolute step sizes will be used henceforth.

For Steppy, each pose in a motion sequence is not just a set of actuator goal position keyframes, but also includes position loop proportional gain commands and slew rate limits for all actuators. Execution may be non-linear, with sequencing logic that makes decisions based on proprioceptive joint position sensor data. The specific sequencing logic for the step task is given as Algorithm L.1, STEPPYSTEP.

Algorithm L.1: STEPPYSTEP

```
execute the common prefix sequence
 ▷right foot now suspended above upper platform
lower right hip pitch servo $p$ to minimum stiffness
rotate $p$ downwards a by a small fixed increment
 ▷heel collides with ~3cm or ~2cm step
proprioceptively measure actual angle $\theta$ at $p$
if $\theta < \theta_{3cm\text{-}threshold}$ then finish with 3cm suffix sequence
else
    rotate $p$ downwards by additional small fixed increment
     ▷heel collides with ~2cm or ~1cm step
    re-measure actual angle $\theta$ at $p$
    if $\theta < \theta_{2cm\text{-}threshold}$ then finish with 2cm suffix sequence
    else finish with 1cm suffix sequence
end
```

L.2.1 Canonical Sequences

The three canonical sequences were designed to maintain about twice the stiffness in the support leg relative to the suspended leg. However, the robot is never truly rigid—even at maximum stiffness the joints still deflect. This, combined with the broad flat feet of the robot and the overall design of the sequences, enables the canonical sequences to passively accommodate up to about ±10mm of variation in step height vs. the design height for the sequence. For example, the 2cm sequence, unmodified and uncorrected, usually succeeds in climbing steps in the range 1-3cm. Thus, while no individual canonical sequence works for all heights in the 0-3.5cm range, it suffices to select one of the three.

L.2.2 Sensing Sequence

For the sensing sequence, I first considered trying to lower the right foot in parallel with the upper platform. This is slightly tricky given the kinematic reachability of the robot (with the added constraint of static stability), and moreover, it highlights an interesting challenge: once the robot enters double support with both feet flat on the ground, it is an overactuated kinematic cycle that is closed through the rigid ground contact [177]. This is not necessarily a problem—one of the strong points of the low-impedance proprioceptive approach is that the joints throughout such a closed chain will deflect according to their compliances and the configuration of the chain. Through a forward kinematic model, measurements of the deflections can yield an estimate of the unknown geometry of the chain-closing link, i.e. the step itself.

In this case I opted to instead explore a more minimal strategy. The right leg is straightened, and the active stiffness of the right hip pitch servo is set to minimum (making it a factor of 3 lower than the rest of the joints in that leg and a factor of 6 lower than the other leg). That servo is then rotated downwards in two incremental probing motions, resulting in an eventual collision of the left corner of the right heel with the upper platform. The corner of the heel establishes a relatively low-friction plastic-on-plastic sliding contact with the upper platform.

The first downward rotation of the right leg during the sensing sequence is intended to enable discrimination of a 3cm step, and the second rotation is invoked as needed to discriminate a 2cm step vs any lower step, for which the 1cm canonical sequence is invoked. The right hip pitch servo deflection sensor readings were manually analyzed for three preliminary trials in each configuration. They repeatably differed by around 10 counts for neighboring configurations, and this discriminating difference was used to formulate the thresholds in STEPPYSTEP.

L.2.3 Experimental Trials

10 autonomous trials were run for each step height at 5mm increments in the closed interval 0-3.5cm (the 3cm canonical sequence was known not to perform well above 3.5cm), with the results for all 80 trials shown in table L.2.3. Each trial took about 90s. The robot terminated in stable double support on the upper platform in all but seven of the 80 trials. The failures can be attributed to at least two causes: the 3cm canonical sequence is not always successful at the 3.5cm height, and the 1cm canonical sequence is similarly not always successful at the 1.5cm height.

In most trials the detected step height was correct, in that it was a canonical height closest to the actual height. This was not always the case, however: in 10 trials (marked * in the table) the height was detected at 3cm where it was actually closer to either 1 or 2cm. This may have been due to an inadvertent perturbation of the angle of the starting platform during these trials. These sensing errors did not actually cause overall failure because the 3cm sequence nevertheless succeeded at climbing these lower heights (it does not work well below 1cm though). For this initial experiment the only actively handled uncertainty is the step height, so it is not surprising that variations in other parameters can cause problems.

Actual Height	**Number of Successful/(Failing) Trials with Detected Height of**		
	3cm	**2cm**	**1cm**
0.0cm	0	0	10
0.5cm	0	0	10
1.0cm	0	0	10
1.5cm	3*	4	0 (3)
2.0cm	7*	3	0
2.5cm	10	0	0
3.0cm	10	0	0
3.5cm	6 (4)	0	0

Table L.1: Steppy step-climbing experimental data.
10 autonomous trials were performed at each step height at 5mm increments in the range 0-3.5cm (total 80 trials). For each trial the number of successes is listed, along with the number of failures (falls), if any, in parenthesis. The vertical columns indicate the step height as detected by the robot during the run; trials where this was not actually nearest the actual step height are marked *.

