

The Virtual Erector Set

by

Peter Schröder

Vor-Diplom (B.S.), Technical University of Berlin, Germany, 1987

Submitted to the Media Arts and Sciences Section
in partial fulfillment of the requirements for the degree of


Master of Science

at the

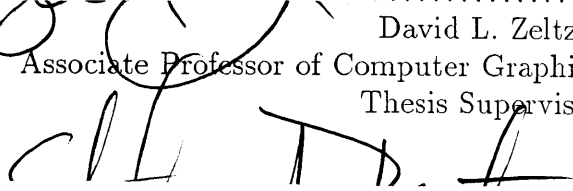
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1990

© Massachusetts Institute of Technology 1990, All Rights Reserved

Signature of Author 
Peter Schröder
Media Arts and Sciences Section
January 12, 1990

Certified by 
David L. Zeltzer
Associate Professor of Computer Graphics
Thesis Supervisor

Accepted by 
Stephen A. Benton
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

FEB 27 1990

LIBRARIES
Rotch

The Virtual Erector Set

by
Peter Schröder

Submitted to the Media Arts and Sciences Section
on January 12, 1990, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis describes the implementation of an algorithm for rigid body dynamics which unifies the advantages of linear recursive algorithms with the advantages of earlier linear algebra based constraint force approaches. No restriction is placed on the joints between links. The algorithm is numerically robust and can deal with arbitrary trees of bodies, including kinematic loops. Motion as well as force constraints on the dynamic behavior of any member of the linkage can be added easily. Through the use of *spatial algebra* notation—including my extension to account for *spatial position*—the mathematical expressions are simplified and more efficient to execute. The algorithm has been implemented on workstation class machines and performs at interactive speeds.

Thesis Supervisor: David L. Zeltzer

Title: Associate Professor of Computer Graphics

⁰This work was supported in part by the National Science Foundation Grant IRI-8712772, BBN Systems and Technology Corp., Rome Air Development Center (RADC) of the Air Force System Command, Defense Advanced Research Projects Agency (DARPA) Grant N00140-88-C-0806, and an equipment grant from Hewlett Packard, Co.

Contents

1	Introduction	6
1.1	Why physics?	7
1.2	Bolio	8
1.3	Realism versus control	8
1.4	Levels of Abstraction	9
1.5	Scenarios of Use	11
1.5.1	The Engineer as User	11
1.5.2	The Animator as User	12
1.5.3	The Student as User	12
2	Related work	14
2.1	Introduction	14
2.2	Constraints as algebraic conditions	15
3	Problems of representing the degrees of freedom of a rigid body	18
3.1	Motivation	18
3.2	The motion of a rigid body	18
3.3	Manifolds	19
3.3.1	Surfaces	21
3.4	The special orthogonal group	21
3.4.1	Rotations as orthogonal matrices	22
3.4.2	Rotations as unit quaternions	23
3.5	The exponential map	23
3.6	Summary	25
4	Numerical Methods	26
4.1	Introduction	26
4.2	Linear Algebra	27
4.3	Importance of local frames in regards to efficiency	28
4.4	Integration	29
4.4.1	Discontinuities of the right hand side	29

4.4.2	The problem of local frames	30
5	Description of the Algorithm	32
5.1	Introduction	32
5.2	Constraints via constraint forces	32
5.2.1	Mathematics of the model	32
5.2.2	Reformulation in Spatial Algebra	35
5.3	Sample Derivations	36
5.3.1	Distance Constraint	36
5.3.2	Point to Point constraint	37
5.4	Implementation	38
5.5	Constraints via recursive propagation	39
5.5.1	Examples of endpoint constraints	40
5.6	Propagating the constrained DOFs	41
5.6.1	The recursion in local frames	43
5.7	Branched kinematic trees	44
5.7.1	Implementation of the matrix reduction	46
5.8	Loops	47
5.8.1	Propagating shared DOFs	48
6	The code	49
6.1	Introduction	49
6.2	Linear algebra classes	49
6.2.1	Generic class of 3 vectors	50
6.2.2	Generic class of spatial vectors	51
6.2.3	Orientations as quaternions	52
6.2.4	Matrix class for 3 by 3 transformations	54
6.2.5	Class of rigid body motions	55
6.2.6	Variable length vectors for Ω -DOF vectors	56
6.2.7	Variable length matrices with 6 rows for \hat{M}	58
6.2.8	Inertial tensor in principal axes of inertia frame	59
6.3	Forces	60
6.4	Support provided by other libraries	62
6.5	The recursion	62
6.6	Front end commands	73
7	Results	76
7.1	Introduction	76
7.2	A tensegrity structure	76
7.3	The policecar scene in “Grinning Evil Death”	80
7.4	Four ways to describe a simple linkage and some of the results	81

7.5	Endpoint constraints and their numerical behavior	84
8	Conclusion and future work	87
A	Spatial Algebra	90
A.1	Line vectors	90
A.1.1	Spatial velocity and acceleration	90
A.1.2	Change of coordinate frame	91
A.1.3	Spatial transpose	91
A.2	Differentiation in moving coordinates	92
A.2.1	Spatial rigid body inertia	93
A.2.2	Spatial force	94

Chapter 1

Introduction

The Virtual Erector Set (VES) is a system which allows the user to build simple mechanical devices (assemblages) in a virtual environment. These devices then follow the laws of physics (Newton's laws). The main objective of this thesis is to build the underlying computational engine for such a system. Since we would like to work with these virtual environments in an interactive setting it is important that the system runs efficiently, yet maintains sufficient generality to be runtime configurable. Obviously these are two conflicting design criteria and we must therefore strive to find a balance.

In the computer graphics community there has been a heightened awareness and interest in accurate physical modeling of rigid-body dynamics Barzel [1], Armstrong [2], Isaacs [3], Wilhelms [4], Witkin [5], Moore [6], Hahn [7]. In computer animation it was found that in order to have realistic looking motion of physical objects, it was not enough to use techniques lent from traditional animation, such as key framing. Either the motion did not "look right", i.e. viewers would be heard saying, "things look like they don't weigh anything", or it required great skill and endless adjustment of non-intuitive parameters to yield a visually pleasing result. On the other hand, there was an impact on the development of interactive systems for modeling physical environments from the engineering disciplines. These already had ways to simulate physical phenomena to great precision but only very limited ways to develop an intuition from the generated data. In these disciplines research in interactive virtual environments opened up the possibility for accelerating the design cycle by allowing the user to iteratively improve a design based on physical criteria, rather than wait for yet another overnight batch simulation to finish.

It is important to notice that the motivations behind these two uses of virtual environments are very different: The designer is interested in seeing and testing an object before it is ever built, allowing her to rapidly refine the design through many iterations before committing to an actual prototype. On the other hand, the animator is not necessarily interested in physical realism per se, but only insofar as it helps to render moving sequences which are perceived as realistic, without the drudgery associated with traditional techniques. Nonetheless both have as a common denominator the use of computer graphics to

look at things that don't exist, i.e. which are *virtual*, and the need for *behavioral realism* (albeit of slightly varying definition). The answer in both cases was the introduction of physical modeling techniques into computer graphics.

Numerical algorithms for the simulation of physical systems have been developed for a long time in many disciplines, especially in engineering. These algorithms however have capitalized on the specific structure of the problems considered to achieve good performance. Thus for every new problem considered by an engineer some amount of recoding was done to simulate the intended system. I, however, strive for a system which allows the user to configure the particular problem of interest at runtime and interactively. Obviously I also have to restrict the problem domain. In my case that restriction is to rigid-body dynamics which I feel is a large and general enough domain to be interesting for applications in virtual environment research, and small enough to allow for an efficient implementation.

A number of approaches for physically realistic modeling in the context of computer graphics have been published, but many of these suffer from slow execution times or lack of generality. It is in the area of speed and generality that the VES makes its contribution.

The VES provides a system which will allow engineers, animators and others to design, construct, test, and explore virtual machines or mechanical assemblies in a simulated Newtonian world. Here two important aspects of design will be unified, the *model building* phase will be linked with the *model simulation* phase.

The VES is able to model rigid body dynamics of trees of links, including loops, with arbitrary joints between the links. Also it is possible to specify arbitrary constraints at the endpoints.

Major parts of the system are

- An input/output module which accepts commands on stdin and writes commands to stdout. Graphics and device management is provided via Bolio (see Brett [8], Zeltzer [9]), so that the VES only needs to know the proper commands for Bolio. This includes the creation of command scripts in Bolio which communicate the relevant information to the VES.
- A constraint force modeling stage to assist in the assembly of structures oriented along the lines of Barzel [1].
- A linear recursive dynamics algorithm able to handle arbitrary trees of linkages including loops based on the work of Lathrop [10].