Bibliography

[1] Eberhard Abele, Stefan Rothenbücher, and Matthias Weigold. Cartesian compliance model for industrial robots using virtual joints. *Production Engineering Research and Development*, 2:339–343, 2008.

[2] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. A. K. Peters, Ltd., 2nd edition, 2002.

[3] Tarek Alameldin, Norman Badler, and Tarek Sobh. A hybrid system for computing reachable workspaces for redundant manipulators. In *SPIE Conference on Machine Vision Systems Integration*, pages 112–120, 1990.

[4] M. Almonacid, R. J. Saltarén, R. Aracil, and O. Reinoso. Motion planning of a climbing parallel robot. *IEEE Transactions on Robotics and Automation*, 19(3):485–489, 2003.

[5] Greg Aloupis, Sébastien Collette, Erik D. Demaine, Stefan Langerman, Vera Sacristán, and Stefanie Wuhrer. Reconfiguration of cube-style modular robots using $O(\log n)$ parallel moves. In *Proceedings of the 19th Annual International Symposium on Algorithms and Computation (ISAAC 2008)*, pages 342–353, 2008.

[6] Hisanori Amano, Koichi Osuka, and Tzyh-Jong Tarn. Development of vertically moving robot with gripping handrails for fire fighting. In *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 661–667, Maui, HI, 2001.

[7] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[8] Hashem Ashrafiuon and Kiran Sanka. Development of virtual link method for the solution of hyper-redundant spatial robots. *Journal of Robotic Systems*, 13:371–378, 1996.

[9] Paolo Baerlocher. *Inverse Kinematics Techniques for the Interactive Posture Control of Articulated Figures.* PhD thesis, Ecole Polytechnique Federale de Lausanne, 2001.

[10] Paolo Baerlocher and Ronan Boulic. Parametrization and range of motion of the ball-and-socket joint. In *Proceedings of the IFIP TC5/WG5.10 DEFORM '2000 Workshop and AVATARS '2000 Workshop on Deformable Avatars*, pages 180–190. Kluwer, B.V., 2001.

[11] Paolo Baerlocher and Ronan Boulic. An inverse kinematics architecture enforcing an arbitrary number of strict priority levels. *The Visual Computer*, 20:402–417, 2004.

[12] David Bau. Faster SVD for matrices with small m/n. Technical Report TR 94-1414, Cornell University Department of Computer Science, March 1994.

[13] Calin Belta and Vijay Kumar. New metrics for rigid body motion interpolation. In *Proceedings of A Symposium Commemorating the Legacy, Works, and Life of Sir Robert Stawell Ball Upon the 100th Anniversary of A Treatise on the Theory of Screws*, 2000.

[14] Eric A. Bier. *Snap-Dragging: Interactive Geometric Design in Two and Three Dimensions.* PhD thesis, EECS Department, University of California, Berkeley, May 1988.

[15] Yvan Bourquin. Personal communication, January 2007. Software developer, Webots, Cyberbotics Ltd.

[16] David Brandt, David Johan Christensen, and Henrik Hautop Lund. ATRON robots: Versatility from self-reconfigurable modules. In *IEEE International Conference on Mechatronics and Automation*, pages 26–32, 2007.

[17] Beat Brüderlin and Dieter Roller. *Geometric Constraint Solving and Applications.* Springer, 1998.

[18] Herman Bruyninckx. Open RObot COntrol Software (OROCOS). `http://www.orocos.org/`.

[19] Herman Bruyninckx. *Kinematic Models for Robot Compliant Motion with Identification of Uncertanties.* PhD thesis, Katholieke Universiteit Leuven, 1995.

[20] Samuel R. Buss. Introduction to inverse kinematics with Jacobian transpose, pseudoinverse, and damped least squares methods. Available on the web at `http://www.math.ucsd.edu/~sbuss/ResearchWeb/ikmethods/iksurvey.pdf`, April 2004.

[21] Jason Cantarella, Erik D. Demaine, Hayley Iben, and James O'Brien. An energy-driven approach to linkage unfolding. In *Proceedings of the 20th Annual ACM Symposium on Computational Geometry (SoCG 2004)*, pages 134–143, 2004.

[22] Aranzazu Casal. *Reconfiguration planning for modular self-reconfigurable robots.* PhD thesis, Stanford University, 2002.

[23] Tony F. Chan. An improved algorithm for computing the singular value decomposition. *ACM Transactions on Mathematical Software*, 8(1):72–83, March 1982.

[24] Joel Chestnutt, James Kuffner, Koichi Nishiwaki, and Satoshi Kagami. Planning biped navigation strategies in complex environments. In *Proceedings of IEEE International Conference on Humanoid Robotics*, 2003.

[25] Chee-Meng Chew, J. Pratt, and G. Pratt. Blind walking of a planar bipedal robot on sloped terrain. In *Proceedings of IEEE ICRA*, volume 1, pages 381–386, 1999.

[26] Pasquale Chiacchio, Stefano Chiaverini, Lorenzo Sciavicco, and Bruno Siciliano. Closed-loop inverse kinematics schemes for constrained redundant manipulators with task space augmentation and task priority strategy. *The International Journal of Robotics Research*, 10(4):410–425, 1991.

[27] G. Chirikjian, A. Pamecha, and I. Ebert-Uphoff. Evaluating efficiency of self-reconfiguration in a class of modular robots. *Journal of Robotics Systems*, 13(5):317–338, 1996.

[28] Gregory S. Chirikjian. Kinematics of a metamorphic robotic system. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, volume 1, pages 449–455, 1994.

[29] Gregory S. Chirikjian and Joel W. Burdick. A hyper-redundant manipulator. *IEEE Robotics & Automation Magazine*, pages 22–29, December 1994.

[30] Gregory S. Chirikjian and Joel W. Burdick. The kinematics of hyper-redundant robot locomotion. *IEEE Transactions on Robotics and Automation*, 11(6):781–793, 1995.

[31] Namik Ciblak and Harvey Lipkin. Synthesis of stiffnesses by springs. In *Proceedings of ASME Design Engineering Technical Conferences*, 1998.

[32] Namik Ciblak and Harvey Lipkin. Design and analysis of remote center of compliance structures. *Journal of Robotic Systems*, 20(8):415–427, 2003.

[33] C. L. Collins, J. M. McCarthy, A. Perez, and H. Su. The structure of an extensible Java applet for spatial linkage synthesis. *Journal of Computing and Information Science in Engineering*, pages 45–49, March 2002.

[34] Robert Connelly and Erik D. Demaine. Geometry and topology of polygonal linkages. In Jacob E. Goodman and Joseph ORourke, editors, *CRC Handbook of Discrete and Computational Geometry*, chapter 9, pages 197–218. CRC Press, 2nd edition, 2004.

[35] John J. Craig and Marc H. Raibert. A systematic method of hybrid position/force control of a manipulator. In *Computer Software and Applications Conference*, pages 446–451, 1979.

[36] Ernest Davis. Approximation and abstraction in solid object kinematics. Technical Report TR1995-706, New York University, Department of Computer Science, 1995.

[37] Erik D. Demaine and Joseph O'Rourke. *Geometric Folding Algorithms: Linkages, Origami, Polyhedra.* Cambridge University Press, 2008.

[38] J. Denavit and R. S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *ASME Journal of Applied Mechanics*, 23:215–221, 1955.

[39] Arati S. Deo and Ian D. Walker. Robot subtask performance with singularity robustness using optimal damped least squares. In *IEEE International Conference on Robotics and Automation*, pages 434–441, 1992.

[40] Rajiv S. Desai and Richard A. Volz. Identification and verification of termination conditions in fine motion in presence of sensor errors and geometric uncertainties. In *Proceedings of IEEE ICRA*, volume 2, pages 800–807, 1989.

[41] Carrick Detweiler, Marsette Vona, Keith Kotay, and Daniela Rus. Hierarchical control for self-assembling mobile trusses with passive and active links. In *IEEE International Conference on Robotics and Automation*, pages 1483–1490, 2006.

[42] Carrick Detweiler, Marsette Vona, Yeoreum Yoon, Seung kook Yun, and Daniela Rus. Self-assembling mobile linkages. *IEEE Robotics and Automation Magazine*, 14:45–55, 2007.

[43] Antonio Diaz-Calderon, Issa A. D. Nesnas, Hari Das Nayar, and Won S. Kim. Towards a unified representation of mechanisms for robotic control software. *International Journal of Advanced Robotic Systems*, 3(1):061–066, 2006.

[44] L. Dobrjanskyj and F. Freudenstein. Some applications of graph theory to the structural analysis of mechanisms. *ASME Journal of Engineering for Industry*, pages 153–158, February 1967.

[45] William Doggett. Robotic assembly of truss structures for space systems and future research plans. In *IEEE Aerospace Conference Proceedings*, March 2002.

[46] Samuel Hunt Drake. *Using Compliance in Lieu of Sensory Feedback for Automatic Assembly.* PhD thesis, Massachusetts Institute of Technology, 1977.

[47] David G. Duff, Mark Yim, and Kimon Roufas. Evolution of PolyBot: a modular reconfigurable robot. In *Proceedings of the Harmonic Drive International Symposium*, Nagano, Japan, November 2001.

[48] Michael E. Erdmann and Matthew T. Mason. An exploration of sensorless manipulation. *IEEE Journal of Robotics and Automation*, 4(4):369–379, August 1988.

[49] Jacob Everist, Kasra Mogharei, Harshit Suri, Nadeesha Ranasinghe, Berok Khoshnevis, Peter Will, and Wei-Min Shen. A system for in-space assembly. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2356–2361, Sendai, Japan, 2004.