1.1 Why physics?

One of the goals of work in the area of virtual environments is to create as close as possible an approximation to the real world. A fundamental part of this is physics. After all we want objects in the virtual environment to behave like objects in the real world, e.g. fall to the floor when we drop them. At the same time the computational models need to be flexible so

that we may “tweak” parameters that describe our models interactively and observe their effects.

Even if we want to model and examine non-physical “things” in our virtual environment the metaphor of physics is a powerful one. For example it makes sense to talk about “pulling” the solution of an optimization problem away from a local minimum to a global minimum (see Witkin [11]). This can be subsumed under the headline of interactive steering of simulations, where simulations can be instantiations of abstract concepts, rather than just concrete physical ones.

Furthermore physics is an important building block for any system that strives to allow the user to specify actions at a high level, such as task level animation (see Zeltzer [9]). For example, if we want to specify the actions of a legged figure at the level of commands such as *walk up the stairs* we not only need a conceptual model of walking but also the computational engine that supports the computations of relevant information, such as load on a leg.

1.2 Bolio

One such system for the exploration of virtual environments which is being developed in the Computer Graphics and Animation Group at the Media Laboratory is *Bolio* [8] [9]. Bolio is a distributed graphical simulation platform which allows a number of simulation modules to be executed in the same virtual universe. One of these modules is the VES. In order for the VES to be integrated into Bolio it is implemented as an independent program which reads commands sent to it from Bolio and writes commands which can be interpreted by Bolio. This modular design has a number of advantages:

- The actual program implementing a given capability (the VES in this case) can focus on its particular problem and does not need to know any details about the inner workings (data structures, flow of control, etc.) of Bolio.
- The program is much easier to build and maintain.
- Modules can be exchanged without disturbing the overall design.
- The usefulness of individual components of the overall system increases in a synergistic way. For example, I/O devices such as the Spaceball can now interact with the virtual environment in a physical way, and the VES can be controlled via the Spaceball.

1.3 Realism versus control

One of the dichotomies which we would like to bridge is that of physical realism versus control over the outcome of a simulation. This is especially true in animation, where after all we have a script that we would like the animation to follow, yet it should do so in a

realistic fashion. Consider the example of throwing a ball. Clearly the ball should fly freely through the air, once released from the arm that threw it. On the other hand we might want to have the ball hit a specific target. One possible approach to this is trial-and-error. This, however, is only feasible if the simulation runs fast enough that we can afford the repeated trials in terms of time. Another approach would be inverse dynamics, in which we specify the trajectory completely and then compute the forces necessary to achieve it. Barzel [1] uses such a scheme to maintain the constraints placed on the system. Witkin [5] used this approach to specify weighted goals to be fulfilled over an entire range of time. While that technique can produce a number of desirable effects the amount of computation necessary makes this technique unusable in interactive contexts. These approaches, while able to achieve any desired effect, are not realistic in the sense that we can specify arbitrary trajectories at the price of computing force profiles which are unrealistic. However it would be useful to be able to specify some components in an inverse dynamics fashion while letting others be influenced entirely by their given initial conditions (forward dynamics). McKenna [12] refers to this as *hybrid dynamics*. This allows us to achieve a whole continuum of levels of control.

Thus we are led to consider only algorithms which incorporate *constraints* into an otherwise forward simulation oriented approach.

1.4 Levels of Abstraction

When defining the problem domain that the VES is limiting itself to the notion of *level of abstraction* becomes important. I am not interested in modeling quantum dynamical details for reasons of efficiency, although it would be the most general level at which to model physics. What is the proper level? I have chosen as primitives in the VES *forces*, *constraints*, and *bodies*. These are sufficiently high level to allow efficient algorithms to be used but are fine grain enough to allow the user to construct objects on the level at which we perceive them in the real world, when we talk about mechanics. Not only has this chunking a strong impact on the details of the algorithms used for the computations but also influences the interface that the user is presented with. For example, the user specifies bodies and their physical properties on the level of dimensions, inertia tensors, and weight. Forces are specified as vectors associated with a coordinate system in respect to which they are parameterized. While constraints by themselves are just mathematical functions which fulfill certain conditions. These are the least specified part of the system, hence the VES provides a number of primitive constraints such as point-to-nail, orientation, but also a number of joints, such as ball-and-socket, rotary, etc.

So far I have only talked about the lowest level of organization when connecting and constraining bodies. It is clear that the user will want to build higher level primitives which were originally modeled as individual parts but which she now wants to refer to on a more abstract level. For example an articulated arm which was built originally from its

constituent bodies and joints but can now be referred to as a higher level entity so that we may instantiate several arms of the same specification. These operations are, strictly speaking, not part of the specification language, but are handled (not unlike the I/O facilities for the C programming language) as scripts of primitive operations. The management of these abstraction mechanisms is therefore a task left to the front end. I took this design choice to keep the underlying system simple and efficient.

1.5 Scenarios of Use

In order to give the reader a better sense of the problems for which the VES offers a solution I will give three scenerios of the use of the VES.

1.5.1 The Engineer as User

Suppose you are an engineer trying to improve the design of a manipulator arm. The work cell in which it is to be used on the factory floor has been changed slightly. Your task is to come up with a minimal design change to the manipulator arm to be useful in the new cell.

You start by invoking the VES and load in a model of the redesigned work cell and a model of the original manipulator arm. This model was built by you earlier and saved with all the necessary parameters in a file. So far, gravity is not turned on to keep objects from falling out of reach. First you position the arm's shoulder joint—a ball and socket joint, one of the primitive joints selectable from a menu—at the correct location within the model of the cell. The location itself is specified either at the command line or interactively, using the spaceball, which controls a cursor. The ball and socket joint is represented by a 3d icon. Fixing the position of an object is represented by a *nail* constraint, which specifies that its argument is to be connected to a fixed reference point.

Now you turn on gravity, and some overall damping and let the manipulator arm move into a rest position. You are ready to verify whether the arm can still reach every part of the work cell.

In order to do this you grab the end of the arm with the DataGlove and move the manipulator arm about. The actual grabbing is represented internally as a spring connecting the kinematic DataGlove position to the dynamic model of the manipulator arm¹.

Watching the simulated arm on the monitor it appears that a new corner in the cell prevents the arm from reaching one of the tools. Connecting the “nail” from which the shoulder hangs, to a knob, you can now interactively change the origin of the shoulder, to find a new position for the arm's origin from which it can again reach every relevant part of the workspace.

You notice, though, that the arm has to extend itself farther now than before. Wondering how that will change the necessary torques at the joints, you invoke the actual program which controls the actuators of the real arm. A menu choice lets you examine the torques at the elbow in a window on the screen.

Using the motor control program, you execute a reaching motion to pick up a tool (which has a certain mass associated with it) and observe the torques at the elbow. Indeed they seem dangerously large.

You continue to execute some more experiments of the above kind and arrive at a

¹Once the 3d force feedback joystick is available, it could be used with the feedback controlled by the VES, based on the results of its internal simulation.

conclusion: the simplest way to adapt the manipulator arm to the new work cell is to move its shoulder slightly and exchange one of the actuator motors at the elbow for a motor which can yield higher torques.

1.5.2 The Animator as User

You are working on an animation which involves a scene in which the two main characters get married. The task is to specify the path of the car as it takes off from the church, and to have a string of cans trailing behind the car.

You have already built a little car yesterday and you used a function editor to specify a path for the car to follow. Now you want to attach the string of cans to the back of the car.

The car is loaded into the VES and you proceed to build the string of cans. This is done by instantiating a number of cans and then connecting them with small rigid pieces of wire. The attachment of a wire (an object you also instantiated) to a can proceeds by using the DataGlove to point to the attachment points of two cans and issuing a command connecting the ends of the wire (its two attachment points) with spherical joints to the cans. In this way you build up a string. The VES uses constraint forces at first to move the cans and wires from their initial positions to positions in which they are connected. Again the VES automatically invokes an algorithm to convert this chain into a Lathrop linkage as soon as the constraints are met. From now on the chain will be simulated with the more efficient linear recursive algorithm described in [10].

The car itself is connected to the path it is to follow by specifying a *link to path* constraint. This constraint is selected from a set of primitive endpoint constraints. Using a command in bolio, the output of the function editor² is connected to the input of the *link to path* constraint inside the VES. This operation is completely transparent to the VES, which knows nothing about the origin of the parameters which specify this endpoint constraint. Bolio will take care of polling the function editor and sending the appropriate parameter setting command to the VES, at the beginning of each frame.

Starting the simulation clock in bolio, the car starts moving along its path while the centripetal forces, acting on the string of cans, make them follow along. All the while their collisions with the ground will make them bounce wildly.