[50] Roy Featherstone and David Orin. Robot dynamics: Equations and algorithms. In *IEEE International Conference on Robotics and Automation*, pages 826–834, 2000.

[51] Robert Fitch and Zack Butler. Million module march: Scalable locomotion for large self-reconfiguring robots. *International Journal of Robotics Research*, 27(3/4):331–343, 2008.

[52] Lorenzo Flückiger. A robot interface using virtual reality and automatic kinematics generator. In *International Symposium on Robotics*, pages 123–126, April 1998.

[53] Toshio Fukuda and Seiya Nakagawa. Dynamically reconfigurable robotic system. In *IEEE International Conference on Robotics and Automation*, pages 1581–1586, 1988.

[54] Michael L. Gleicher. *A Differential Approach to Graphical Interaction.* PhD thesis, Carnegie Mellon University, School of Computer Science, 1994. CMU-CS-94-217.

[55] F. Sebastian Grassia. Practical parameterization of rotations using the exponential map. *Journal of Graphics Tools*, 3(3):29–48, 1998.

[56] Jack E. Graver. *Counting on Frameworks.* Cambridge University Press, 2001.

[57] Aaron Greenfield, Alfred A. Rizzi, and Howie Choset. Dynamic ambiguities in frictional rigid-body systems with application to climbing via bracing. In *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 1959–1964, Barcelona, Spain, April 2005.

[58] You Liang Gu. An exploration of orientation representation by lie algebra for robotic applications. *IEEE Transactions on Systems, Man, and Cybernetics*, 20:243–248, 1990.

[59] Jens-Steffen Gutmann, Masaki Fukuchi, and Masahiro Fujita. Stair climbing for humaniod robots using stereo vision. In *Proceedings of IEEE/RSJ IROS*, pages 1407–1413, 2004.

[60] Ronald Van Ham, Bram Vanderborght, Michaël Van Damme, Björn Verrelst, and Dirk Lefeber. MACCEPA: the mechanically adjustable compliance and controllable equilibrium position actuator for 'controlled passive walking'. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, pages 2195–2200, May 2006.

[61] Gregory J. Hamlin and Arthur C. Sanderson. Tetrobot: A modular approach to parallel robotics. *IEEE Robotics & Automation Magazine*, pages 42–49, March 1997.

[62] Kris Hauser, Timothy Bretl, Jean-Claude Latombe, and Brian Wilcox. Motion planning for a six-legged lunar robot. In *Proceedings of WAFR*, pages 301–316, 2006.

[63] J. M. Hervé. Analyse structurelle des mécanismes par groupe des déplacements. *Mechanism and Machine Theory*, 13:437–450, 1978. in French.

[64] Matt Heverly. Personal communication, May 2008.

[65] Kazuo Hirai, Masato Hirose, Yuji Haikawa, and Toru Takenaka. The development of honda humanoid robot. In *Proceedings of IEEE ICRA*, 1998.

[66] Christoph M. Hoffmann. D-Cubed's Dimensional Constraint Manager. *Journal of Computing and Information Science in Engineering*, 1:100–101, 2001.

[67] Christoph M. Hoffmann and Robert Joan-Arinyo. A brief on constraint solving. *Computer Aided Design and Applications*, 2(5):655–663, April 2005.

[68] Neville Hogan. Impedance control: An approach to manipulation: Part I—theory. *Journal of Dynamic Systems, Measurement, and Control*, 107:1–7, 1985.

[69] Neville Hogan and Stephen P. Buerger. Impedance and interaction control. In Thomas R. Kurfess, editor, *CRC Robotics and Automation Handbook*, chapter 19. CRC Press, 2005.

[70] John Hopcroft, Deborah Joseph, and Sue Whitesides. Movement problems for 2-dimensional linkages. *SIAM Journal of Computing*, 14:315–333, 1985.

[71] Berthold K.P. Horn. Some notes on unit quaternions and rotation, 2001. `http://people.csail.mit.edu/bkph/articles/Quaternions.pdf`.

[72] A. Scott Howe and Ian Gibson. Trigon robotic pairs. In *AIAA Space 2006 Conference*, 2006.

[73] K. H. Hunt. *Kinematic Geometry of Mechanisms.* Clarendon Press, 1978.

[74] Ben Iannotta. Creating robots for space repairs. *Aerospace America*, pages 36–40, May 2005.

[75] Oleg Ivlev and Axel Gräser. An analytical method for the inverse kinematics of redundant robots. In *Proceedings of 3rd ECPD Int. Conf. on Advanced Robots, Intelligent Automation and Active Systems*, pages 416–421, 1997.

[76] Oleg Ivlev and Axel Gräser. Resolving redundancy of series kinematic chains through imaginary links. In *Proceedings of CESA '98 IMACS Multiconference, Computational Engineering in Systems Applications*, pages 477–482, 1998.