1.5.3 The Student as User

Physics class, today's topic: Buckminster Fuller's invention, tensegrity structures. The teacher explains that tensegrity structures are made of rods, which, at their ends, are connected to other rods via rubber strings. One of their peculiar properties is that they will, when correctly assembled, yield a rigid structure, although no two rods will directly connect. This defies intuition and so you go about building one in the VES.

²The function editor is just another client program of bolio, itself attached with a *pipng* command. Typically this might be a spline editor capable of returning a position in space when passed a time.

First you instance a number of rods and some rubber strings—actually modeled as springs—that only exert a force when stretched. Right now you freeze time in the VES, so that you can specify the connectivity of the structure without it bouncing around wildly and collapsing. Following the diagram in the book you specify the connectivity using the Spaceball input device as a 3d pointing cursor.

Now you start the clock in the VES. Since a great number of constraints are all switched on at once, the integrator in the VES takes a few seconds to get going. But then your structure slowly assembles into a flat mess.

Looks like you did some of the connectivity wrong. You stop time and request a connectivity description from the VES. Sure enough, you connected two rubber bands to the wrong rods. That is fixed easily—in this case, by just typing the command to unpost a certain connection at the bolio prompt, followed by the instantiation of the right connectivity.

Starting time again the rods and rubber bands indeed inflate to a soccerball like structure, which lies on your virtual table, slightly flattened by gravity.

Meanwhile the VES is trying to convert the converging structure into a tree with some loops to switch over to the linear recursive algorithm. It is having a very hard time deciding how to cut the loops in this structure, since there are no obvious choices. At this point the VES requests advice from the user (or from an expert system) as to where best to cut loops. Having a higher-level understanding of the topology of the structure, it is an easy matter for you to point to joints at which to cut the structure.

As a result the simulation speeds up noticeably. Now you try an experiment: you instance a large heavy plate. Using the DataGlove you proceed to lay this plate on top of the tensegrity structure. Sure enough, the structure flattens out more, but can clearly support the load put on its top. Another experiment immediately suggests itself: you increase the stiffness of the rubber bands. This is done by using a knob in bolio. As expected the whole structure slowly inflates and deflates as you turn the knob.

You are also interested in finding out what the actual forces are, that act on the various components. From a menu you select *enquire state*, which gives you a submenu of objects to choose from. Selecting *rod 15* you find the compressive force on it to be very small compared to the weight of the plate on top. Observing the relationship between the weight on top and the compressive force in one of the rods you conjecture a simple functional relationship between the two.

In the next class you find out that the actual relationship between load and compressive force is more complicated than the one you conjectured, but is approximately true for a structure whose own weight is negligible compared to the load placed on it.

I have outlined three different scenarios in which the VES is a useful tool. The common thread is the ability to experiment with design choices and receive rapid feedback on their effect. In this way conjectures can be formed and examined quickly. The design of the VES allows for taking advantage of many input modalities, such as menus, scripts, the spaceball, and the DataGlove.

Chapter 2

Related work

2.1 Introduction

When simulating rigid body dynamics, one can distinguish between *forward* and *inverse* simulation. In inverse simulation the user specifies positions for the objects as a function of time and the system responds by computing forces necessary to achieve these goals. While these systems use physical means (forces) to achieve their goals the forces required can be arbitrary and may not be physically realizable. Barzel [1] shows the usefulness of these techniques for the special case of assembling objects.

In contrast, forward simulation techniques are essentially initial value problems. Once we specify the initial conditions and external forces acting on the system, such as gravity, or actuators at a joint, the systems runs its course.

Inverse techniques can still be useful in forward simulation systems, however. E.g. Barzel [1] uses inverse dynamics to establish and maintain connections between bodies while forward simulating the assemblage as a whole.

For the special case of simulating multi-body systems we can distinguish two approaches. Firstly, each body can be treated individually. In this case one simply integrates the equations of motion for each body. In order to connect bodies in linkages or express geometric relationships with the surrounding world, one adds forces to the system to enforce these constraints. This is the approach taken by Barzel [1]. For every body we keep track of 6 equations. Hence the DOFs which get integrated and manipulated by the linear algebra correspond directly to physical DOFs. While easy to set up, this approach requires a maximal set of DOFs (i.e. six per body), and the resulting set of equations is unnecessarily large. What makes matters worse is that these systems also tend to be ill-conditioned, making them fragile in the numerical sense. This has prompted Barzel [1] to use singular value decomposition (SVD) in the solution of this system, making the time complexity of their algorithm cubic.

Secondly, there are those approaches which find the smallest set of DOFs that uniquely

parameterizes the total system; these DOFs don't necessarily correspond one-to-one to actual physical parameters. The main advantage of this is that we only need to keep track of a set of equations of the size of the number of independent DOFs in the system. E.g. Isaacs [3] uses a minimal-set-of-DOFs technique to derive their linear algebra system. Notice, though, that in their extension [13], which allows for kinematic loops, they had to add more variables for the loop closure forces.

Linear recursive schemes are another set of approaches which keep track of only a minimal set of DOFs. As pointed out by others (Wilhelms [14], Isaacs [3]) the recursive schemes are preferable. However, they have not been widely used since they can accommodate neither constraints (other than at the root, or via penalty forces) nor loops.

In this chapter I will discuss the reasoning and arguments behind those approaches which essentially treat the bodies in an assemblage as separate with algebraic conditions enforcing certain constraints among the bodies.

2.2 Constraints as algebraic conditions

The matrix based approaches treat constraints on the dynamic behavior of the system which is being simulated as algebraic relationships between state variables. As an example consider two bodies which are connected by a three DOF rotational joint. This can be expressed as two points, one on each body, being coincident throughout the simulation. A simple way to algebraically encode this is by taking the difference of these two points and requiring it to vanish. In the mathematical literature these systems of differential equations combined with algebraic equations are referred to as *Differential Algebraic Systems* and special differential equation solvers have been proposed for them by Petzold [15].

The algebraic conditions imposed by kinematic constraints are typically not affine spaces of the space of all configurations of the system—a linear space of dimension $6n$ where n is the number of bodies—but *manifolds*. I will refer to these as *constraint manifolds*. This suggests one possible approach for understanding how to integrate the state of an assemblage of bodies. If the system is currently in an allowable configuration, i.e. in the constraint manifold, then it will stay in the manifold to first order if its velocity does not take it out. That is to say, we would like to constrain the system's first derivative to lie in the tangent space to the manifold.

Since this is only a first order technique we need to compensate for errors. This is especially true in practice since we will also have to compensate for numerical error in the integration. One way to do this is to employ a feedback technique which penalizes any deviation from the constraint manifold. A simple approach is to add a force which is proportional to the distance to the constraint manifold. This can be interpreted as a spring which pulls the solution towards the constraint manifold. In practice we also employ a term which is proportional to the velocity taking the system away from the constraint manifold. This corresponds to a damper. Using a damper in parallel is important since a spring alone

will not become active until the system has already left the constraint manifold. Thus if an assemblage is in the manifold, but has a velocity which would take it off at the next instant, it will not experience any corrective term until the assemblage has left the manifold. One way to overcome this is to use very stiff springs (very large spring constants). This has the disadvantage of yielding a so called *stiff system of equations*. Mathematically speaking the system will have widely varying timeconstants, forcing us to take small timesteps even if little motion is occurring. By adding a damper we in effect counteract velocities, which threaten to take the assemblage out of the manifold, before these velocities can generate actual displacements. This approach is used by Isaacs [13], which is similar to the use of critically damped forces by Barzel [1].

So far we have not addressed the problem of moving the assemblage into the manifold. When first assembling a linkage we have to achieve a valid initial configuration. In some cases this is trivial. For example, when connecting two bodies together, it is an easy matter to simply instantiate the second body in a configuration which has it connected to the first. In other cases this is not so easy. Consider a chain which we want to suspend inbetween two poles. We can instantiate the first link in the chain at one pole and all successive links at their predecessors. However it is not clear how to connect the last link to the second tower. We cannot simply instantiate it at the second tower since in general it will then not be connected to its predecessor link. More generally speaking we have to adjust all parameters of the system simultaneously to achieve a valid initial configuration. Since all allowable configurations of our assemblage describe a manifold in state space we can express the problem of finding a valid initial configuration as a multi dimensional root finding problem.

In the numerical literature many schemes can be found for multidimensional root finding. A simple one would be the multidimensional extension of a Newton-Raphson root finder. Multidimensional root finding is a difficult problem in general, but we do have a considerable amount of a priori knowledge of the functions (constraints) whose zeros we are trying to locate. Using such an approach would move the assemblage into an initial configuration in a non-physical way. Depending on the application this can be a reasonable solution.