[77] Christophe Jermann, Giles Trombettoni, Bertrand Neveu, and Pascal Mathis. Decomposition of geometric constriant sytems: a survey. *International Journal of Computational Geometry & Applications*, 23:379–414, 2006.

[78] R. Joan-Arinyo, A. Soto-Riera, S. Vila-Marta, and J. Vilaplana-Pasto. Transforming an under-constrained geometric constraint problem into a well-constrained one. In *ACM Solid Modeling 2003*, 2003.

[79] Denis Jordan and Michael Steiner. Configuration spaces of mechanical linkages. *Discrete Computational Geometry*, 22:297–315, 1999.

[80] Alfred Bray Kempe. On a general method of describing plane curves of the nth degree by linkwork. *Proceedings of the London Mathematical Society*, 7:213–216, 1876.

[81] W. Khalil and J. Kleinfinger. A new geometric notation for open and closed-loop robots. In *IEEE International Conference on Robotics and Automation*, volume 3, pages 1174–1179, 1986.

[82] Henry C. King. Planar linkages and algebraic sets. *Turkish Journal of Mathematics*, 23:33–56, 1999.

[83] Charles A. Klein and Ching-Hsiang Huang. Review of pseudoinverse control for use with kinematically redundant manipulators. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(3):245–250, 1983.

[84] Evangelos Kokkevis. Practical physics for articulated characters. In *Game Developers Conference*, 2004.

[85] Ulrich Kortenkamp. *Foundations of Dynamic Geometry.* PhD thesis, Swiss Federal Institute of Technology Zurich, 1999.

[86] Keith Kotay. *Self-Reconfiguring Robots: Designs, Algorithms, and Applications.* PhD thesis, Dartmouth College, December 2003.

[87] Keith Kotay, Daniela Rus, Marsette Vona, and Craig McGray. The self-reconfiguring robotic molecule. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 424–431, 1998.

[88] Keith Kotay, Daniela Rus, Marsette Vona, and Craig McGray. The self-reconfiguring robotic molecule: Design and control algorithms. In *Workshop on the Algorithmic Foundations of Robotics*, pages 375–386, 1998.

[89] Glenn A. Kramer. *Solving Geometric Constraint Systems.* MIT Press, 1992.

[90] Haruhisa Kurokawa, Akiya Kamimura, Eiichi Yoshida, Kohji Tomita, Shigeru Kokaji, and Satoshi Murata. M-TRAN II: metamorphosis from a four-legged walker to a caterpillar. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2454–2459, Las Vegas, Nevada, October 2003.

[91] Roberto Lampariello, Satoko Abiko, and Gerd Hirzinger. Dynamics modeling of structure-varying kinematic chains for free-flying robots. In *IEEE International Conference on Robotics and Automation*, pages 1207–1212, 2008.

[92] Tine Lefebvre, Herman Bruyninckx, and Joris De Schutter. Task planning with active sensing for autonomous compliant motion. *IJRR*, 24(1):61–81, 2005.

[93] Tine Lefebvre, Jing Xiao, Herman Bruyninckx, and Gudrun De Gersem. Active compliant motion: a survey. *Advanced Robotics*, 19(5):479–499, 2005.

[94] Alain Liégeois. Automatic supervisory control of the configuration and behavior of multibody mechanisms. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC–7(12):868–871, December 1977.

[95] Robert Light and David Gossard. Modification of geometric models through variational geometry. *Computer-Aided Design*, 14:209–214, 1982.

[96] Henrik Hautop Lund, Richard Beck, and Lars Dalgaard. ATRON hardware modules for self-reconfigurable robotics. In Sugisaka and Takaga, editor, *Proceedings of 10th International Symposium on Artificial Life and Robotics (AROB'10), ISAROB*, Oita, 2005.

[97] Anthony A. Maciejewski and Charles A. Klein. Numerical filtering for the operation of robotic manipulators through kinematically singular configurations. *Journal of Robotic Systems*, 5:527–552, 1988.

[98] Matthew T. Mason. *Mechanics of Robotic Manipulation.* MIT Press, 2001.

[99] Matthew T. Mason and J. Kenneth Salisbury, Jr. *Robot hands and the mechanics of manipulation.* MIT Press, 1985.

[100] Matthew Thomas Mason. Compliance and force control for computer controlled manipulators. Master's thesis, Massachusetts Institute of Technology, April 1979.

[101] Tad McGeer. Passive dynamic walking. *The International Journal of Robotics Research*, 9(2):62–82, 1990.

[102] Wim Meeussen, Joris De Schutter, Herman Bruyninckx andJing Xiao, and Ernesto Staffetti. Integration of planning and execution in force controlled compliant motion. In *Proceedings of IROS*, pages 2550–2555, 2005.

[103] David S. Mittman, Jeffrey S. Norris, Mark W. Powell, Recaredo J. Torres, Christopher McQuin, and Marsette A. Vona. Lessons Learned from All-Terrain Hex-Limbed Extra-Terrestrial Explorer Robot Field Test Operations at Moses Lake Sand Dunes, Washington. In *Proceedings of AIAA Space*, September 2008.

[104] Mark Moll and Daniela Rus (eds.). Special issue on self-reconfiguring modular robots. *International Journal of Robotics Research*, 27(3/4), March/April 2008.

[105] Satoshi Murata, Haruhisa Kurokawa, and Shigeru Kokaji. Self-assembling machine. In *IEEE Conference on Robotics and Automation*, 1994.

[106] Satoshi Murata, Haruhisa Kurokawa, Eiichi Yoshida, Kohji Tomita, and Shigeru Kokaji. A 3-D self-reconfigurable structure. In *Proceedings of the 1998 IEEE International Conference on Robotics and Automation*, pages 432–439, Leeuven, Belgium, May 1998.

[107] Y. Nakamura and H. Hanafusa. Inverse kinematics solutions with singularity robustness for robot manipulator control. *Journal of Dynamic Systems, Measurement, and Control*, 108:163–171, 1986.

[108] Yoshihiko Nakamura and Katsu Yamane. Dynamics computation of structure-varying kinematic chains and its application to human figures. *IEEE Transactions on Robotics and Automation*, 16(2):124–134, 2000.

[109] Shinichiro Nakaoka, Atsushi Nakazawa, Kazuhito Yokoi, Hirohisa Hirukawa, and Katsushi Ikeuchi. Generating whole body motions for a biped humanoid robot from captured human dances. In *Proceedings of IEEE ICRA*, pages 3905–3910, 2003.

[110] Michael Nechyba and Yangsheng Xu. Human-robot cooperation in space: SM2 for new space station structure. *IEEE Robotics and Automation Magazine*, 2(4):4–11, December 1995.

[111] netlib-java library, 2009. `http://code.google.com/p/netlib-java/`.

[112] An Nguyen, Leonidas J. Guibas, and Mark Yim. Controlled module density helps reconfiguration planning. In *Proceedings of WAFR 2000: New Directions in Algorithmic and Computational Robotics*, pages 23–36, 2001.

[113] Dohmen Noort and Willem F. Bronsvoort. Solving over- and underconstrained geometric models. In B. Brüderlin and D Roller, editors, *Geometric Constraint Solving and Applications*, pages 107–127. Springer, 1998.

[114] Michael A. O'Connor and Vijay Srinivasan. Connected lie and symmetry subgroups of the rigid motions: Foundations and classification. Technical Report RC 20512, IBM Research Division, T.J. Watson Research Center, July 1996.

[115] Denny Oetomo, Marcelo H. Ang, and Tao Ming Lim. Singularity robust manipulator control using virtual joints. In *IEEE Ingternational Conference on Robotics and Automation*, pages 2418–2423, 2002.

[116] David E. Orin and William W. Schrader. Efficient compuation of the Jacobian for robot manipulators. *The International Journal of Robotics Research*, 3:66–75, 1984.

[117] J.C. Owen. Algebraic solution for geometry from dimensional constraints. In *Proceedings of the first ACM symposium on Solid modeling foundations and CAD/CAM applications*, pages 397–407, 1991.

[118] Pack, Christopher, and Kawamura. A rubbertuator-based structure-climbing inspection robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, pages 1869–1874, 1997.

[119] A. Pamecha, I. Ebert-Uphoff, and G.S. Chirikjian. Useful metric for modular robot motion planning. *IEEE Transactions on Robotics and Automation*, 13(4):531–545, 1997.

[120] Chanda Paul. Morphology and computation. In *Proceedings of the International Conference on the Simulation of Adaptive Behaviour*, pages 33–38, 2004.

[121] Rolf Pfeifer and Fumia Iida. Morphological computation: connecting body, brain, and environment. *Japanese Scientific Monthly*, 58(2):130–136, 2005.

[122] Rolf Pfeifer, Fumiya Iida, and Gabriel Gómez. Morphological computation for adaptive behavior and cognition. *International Congress Series, Brain-Inspired IT II: Decision and Behavioral Choice Organized by Natural and Artificial Brains. Invited and selected papers of the 2nd International Conference on Brain-inspired Information Technology*, 1291:22–29, 2006.

[123] Cary B. Phillips, Jianmin Zhao, and Norman I. Badler. Interactive real-time articulated figure manipulation using multiple kinematic constraints. In *Proceedings of SIGGRAPH*, pages 245–250, 1990.

[124] Jack Phillips. *Freedom in Machinery.* Cambridge University Press, 2007.

[125] G. Pratt and M. Williamson. Series elastic actuators. In *Proceedings of IEEE International Conference on Intelligent Robots and Systems*, pages 399–406, 1995.

[126] Gill Andrews Pratt. Low impedance walking robots. *Integrative and Comparative Biology*, 42:174–181, 2002.