Barzel [1] uses a technique of critically damped forces to “self assemble” a system of bodies. The approach is mathematically equivalent to the one outlined above but is motivated very differently. He argues that in order to connect two bodies, for example, we can introduce forces into the system, which in effect pull the two bodies together. Again a constraint is nothing more than an algebraic relationship of state variables. The force then to “en-force” the constraint is derived by considering a critically damped differential equation. This method was chosen since the solution to the critically damped second order differential equation asymptotically approaches zero, that is, the state in which the constraint is met. By casting the root finding problem in this form it becomes easy to incorporate outside forces such as gravity, since the root finding itself is cast in terms of forces.

If one desires to use a matrix-based approach to forward simulate a constrained system, as Barzel does, it makes sense to use a “physical” way to initially assemble the system. That way modeling and simulation are unified. As we will see, though, there are numerical

drawbacks to using such a “root finder”.

Chapter 3

Problems of representing the degrees of freedom of a rigid body

3.1 Motivation

When simulating rigid body dynamics we need to consider various derivatives of the location and orientation of a body as a function of time. While the linear (translational) aspects of a body's motion do not present any problems, the rotational (orientation) aspects do require some special attention due to the structure of the space of rotations (namely, it is not a Euclidean space). Traditionally the rotational and translational aspects have been treated separately, leading to expressions which can become very complicated. Spatial algebra (see the appendix) has remedied this situation and provides us with a powerful notation. However the derivative of position and orientation, as well as the integral of spatial velocity have not been treated in the same consistent fashion.

In this chapter I will present an analysis of the derivative of orientation, using some results from differential geometry. With the help of this analysis I argue that a compact notation for *spatial position* is possible, and that this notation has numerical as well as computational advantages. The latter arise from the economy of computation possible due to this notation.

3.2 The motion of a rigid body

The motion of a rigid body is fully characterized by its time dependent orientation and location in three space. While this is intuitively obvious we have strictly speaking used a result from the theory of differential equations and the theory of Frenet curves when stating this. Namely, a Frenet curve in three space is unique up to orthogonal transformations and translations.

There is a fundamental difference between the three translational DOFs and the three

rotational DOFs. The set of all translations of three space has the structure of Euclidean space, while the set of all rotations of three space has the structure of a (hyper-)sphere. To see that this is a significant difference consider that after a rotation of 2π one arrives again in the original orientation. Mathematically we have that the state of a rigid body for purposes of dynamics is a point in the manifold $\mathfrak{R}^3 \times S^3$. This means that we have to use the rules of *calculus on manifolds* when differentiating. As we will see these rules are different from ordinary calculus. Using some results from calculus on manifolds simplifies the expressions we arrive at and sheds additional light on the quantities involved.

3.3 Manifolds

When the student is first introduced to calculus the concept of a linear vector space plays a crucial role. This is so because we are interested in calculus in studying linear approximations to functions. For example consider a function

$$f : \mathfrak{R}^n \rightarrow \mathfrak{R}^m \tag{3.1}$$

from a real vector space of dimension n into a real vector space of dimension m . From ordinary calculus we know that if this function is differentiable we have a function

$$Df : \mathfrak{R}^n \rightarrow M(m, n) \tag{3.2}$$

where $M(m, n)$ denotes the (itself linear) vector space of all m by n matrices. This is to say, for each point x in \mathfrak{R}^n , Df will give us a linear map—an element—from $M(m, n)$, the *derivative* of f in x . This map I will also call $D_x f$.

So far so good. But what about functions which do not go from a linear space into another linear space? Consider a function

$$g : \mathfrak{R} \rightarrow S^2 \tag{3.3}$$

where S^2 denotes the two dimensional unit sphere. For example g could be the parameterization with respect to time of a curve on the unit sphere in \mathfrak{R}^3 . Clearly we would like to be able to talk about the derivative of such a function. This is where the problems start. S^2 is not a linear space, since the sum of two points on a sphere, for example, does not itself lie on the sphere.

Why is linearity so important? Clearly we can draw a picture of the unit sphere in three space, we can imagine a curve on this sphere, and we have no reason do doubt that the derivative vector of such a function is well defined (given the obvious necessary differentiability assumptions). When we talk about a surface we can always come up with a parameterization (at least locally) and a surrounding Euclidean space and thus know

what it means to take derivatives of functions on these surfaces ¹. What is wrong with this picture? One can indeed do calculus on surfaces in three space (and higher order spaces) without ever using the word manifold. However, if the reader has ever looked into a book that assumes this approach, she will quickly find herself engulfed by rather unwieldy formulas. This is so because we need to parameterize the surface first and then differentiate the resulting nested functions. This invokes the differentiation rule for nested functions and leads to rather complicated formulas.

That this approach is almost hopeless becomes clear if we consider higher dimensional spaces or such “strange” sets as the set of all rotations in the set of all matrices². Nonetheless intuition tells us that there should be a way to do calculus on these surfaces in a very natural way, since they are *almost* linear. It is exactly this idea that the notion of a manifold aims to make precise.

A m dimensional **manifold** M is a set such that for each point $p \in M$ there exists a function $f : G \rightarrow \mathbb{R}^m$, where G is an open set of M that contains p , which is differentiable and whose inverse exists and is also differentiable. Loosely speaking we may say that every set which “looks” like Euclidean space is a manifold. As an example consider any surface patch in three space. Locally it is almost flat. Or, “almost linear”.

For example the unit sphere in three space is a two dimensional manifold. Or, a more interesting example, the set of all rotations of three space is a three dimensional manifold.

Now if we have a function

$$f : M \rightarrow N \tag{3.4}$$

from one manifold into another what is the derivative of this function? One obvious way to define it is via the parameterizations leading from \mathbb{R}^m to M , from there to N , and back to \mathbb{R}^n . Now we have a function from \mathbb{R}^m to \mathbb{R}^n and we know how to differentiate those. Indeed if we ever need to compute a derivative for such an f this is just what we need to do. Instead of writing this out, which leads to the mess alluded to above, we continue in a more abstract fashion.

The **tangent space** to a m dimensional manifold M in the point $p \in M$ is defined as $T_p M = \{(p, v) \mid v \in D_q \phi(\mathbb{R}^m)\}$ where $\phi : \mathbb{R}^m \rightarrow M$ is a parameterization of M around p . As an example consider the tangent plane to the two dimensional sphere S^2 in \mathbb{R}^3 in some point. This is exactly the tangent space to S^2 in that point. What used to be $D_x f$ for functions of Euclidean space will now be referred to as $d_p f : T_p M \rightarrow T_{f(p)} N$ on manifolds.

¹The celebrated result of minimal embedding states that any abstract manifold has an embedding in a sufficiently high dimensional $(2m + 1)$ Euclidean space. For a given manifold it might be practically impossible to find this embedding, though.

²As an exercise try to visualize this set!

3.3.1 Surfaces

An interesting class of manifolds are iso-surfaces. An iso-surface may be defined as the pre-image of a point q under some function $f : G \rightarrow W$. The function needs to satisfy certain differentiability conditions which we won't worry about here. Formally we have: Let G be open in V , $f : G \rightarrow W$ be a differentiable function, $\forall p \in G : D_p f$ surjective³, $q \in W$, then $M = f^{-1}(\{q\})$ is a ($m = \dim(V) - \dim(W)$) dimensional manifold with $T_p M = \{(p, v) \mid D_p f(v) = 0\} = \{p\} \times \text{Ker}(D_p f)$. This result is also known as the *Submersion Theorem* for manifolds [16].

An example: the unit sphere in three space An example is the unit sphere S^2 in \mathfrak{R}^3 . Here $f : \mathfrak{R}^3 \rightarrow \mathfrak{R}$ is defined by $f(x, y, z) = x^2 + y^2 + z^2$. By the above we then have that $f^{-1}(\{1\})$ is a $3 - 1 = 2$ dimensional manifold and for any point $p \in S^2$ we have it's tangent space $T_p S^2 = \{v \in \mathfrak{R}^3 \mid v \perp p\}$, which comes as no surprise.

While this is a rather simple example we are now equipped to examine an example which is central to our discussion, the manifold of orientations of three space. But before we go on to that a few remarks are in order about derivatives.

Suppose you have a function of time (one real variable) into a manifold (say, a surface). If we consider the derivative vector to this function we have at each point along its path a tangent space to the manifold (which is a different space at each point!) and a unique vector therein. Given for example S^3 we find that its tangent space at any point $p \in S^3$ is isomorphic to \mathfrak{R}^3 . Thus it makes sense to talk about the derivative curve of our original function. If we differentiate again we find that the tangent space to \mathfrak{R}^3 is naturally \mathfrak{R}^3 . Therefore it is not too surprising when we find below that things are “strange” only on the first derivative.

Later we will consider the inverse problem of finding *integral curves* to a vector field in the *tangent bundle* of a manifold—given a curve in the tangent spaces of some manifold M recover the original function f in M .

3.4 The special orthogonal group

One particular manifold that has received a lot of attention for various reasons is the so called *special orthogonal group* $SO(3)$. It is interesting to us because $SO(3)$ is the set of all rotations of three space. Let Af be the set of all affine transformations of three space. An element of Af consists of an orthogonal linear transformation and a translation. Let these be denoted by (E, \vec{t}) . We can then think of the path of a rigid body as an affine

³I.e. $D_p f$ has rank $\dim(W)$.

transformation-valued function of time

$$\begin{aligned} f : \mathfrak{R} &\rightarrow Af \\ t &\mapsto (E, \vec{t}) \end{aligned} \quad (3.5)$$

That way at any point in time we can take a point x on the body and find its location $f(t)(\vec{x}) = E\vec{x} + \vec{t}$. If we want to find the velocity of a point on the body we differentiate its position function

$$v_x = \left(\frac{d}{dt}f(t)\right)(x) = \left(\frac{d}{dt}E\right)(x) + \frac{d}{dt}\vec{t} \quad (3.6)$$

We don't have any problems differentiating t since it corresponds to the translational DOFs of our body which have the structure of Euclidean three space, but what is $\frac{d}{dt}E$?

The answer to the last question can be found easily if we use the fact that the set of all rotations is a manifold. To see this we will use two different representations for rotations, quaternions and matrices.

3.4.1 Rotations as orthogonal matrices

Consider the set $G = \{A \in M(3, 3) \mid \det(A) > 0\}$ of all matrices with positive determinant⁴, and $W = \{A \in M(3, 3) \mid A = A^T\}$. Let $f : G \rightarrow W$ be defined as $f(A) = AA^T$. Notice that $d_A f$ is surjective for all $A \in G$ (Why?). Therefore by the Submersion theorem $f^{-1}(\{id\}) = \{A \in G \mid AA^T = id\} = SO(3)$ is a three dimensional manifold. Furthermore we know that the tangent space to $SO(3)$ at some point $A \in SO(3)$ is of the form $Ker(d_A f)$. For some $B \in M(3, 3)$ we have $d_A f(B) = BA^T + AB^T$ by the product rule. Thus for any $A \in SO(3)$ we have $T_A SO(3) = \{A\} \times \{X \mid XA^T + AX^T = 0\} = \{A\} \times \{YA \mid Y = -Y^T\}$ (Why?).

At this point we know that whatever $\frac{d}{dt}E$ might be, we can write it as XE where X is an antisymmetric three by three matrix and E is the matrix representation of the orientation. We now use the fact that any three by three antisymmetric matrix can be thought of as a cross product with a specific vector. In our case let's call this vector ω and we can write

$$\frac{d}{dt}E = \omega \times E \quad (3.7)$$

The choice of the name ω is of course deliberate! What properties does ω possess? Consider a point x on our body. What do we know about its velocity? For simplicity assume that the body is only rotating and furthermore that we express x in the coordinates of the frame

⁴Why is this set an open set? Remember that this was one of the preconditions of our theorem about surfaces.

established with E , i.e. $y = Ex$. We then have

$$\frac{d}{dt}f(t)(x) = \omega \times y \quad (3.8)$$

From this we can see that any point y in the direction of ω has a velocity of zero, and the velocity of any other point is proportional to the length of ω and its perpendicular distance to ω . Thus we are justified in calling ω the angular velocity.

We use this in recovering $f(t)$ in our dynamics computations from the current orientation and the angular velocity and integrating. For numerical purposes though one is well advised not to “integrate up” all entries of a matrix as integration error will introduce noticeable skewing. Instead we use quaternion notation.

3.4.2 Rotations as unit quaternions

Consider the linear space \mathfrak{R}^4 with the structure of quaternion algebra, for $p, q \in \mathfrak{R}^4$ define $pq = (s, \vec{x})(r, \vec{y}) = (sr - \vec{x}\vec{y}, s\vec{y} + r\vec{x} + \vec{x} \times \vec{y})$ and $\bar{q} = \overline{(s, \vec{x})} = (s, -\vec{x})$ Again we construct a manifold as before:

$$\begin{aligned} f: \mathfrak{R}^4 &\rightarrow \mathfrak{R}^4 \\ f(q) &= q\bar{q} \\ S^3 &= f^{-1}(\{(1, \vec{0})\}) \end{aligned} \quad (3.9)$$

Or in other words all quaternions with unit length and the property that their conjugate is their inverse. S^3 as the name suggests is three dimensional and there exists a map from its tangent space into S^3 which we will meet later on. For now we notice that one possible map from the elements of S^3 to $SO(3)$ is given by

$$\begin{aligned} q &= (\cos(\frac{\theta}{2}), \sin(\frac{\theta}{2})\vec{r}) \mapsto R(\vec{r}, \theta) \\ R(\vec{r}, \theta)\vec{v} &= q(0, \vec{v})q^{-1}, \vec{v} \in \mathfrak{R}^3 \end{aligned} \quad (3.10)$$

where the latter is quaternion multiplication as defined above. $R(\vec{r}, \theta)$ is a rotation about \vec{r} by an angle of θ .

What is the structure of the tangent space to S^3 ? As before we consider $T_q S^3 = \{q\} \times \{p \mid p\bar{q} + q\bar{p} = 0\} = \{q\} \times \{qp \mid -\bar{p} = p = (0, \vec{x})\}$. As it turns out we will find that $2\vec{x} = \omega$. For numerical purposes we now have a much easier equation

$$\frac{d}{dt}q = \frac{1}{2}q\omega \quad (3.11)$$

3.5 The exponential map

We can now see that when we integrate the equations of motion we are taking curves in tangent spaces into their respective manifolds. This is easy for going from acceleration to velocity, since we are taking a function from one Euclidean space $T_p \mathfrak{R}^6$ into another \mathfrak{R}^6 .

However when we go from the tangent space of our configuration manifold $T_p(\mathbb{R}^3 \times S^3)$ to the manifold itself we have

$$\hat{v} \mapsto \left(\int^t \frac{1}{2} q(0, \omega), \int^t \vec{v} \right) \quad (3.12)$$

That something has changed can be noticed from the fact that differentiating q leads to a differential equation involving q on both sides.

When we draw a picture of a manifold and its associated tangent space in a point, it seems reasonable from the picture that there should be a map which projects a vector from the tangent space into a curve on the manifold. This problem is also referred to as the question of *integral curves*. Given a smooth continuum of vectors in tangent spaces to a manifold, does there exist a function on the manifold whose derivative gives rise to exactly these tangent vectors? One of the first manifolds for which this question was studied was indeed $SO(3)$. For $SO(3)$ the map which takes tangent vectors into curves on the manifold can be found easily. Consider the differential equation

$$\dot{y} = yf \quad (3.13)$$

which has the solution

$$y = e^{\int f} c \quad (3.14)$$

where c is a constant of integration to be adjusted to satisfy the initial conditions. When we compare this equation to equation 3.11 we can see that at least formally we have

$$q(t) = q_0 e^{\int^t \frac{1}{2}(0, \omega)} \quad (3.15)$$

The question that remains is, what does it mean to take the exponential of a quaternion? Since the quaternions have the structure of an algebra we can write

$$e^q = \sum_0^{\infty} \frac{q^k}{k!} \quad (3.16)$$

Since the the Taylor expansion for the exp function is absolutely convergent, the above some converges for any $q \in \mathbb{R}^4$. In practice we would want to avoid an explicit summing of this series. As it turns out we can evaluate the above some in terms of elementary functions for the case of q whose scalar part vanishes. We start by observing that

$$\begin{aligned} (0, \vec{x})(0, \vec{x}) &= (\|\vec{x}\|^2, \vec{x}) \\ (\|\vec{x}\|^2, \vec{0})(0, \vec{x}) &= (0, \frac{\|\vec{x}\|^3}{\|\vec{x}\|} \vec{x}) \end{aligned} \quad (3.17)$$

This allows us to evaluate the exponential by reordering the series and collecting even and

odd powers

$$\begin{aligned}
e^{(0,\vec{v})} &= \sum_0^\infty \frac{(0,\vec{v})^k}{k!} \\
&= \left(\sum_0^\infty \frac{\|\vec{v}\|^{2k}}{(2k)!}, \sum_0^\infty \frac{\|\vec{v}\|^{2k+1}}{(2k+1)!} \vec{v} \right) \\
&= \left(\cos \|\vec{v}\|, \frac{\sin(\|\vec{v}\|)}{\|\vec{v}\|} \vec{v} \right)
\end{aligned} \tag{3.18}$$

The above equation is of fundamental importance to our computing technique, since we can solve equation 3.11 with analytic methods if only we have $\int \omega$. The latter however falls out for free if we just integrate up \hat{v} . Since the rotation of three space which is parameterized by a quaternion has a factor of 1/2 (see equation 3.10) we can now see why we claimed earlier that $2\vec{x} = \omega$. It is also interesting to observe that for a local frame attached at the origin equation 3.11 reduces to multiplying the current angular velocity by the identity, which has no effect.