[127] Jerry Pratt, Chee-Meng Chew, Ann Torres, Peter Dilworth, and Gill Pratt. Virtual model control: An intuitive approach for bipedal locomotion. *The International Journal of Robotics Research*, 20(2):129–143, 2001.

[128] Jerry E. Pratt. Virtual model control of a biped walking robot. Master's thesis, Massachusetts Institute of Technology, 1995.

[129] Jerry E. Pratt. *Exploiting Inherent Robustness and Natural Dynamics in the Control of Bipedal Walking Robots.* PhD thesis, Massachusetts Institute of Technology, 2000.

[130] Mitchell W. Pryor, Ross C. Taylor, Chetan Kapoor, and Delbert Tesar. Generalized software components for reconfiguring hyper-redundant manipulators. *IEEE/ASME Transactions on Mechatronics*, 7(4):475–478, December 2002.

[131] Mitchell Wayne Pryor. *Task-Based Resource Allocation for Improving the Reusability of Redundant Manipulators.* PhD thesis, University of Texas at Austin, 2002.

[132] Stephane Redon and Ming C. Lin. An efficient error-bounded approximation algorithm for simulating quasi-statics of complex linkages. *Computer-Aided Design*, 38:300–314, 2006.

[133] F. Reuleaux. *Kinematics of Machinery.* Macmillan and Co., 1876.

[134] Zaidi Mohd Ripin, Tan Beng Soon, A.B. Abdullah, and Zahurin Samad. Development of a low-cost modular pole climbing robot. In *TENCON*, volume I, pages 196–200, Kula Lumpur, Malaysia, 2000.

[135] Daniela Rus, Zack Butler, Keith Kotay, and Marsette Vona. Self-reconfiguring robots. *Communications of the ACM*, 45(3):39–45, 2002.

[136] Daniela Rus and Gregory S. Chirikjian (eds.). Special issue on self-reconfiguable robots. *Autonomous Robots*, 10(1), January 2001.

[137] Daniela Rus and Marsette Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *Autonomous Robots*, 10(1):107–124, January 2001.

[138] Charles Sherrington. *The integrative action of the nervous system.* Yale University Press, 1906.

[139] J. E. Shigley and J. Uicker. *Theory of Machines and Mechanisms.* McGraw-Hill, 1980.

[140] Ching-Long Shih. Ascending and descending stairs for a biped robot. *Systems, Man, and Cybernetics, IEEE Transactions on*, 29(3):255–268, 1999.

[141] Ken Shoemake. Animating rotation with quatnerion curves. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 245–254, 1985.

[142] B. Siciliano and J.-J.E. Slotine. A general framework for managing multiple tasks in highly redundant robotic systems. In *Fifth International Conference on Advanced Robotics*, pages 1211–1216, 1991.

[143] Russel Smith. Open dynamics engine, 2008. `http://www.ode.org`.

[144] Tristan B. Smith, Javier Barreiro, David E. Smith, Vytas SunSpiral, and Daniel Chavez-Clemente. ATHLETE's Feet: Multi-Resolution Planning for a Hexapod Robot. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, September 2008.

[145] Mark W. Spong, Seth Hutchinson, and M. Vidyasagar. *Robot Modeling and Control.* Wiley, 2006.

[146] Michael Spreng. A probabilistic method to analyze ambiguous contact situations. In *Proceedings of IEEE ICRA*, volume 3, pages 543–548, 1993.

[147] Vijay Srinivasan. *Theory of Dimensioning.* Marcell Dekker, 2003.

[148] Peter J. Staritz, Sarjoun Skaff, Chris Urmson, and William Whittaker. Skyworker: A robot for assembly, inspection and maintenance of large scale orbital facilities. In *IEEE ICRA*, pages 4180–4185, Seoul, Korea, 2001.

[149] Kasper Støy. The ATRON self-reconfigurable robot: challenges and future directions. Presentation at the Workshop on Self-reconfigurable Robotics at the Robotics Science and Systems Conference, July 2005.

[150] Vytas SunSpiral, Daniel Chavez-Clemente, Michael Broxton, Leslie Keely, Patrick Mihelich, David Mittman, and Curtis Collins. FootFall: A ground based operations toolset enabling walking for the ATHLETE rover. In *Proceedings of AIAA Space*, September 2008.

[151] Jackrit Suthakorn and Gregory S. Chirikjian. A new inverse kinematics algorithm for binary manipulators with many actuators. *Advanced Robotics*, 15(2):225–244, 2001.

[152] Ivan Edward Sutherland. *Sketchpad, A Man-Machine Graphical Communication System.* PhD thesis, Massachusetts Institute of Technology, 1963.

[153] Tanio K. Tanev. Kinematics of a hybrid (parallel-serial) robot manipulator. *Mechanism and Machine Theory*, 35:1183–1196, 2000.

[154] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, June 1972.

[155] Russell L. Tedrake. *Applied Optimal Control for Dynamically Stable Legged Locomotion.* PhD thesis, Massachusetts Institute of Technology, 2004.

[156] Cem Unsal, Han Kiliccote, and Pradeep Khosla. A modular self-reconfigurable bipartite robotic system: Implementation and motion planning. *Autonomous Robots*, 10(1):23–40, January 2001.

[157] Serguei Vassilvitskii, Jeremy Kubica, Elanor Rieffel, John Suh, and Mark Yim. On the general reconfiguration problem for expanding cube style modular robots. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*, pages 801–808, Washington, DC, May 2002.

[158] Marsette Vona, Carrick Detweiler, and Daniela Rus. Shady: Robust truss climbing with mechanical compliances. In *International Symposium on Experimental Robotics*, pages 431–440, 2006.

[159] Marsette Vona, David S. Mittman, Jeffrey S. Norris, and Daniela Rus. Using virtual articulations to operate high-DoF manipulation and inspection motions. In *Proc. of Field and Service Robotics*, Cambridge, MA, Jul 2009.

[160] K. J. Waldron. On surface contact joints. Technical Report 1971/AM/2, School of Mechanical and Industrial Engineering, University of New South Wales, Australia, 1971.

[161] K. J. Waldron. A method of studying joint geometry. *Mechanism and Machine Theory*, 7:347–353, 1972.

[162] C. W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least squares methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 16:93–101, 1986.

[163] Chris Welman. Inverse kinematics and geometric constraints for articulated figure manipulation. Master's thesis, Simon Fraser University, 1993.

[164] D. E. Whitney. Resolved motion rate control of manipulators and human prostheses. *IEEE Transactions on Man Machine Systems*, 10(2):47–53, June 1969.

[165] Brian H. Wilcox. ATHLETE: A mobility and manipulation system for the moon. In *IEEE Aerospace Conference*, pages 1–10, March 2007.

[166] Brian H. Wilcox. ATHLETE: A mobility and manipulation system for mobile lunar habitats. In *Lunar and Planetary Science XXXIX*, page 1419, 2008.

[167] Brian H. Wilcox, Todd Litwin, Jeff Biesiadecki, Jaret Matthews, Matt Heverly, Jack Morrison, Julie Townsend, Norman Ahmad, Allen Sirota, and Brian Cooper. ATHLETE: A cargo handling and manipulation robot for the moon. *Journal of Field Robotics*, 24(5):421–434, 2007.

[168] Robert L. Williams, II and James B. Mayhew, IV. Control of truss-based manipulators using virtual serial models. In *The 1996 ASME Design Engineering Technical Conferences and Computers in Engineering Conference*, 1996.

[169] Robert L. Williams, II and James B. Mayhew, IV. Cartesian control of VGT manipulators applied to DOE hardware. In *Proceedings of the Fifth National Conference on Applied Mechanisms and Robotics*, Cincinnati, OH, October 1997.

[170] A. Wolf, H. B. Brown, R. Casciola, A. Costa, M. werin, E. Shamas, and H. Choset. A mobile hyper redundant mechanism for search and rescue tasks. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2889–2895, Las Vegas, Nevada, October 2003.

[171] Giles D. Wood and Dallas C. Kennedy. Simulating mechanical systems in simulink with SimMechanics. Technical report, The Mathworks, 2003.

[172] Jing Xiao and Richard A. Volz. On replanning for assembly tasks using robots in the presence of uncertainties. In *Proceedings of IEEE ICRA*, volume 2, pages 638–645, 1989.

[173] Yeoreum Yoon. Modular robots for making and climbing 3-D trusses. Master's thesis, Massachusetts Institute of Technology, June 2006.

[174] Mark Yim, Ying Zhang, John Lamping, and Eric Mao. Distributed control for 3D metamorphosis. *Autonomous Robots*, 10(1):41–56, 2001.

[175] Yeoreum Yoon and Daniela Rus. Shady3D: a robot that climbs 3D trusses. In *Proceedings of the IEEE International Conference of Robotics and Automation*, pages 4071–4076, 2007.

[176] Kourosh E. Zanganeh and Jorge Angeles. A formalism for the analysis and design of modular kinematic structures. *The International Journal of Robotics Research*, 17(7):720–730, 1998.

[177] Vladimir M. Zatsiorski. *Kinematics of Human Motion*. Human Kinetics, 1998.

[178] Ruixiang Zhang and Prahlad Vadakkepat. Motion planning of biped robot climbing stairs. In *FIRA Robot World Cup*, 2003.

[179] Ying Zhang, Mark Yim, Craig Eldershaw, Dave Duff, and Kimon Roufas. Scalable and reconfigurable configurations and locomotion gaits for chain-type modular reconfigurable robots. In *IEEE International Symposium on Computational Intelligence in Robotics and Automation*, volume 2, pages 893–899, 2003.

[180] Yuan F. Zheng and Jie Shen. Gait synthesis for the SD-2 biped robot to climb sloping surface. *Robotics and Automation, IEEE Transactions on*, 6(1):86–96, 1990.