3.6 Summary

Considering the manifold structure of the set of all rotations I have shown that the integral of ω corresponds exactly to the log of the current orientation. Furthermore exponentiating $\int \omega$ can be thought of as creating a rotation about $\int \omega$ by an amount of $\|\int \omega\|$ radians, giving us a pleasing intuitive interpretation as well. Not only does this simplify the computations necessary in the integrator, but it also reduces the number of equations that need to be integrated from 7 to 6.

In the old formulation we would have had to execute a cross product in the integrator every time the derivative of orientation was requested (as does Barzel [1]), we now simply return the current velocity. Another advantage of this scheme is the fact that we do not need to reunitize the quaternion representation of an orientation in the integrator to avoid numerical drift. This occurs automatically when exponentiating the angular velocity, since for any 3 vector its quaternion exponential is of unit length by definition. Numerical precision is increased at the same time that we achieve a better economy of computation.

Chapter 4

Numerical Methods

4.1 Introduction

When considering the actual implementation of the algorithms discussed in this thesis a number of numerical problems, some of which are unique to doing simulation of dynamics, need attention.

As an example consider the fact that when we perform a simulation of dynamics as opposed to kinematics we are dealing with differential equations. This allows errors to feed back into the system in such a way that, for example, an object accelerates without a force acting on it. Barr reports in [17, pages E44–E46] that performing the computations for the constraint force approach in single precision arithmetic was not sufficient. Wilhelms also reports problems in [14] when attempting to reliably solve the matrix problem. In an earlier implementation of the Barzel [1] algorithm I found matrices with condition numbers surpassing the dynamic range of double precision numbers (approximately 10^{16}). Barzel has used a singular value decomposition (SVD) to deal with this as best as possible. I have also found that some simulations required the use of a SVD to avoid instabilities and that an ordinary conjugate gradient (CG) algorithm was not sufficient. This is a significant drawback insofar as a SVD does not allow for exploiting the considerable sparsity of the matrices involved.

On the other hand the recursive algorithms, one of which I implemented, amount to a direct technique for solving the associated matrix problem. Consider the simple case of a linear chain. Since every body is only connected to its immediate left and right neighbor the associated matrix has tridiagonal structure. Gaussian elimination on such a matrix can be done in linear time and corresponds exactly to the recursion which I would perform in the recursive algorithm.

In the following paragraphs I will discuss some of unique problems associated with the linear algebra based approaches and ways to attack them. This is followed by more detailed considerations of the implementation consequences of local frames, and the problems unique

to integration.

4.2 Linear Algebra

For solving the matrix problem which arises in the constraint force approach we need to use an algorithm which can deal with over-constrained as well as under-constrained systems of linear equations. This is so, since the user might specify constraints which cannot be satisfied at the same time, or constraints which are not sufficient to yield a unique configuration of the assemblage being simulated. While it is reasonable to consider an impossible set of constraints an error, it is nonetheless desirable to return with the “best possible compromise”. Algebraically we have that the right hand side of the system

$$\mathcal{J}\ddot{q} = f \tag{4.1}$$

does not lie in the range of the matrix. In this case the “best possible” solution is the projection of the right hand side onto the range of the matrix. Or in other words, the closest (in the sense of the Euclidean metric) vector to f in the range of \mathcal{J} . If f is in the range of \mathcal{J} but the matrix is rank deficient we have many possible solutions. In this case we want a solution which is orthonormal to the null space of \mathcal{J} . Solutions which do have a contribution in the null space of the matrix are undesirable, since they correspond to solutions which contribute forces to the system that are not necessary to maintain the constraints. At best these forces will move the system in legal but undesired ways, at worst they will cause the integrator to diverge.

One way to find such solutions is via a SVD. When performing a SVD on the matrix we compute explicitly a set of orthogonal vectors which span the range of the matrix and a set of orthogonal vectors which span the null space of the matrix. Hence it is an easy matter to find the desired solution by projecting f onto the vectors that span the range of \mathcal{J} . The numerical properties of the SVD are excellent due to the orthogonality of the bases computed. But as pointed out above a SVD has cubic time complexity even for sparse matrices.

Another set of algorithms available to find a solution which satisfies the above criteria, are the CG algorithms. Given exact arithmetic CG schemes find a least squares solution in a finite number of steps. In practice they are always implemented as iterative algorithms with a criterion for stopping the iteration. All of them are based on the observation that any solution to the original problem also satisfies

$$\|\mathcal{J}\ddot{q} - f\| = 0 \tag{4.2}$$

If the problem is overconstrained and no solution exists, then finding that solution which minimizes the above quantity satisfies our criteria. Avoiding contributions in the null space of \mathcal{J} is achieved by using a starting vector which has no such contribution and only adding

correction terms in the iteration which are not in the null space of \mathcal{J} themselves.

These algorithms can take advantage of the sparsity of the matrix by virtue of the fact that they require the multiplication of \mathcal{J} with certain test vectors. This allows the implementor to use any data structure desired for the matrix itself so long as she supplies a function which executes a matrix multiply. The disadvantage of these schemes is that they tend to square the condition number of the matrix involved. For ill-conditioned systems this actually makes things much worse. One CG algorithm that does not suffer from this is the LSQR algorithm [18]. In their paper Page and Saunders compare the LSQR algorithm with various other CG algorithms and show that it performs by far the best. This means that it converges in the least number of iterations especially when confronted with ill-conditioned systems.

During various rounds of debugging the code I have compared actual solution vectors found by my implementation of the LSQR algorithm with the solution produced by a SVD executed in Mathematica. The difference was on the order of double precision accuracy, 10^{-16} .

4.3 Importance of local frames in regards to efficiency

Featherstone discusses in great length [19] the efficiency advantages of local frames. These fall into the following categories

- simplification of the algebraic expressions due to certain quantities being constant in local frames
- simplification of the computations due to special structure of certain quantities in local frames
- increased numerical precision due to local frames, allowing the integrator to take larger steps

For example, the inertia tensor is constant in body local coordinates. Furthermore by choosing a *principal axes of inertia* frame the inertia tensor itself is fully expressed by four scalars and trivially inverted. Joint axes can also be chosen so as to be constant in the local frame. Another set of simplifications is possible by aligning the local coordinate axes with a joint axis. This means that dot products involving the joint axes reduce to selecting an entry from the vector with which the joint axis is dotted.

The internal representation of spatial transforms also has a great influence on efficiency. A spatial transform can be realized as a 6 by 6 matrix. Due to its spatial orthogonality we can store it as an affine transform (i.e. a 3 vector and a rotation matrix) and execute multiplies with an efficiency gain of factor 6 (for more such optimizations refer to Featherstone [19]). This also has the side effect that all transforms, whether they transform 3 vectors or spatial vectors have the structure of affine transformations.

When bodies are far from the origin, a small change in rotation can result in a large linear displacement in space. Hence the integrator can take larger time steps for a given amount of precision when using local frames (see McKenna [12]).

4.4 Integration

In the spirit of efficiency as well as stability of the numerical computations I am using an adaptive stepsize, adaptive order, Adams-Bashforth predictor-corrector algorithm for the integration (see Gear [20]). I chose this particular algorithm since it requires the least number of right-hand-side evaluations, which, after all, is the most expensive part of the system. This is achieved by using past values of the integrand to estimate future values, as opposed to a Runge-Kutta algorithm which uses several values during the current time step only to discard them immediately afterwards. Aside from the usual adaptive stepsize it also chooses the order of integration adaptively from 1 to 12 (in practice I have mostly seen third to fifth order). These algorithms are in our opinion the best available but are typically very difficult to implement. One has to be mindful of two issues though. One concerns the question of how to deal with discontinuities and the other concerns changes to the algorithm to accommodate the local frames (in essence we are constantly changing the reference frame, leading the integrator to believe that the function to be integrated continually stays at the origin).

4.4.1 Discontinuities of the right hand side

Whenever a new force gets added to the system the joint accelerations are discontinuous. This of course violates the basic assumption of the integrator, which after all uses a polynomial approximation of the function to be integrated. While it is possible to use polynomial approximations of arbitrary precision to “negotiate” the discontinuity it will require the integrator to take much smaller steps, noticeably slowing down the simulation. This in effect removes the advantages that the multistep formulas, such as Adams Bashforth, have over multistage approximations, such as Runge-Kutta. Lotstedt has shown in [21], [22] that the obvious thing to do, i.e. to restart the integrator at the discontinuity, is the mathematically correct thing to do. The overall order of approximation of the integrator does not suffer under this scheme.

Implementation of restart

Since the code in effect uses a polynomial approximation internally, one can request the integrator to interpolate up to a specific time and then restart. Whenever an event occurs at the top level that has an influence on the integration (these are mostly user actions), the algorithm interpolates all states to the current time and uses these on the next step as an initial value problem. Another important support is available for specific events that happen

at a point in time in the future. It is possible to set a “tcrit” value, in effect preventing the integrator from overshooting this particular value of the independent variable when doing the predictor corrector iteration.

Any command that can cause a discontinuity sets an *integrator dirty* flag to trigger the above actions.

4.4.2 The problem of local frames

The actual code I am using for the integration is derived from original sources of Hindmarsh [23], who implemented and improved the algorithm originally proposed by Gear [20]. It assumes a constant reference frame when computing the solution of the system of differential equations. Since I am using local frames the code had to be changed to do this. In a multistage solver such as Runge-Kutta essentially the same problem has to be accounted for, but it is easier, since for a Runge-Kutta integrator one only needs to maintain a consistent coordinate frame for a single step. For a multistep integrator, such as the one I use all past values used in the integration have to be in the same coordinate frame. Since this frame changes at every step one needs to incrementally transform all of the current set of past values at every step. This is further complicated by the fact that the integrator will back-up at times.

Multistep methods construct a polynomial approximation to the function to be integrated. This polynomial can be represented uniquely in various bases. One such basis is the Lagrange basis. This corresponds to maintaining a set of past values of the dependent variable. The basis is dependent on the stepsize currently used in the integrator. Since the integrator adapts its stepsize, this is not the best basis to use. Another basis is given by successive powers of the independent variable. This corresponds to constructing a Taylor polynomial from successive derivatives of the dependent variable. The coefficients of the Taylor polynomial are also dependent on the stepsize. To change stepsize it is only necessary to multiply these coefficients with successive powers of the ratio of the new and the old stepsize. It is for this reason that the integrator I am using, LSODE, uses past values in the equivalent form of higher order derivatives at the current value of the independent variable. This is also referred to as the so-called Nordsiek history representation (see Gear [20]).

Implementation of local frames in LSODE

I modified the code in such a way that every time the integrator attempts to make another step a user-supplied function to set the reference frame is called. For the VES this function linearly transforms the Nordsiek history representation into a set of coefficients which correspond to a basis consisting of the current values of the dependent variable, the current value of the first derivatives, and past values of the first derivatives. By using past values of the *first derivatives* the computation of the new reference frame is further simplified. I use the current value of the spatial position to construct an incremental “push-down” transfor-

mation. This transformation is used to incrementally transform the past derivative values, i.e. velocities and accelerations. Had I used *past values*, which is mathematically equivalent, each one of the past spatial positions would have needed to be converted to an affine transform, multiplied, and converted back. This would increase the number of operations considerably. After this transformation the values are converted back into the Nordsiek history representation. The transformations to and from the Nordsiek representation are precomputed for all orders of integration from 1 to 12 and the code dynamically switches between these depending on the current order of integration.

Chapter 5

Description of the Algorithm

5.1 Introduction

In the previous chapter I have discussed the algorithmic choices that I have made in the implementation. These were based on numerical precision considerations as well as economies of representation. By themselves they were general remarks on the low level elements of the algorithm. That description is followed in this chapter by a detailed description of the linear algebra based approach and the recursive approach.

I will describe the derivation of the mathematics followed by some actual examples of its application. Whenever necessary detailed descriptions of the implementation are also included. The chapter finishes off with a description of my implementation of branches and loops, which Lathrop in his original article ([10]) only treated very cursory and in an extension respectively.

5.2 Constraints via constraint forces

The first stage of the algorithm consists of bringing the assemblage into a valid initial configuration. For this I use an approach similar to the one proposed by Barzel [1]. Since he did not use spatial algebra notation to express his constraints I will rederive this approach in the following paragraph using spatial algebra.

5.2.1 Mathematics of the model

This method describes constraints as functions

$$C_j : Q \rightarrow \mathfrak{R}^{d_j}, \quad j = 1, \dots, n \quad (5.1)$$

where Q is the state space of the system (typically the positions and orientations of the bodies involved, although it is not limited to these) Each such constraint has a certain

dimensionality which is indicated in the above by d_j . For example the constraint that connects two points has $d_j = 3$ since 3 variables (translation) are constrained. C_j must be twice differentiable and

$$C_j(Q) = \vec{0} \iff \text{the constraint is met} \quad (5.2)$$

He then proceeds to derive the forces necessary to meet the constraints and keep them met under the influence of outside forces. The relationship between the force to apply and the current state of the system is derived by considering the constraint function to be subject to the critically damped second order differential equation

$$C_j^{(2)} + \frac{2}{\tau} C_j^{(1)} + \frac{1}{\tau^2} C_j = \vec{0}, \quad j = 1, \dots, n \quad (5.3)$$

This is in contrast to earlier approaches which used springs to pull the system towards satisfying the constraints. By requiring the constraint function to satisfy this differential equation it is ascertained that the variable will asymptotically approach zero as required. Since this a linear differential equation, we now have a linear relationship between the constraint force and other state dependent quantities. This way the total force acting on a body is

$$\vec{F}^i = \left(\sum_{j \text{ acting on body } i} \vec{F}_{C_j} \right) + \vec{F}_e^i, \quad i = 1, \dots, k \quad (5.4)$$

Here \vec{F}_e^i is the external force applied to the body, and i indexes the bodies. \vec{F}_{C_j} arises from constraint C_j and is parameterized by equation 5.3.

This then leads to a linear algebra problem of the form

$$\mathcal{H} \begin{pmatrix} \vec{F}^1 \\ \vdots \\ \vec{F}^k \end{pmatrix} = \begin{pmatrix} -\beta_1 - \frac{2}{\tau_1} C_1^{(1)} - \frac{1}{\tau_1^2} C_1 \\ \vdots \\ -\beta_n - \frac{2}{\tau_n} C_n^{(1)} - \frac{1}{\tau_n^2} C_n \end{pmatrix} \quad (5.5)$$

where β_j contains the parts of $C_j^{(2)}$ which are independent of force and \mathcal{H} is the matrix composed of the $C_j^{(2)}$, $j = 1, \dots, n$ as functionals of a force argument.

There are a number of observations to be made here. When comparing the above formulation with the derivation in Barzel [1] we find an apparently large number of differences. However this is so only on the surface. As a matter of fact all of the additional quantities in [1] are just coordinate transforms. To wit

1. Γ^i and Δ^i are the operators which take a force at some specific point on the body into an acceleration with reference to the center of mass of that body.
2. G_j^i and H_j^i serve as coordinate transforms which take the force \vec{F}_{C_j} and express it in

the coordinate frame of body i .

Another way of looking at the problem is to think of the constraints as defining a manifold in configuration space (as suggested by Witkin [11] and Isaacs [13]). Keeping the constraint met is equivalent to staying in this manifold. At any instant in time this can be achieved by limiting motion to the tangent plane of this manifold.

Assume that the constraint is currently met. The constraint will stay met (to first order) if

$$\dot{C}_j = \nabla_{\dot{q}} C_j = 0 \quad (5.6)$$

This can be interpreted as requiring that the change in state be in a direction in which the constraint itself does not change (at least infinitesimally). Thus the notion of staying in the tangent space of the constraint manifold.

If we assume that velocity is proportional to force, as is approximately the case in an environment dominated by friction and damping, we are let to consider the following linear algebra problem

$$\mathcal{J} \begin{pmatrix} \vec{F}_1 + \vec{F}_{e_1} \\ \vdots \\ \vec{F}_k + \vec{F}_{e_k} \end{pmatrix} = \vec{0} \quad (5.7)$$

where \vec{F}_i , and \vec{F}_{e_i} , $i = 1, \dots, k$ are the combined constraint forces and external forces felt by body i respectively. \mathcal{J} is the matrix of derivatives of the constraints.

The argument can easily be extended to a second order approximation in the following way. We require that the second derivative be zero as well

$$\ddot{C}_j = 0 = \nabla_{\dot{q}} \nabla_{\dot{q}} C_j + \nabla_{\dot{q}} \nabla_{\dot{q}} C_j \quad (5.8)$$

In this fashion we can avoid the buildup of a velocity that would take us off the constraint manifold. Since numerical error will tend to take the system off the manifold anyway, I use a simple control technique which opposes velocities as well as displacements off of the constraint manifold

$$\nabla_{\dot{q}} C_j = -\nabla_{\dot{q}} \nabla_{\dot{q}} C_j - \frac{2}{\tau_j} \nabla_{\dot{q}} C_j - \frac{1}{\tau_j^2} C_j \quad (5.9)$$

The τ_j are time constants which parameterize the speed with which errors are balanced. As can be seen by comparing this equation with equation 5.3 this approach is mathematically equivalent to Barzel's formulation, driving the constrained variables to zero. It is this property that I exploit to bring the assemblage into a valid initial configuration, if this is not already trivially the case.

5.2.2 Reformulation in Spatial Algebra

Since the aim is to use the recursive algorithm I decided to formulate the above in terms of spatial algebra. In this way a uniform treatment of the initial phase of building the assemblage and of forward simulating it, is achieved. Recall Newton's equations are expressed in spatial notation as

$$\hat{\mathbf{a}} + \hat{\mathbf{v}} \hat{\times} \hat{\mathbf{v}} = \hat{\mathbf{f}} \quad (5.10)$$

Solving the above for $\hat{\mathbf{a}}$ and substituting for \ddot{q} in equation 5.9 we get

$$\begin{aligned} & \mathcal{J} \begin{pmatrix} \hat{\mathbf{l}}_1^{-1}(\hat{\mathbf{f}}_{e_1} + \hat{\mathbf{f}}_{C_{b_1}} - \hat{\mathbf{v}}_1 \hat{\times} \hat{\mathbf{l}}_1 \hat{\mathbf{v}}_1) \\ \vdots \\ \hat{\mathbf{l}}_k^{-1}(\hat{\mathbf{f}}_{e_k} + \hat{\mathbf{f}}_{C_{b_k}} - \hat{\mathbf{v}}_k \hat{\times} \hat{\mathbf{l}}_k \hat{\mathbf{v}}_k) \end{pmatrix} \\ & = - \begin{pmatrix} \nabla_{\hat{\mathbf{v}}_1 \dots \hat{\mathbf{v}}_k} \nabla_{\hat{\mathbf{v}}_1 \dots \hat{\mathbf{v}}_k} C_1 \\ \vdots \\ \nabla_{\hat{\mathbf{v}}_1 \dots \hat{\mathbf{v}}_k} \nabla_{\hat{\mathbf{v}}_1 \dots \hat{\mathbf{v}}_k} C_n \end{pmatrix} - \text{Diag}(\frac{2}{\tau_1} \dots \frac{2}{\tau_n}) \mathcal{J} \begin{pmatrix} \hat{\mathbf{v}}_1 \\ \vdots \\ \hat{\mathbf{v}}_k \end{pmatrix} - \begin{pmatrix} \frac{1}{\tau_1^2} C_1 \\ \vdots \\ \frac{1}{\tau_n^2} C_n \end{pmatrix} \end{aligned} \quad (5.11)$$

Here the $\hat{\mathbf{f}}_{C_{b_i}}$ are the accumulated constraint forces as felt by body b_i which in general will be a sum of forces which arise from several constraints all acting on a given body. Notice that in all of this I have ignored all transform matrices.

When the algorithm evaluates equation 5.11 all quantities but $\hat{\mathbf{f}}_{C_{b_i}}$ are known. In order to bring this system into the standard form for solving it we rearrange

$$\begin{aligned} & \mathcal{J} \begin{pmatrix} \hat{\mathbf{a}}_{C_{b_1}} \\ \vdots \\ \hat{\mathbf{a}}_{C_{b_k}} \end{pmatrix} \\ & = -\mathcal{J} \begin{pmatrix} \hat{\mathbf{l}}_1^{-1}(\hat{\mathbf{f}}_{e_1} - \hat{\mathbf{v}}_1 \hat{\times} \hat{\mathbf{l}}_1 \hat{\mathbf{v}}_1) \\ \vdots \\ \hat{\mathbf{l}}_k^{-1}(\hat{\mathbf{f}}_{e_k} - \hat{\mathbf{v}}_k \hat{\times} \hat{\mathbf{l}}_k \hat{\mathbf{v}}_k) \end{pmatrix} - \begin{pmatrix} \nabla_{\hat{\mathbf{v}}_1 \dots \hat{\mathbf{v}}_k} \nabla_{\hat{\mathbf{v}}_1 \dots \hat{\mathbf{v}}_k} C_1 \\ \vdots \\ \nabla_{\hat{\mathbf{v}}_1 \dots \hat{\mathbf{v}}_k} \nabla_{\hat{\mathbf{v}}_1 \dots \hat{\mathbf{v}}_k} C_n \end{pmatrix} \\ & - \text{Diag}(\frac{2}{\tau_1} \dots \frac{2}{\tau_n}) \mathcal{J} \begin{pmatrix} \hat{\mathbf{v}}_1 \\ \vdots \\ \hat{\mathbf{v}}_k \end{pmatrix} - \begin{pmatrix} \frac{1}{\tau_1^2} C_1 \\ \vdots \\ \frac{1}{\tau_n^2} C_n \end{pmatrix} \end{aligned} \quad (5.12)$$

I will write this in the following short form

$$\mathcal{J}\hat{\mathbf{a}} = \phi((\hat{\mathbf{v}}, \hat{\mathbf{p}})_{i=1,\dots,k}, t) \quad (5.13)$$

The actual computation will be performed in each bodies local coordinate frame to avoid numerical difficulties. In the constraint derivations below I will spell out explicitly all the transformations involved.

5.3 Sample Derivations

5.3.1 Distance Constraint

Suppose we have two bodies b_1, b_2 and we want to constrain these two bodies to have a constant, non-zero distance. In general we may wish to constrain specific points on each of the bodies p_1, p_2 to have a desired distance $dd > 0$. The distance constraint then is

$$\langle p_1 - p_2, p_1 - p_2 \rangle - dd^2 = C(p_1, p_2) \stackrel{!}{=} 0 \quad (5.14)$$

where the p_i are in worldspace. For the formulation of the actual algorithm however I will cast everything into body local coordinates.

First however we need to establish the necessary transforms. Let the subscripts $1, 2$ relate to quantities of body 1 and 2 respectively and define

$$\begin{aligned} {}^G\hat{\mathcal{X}}_{L_i} &= \begin{pmatrix} E_i & 0 \\ r_i \times E_i & E_i \end{pmatrix} \\ \hat{\mathcal{T}}_{p_i} &= \begin{pmatrix} id & 0 \\ p_i \times^T & id \end{pmatrix} \\ {}^G\mathcal{X}_{L_i} &= (E_i, r_i) \\ {}^G\mathcal{X}_{L_i}^{-1} &= {}^{L_i}\mathcal{X}_G = (E_i^T, -E_i^T r_i) \\ \hat{\Pi}_1([a, a_0]) &= a \\ \hat{\Pi}_2([a, a_0]) &= a_0 \end{aligned} \quad (5.15)$$

where $i = 1, 2$. The subscripts L and G stand for local and global respectively. The local frame being the center of mass, principal axes of inertia frame and the global being an arbitrary but fixed inertial frame. E is the matrix composed of the column vectors which describe the local frame with respect to the global frame (alternatively it may be thought of as made of rows which are the global frame basis vectors in the local coordinate frame). r_i refers to the location of the center of mass of the respective body in world space. In computer graphics notation the matrix E and the vector r are the rotation and translation respectively that one would use to go from modeling to world coordinate frame.

With these definitions we can formulate the distance constraint precisely (notice that

time is a variable, since the transforms depend on it. However, for clarity I will not write the explicit dependence on t). In order for the expression to be well-defined we need to convert all quantities to some common space. For this I arbitrarily choose the local space of body 1 with ${}_{L_1}X_{L_2} = {}_{L_1}X_G G X_{L_2}$ and ${}_{L_1}\hat{X}_{L_2} = {}_{L_1}\hat{X}_G G \hat{X}_{L_2}$

$$\begin{aligned}
& \langle p_1 - {}_{L_1}X_{L_2} p_2, p_1 - {}_{L_1}X_{L_2} p_2 \rangle - dd^2 = C(p_1, p_2, t) \\
2 \left\langle \begin{pmatrix} 0 \\ p_1 - {}_{L_1}X_{L_2} p_2 \end{pmatrix}, \hat{T}_{p_1} \hat{v}_1 - \hat{T}_{L_1 X_{L_2} p_2} {}_{L_1}\hat{X}_{L_2} \hat{v}_2 \right\rangle &= \nabla_{\hat{v}_1, \hat{v}_2} C(p_1, p_2, t) \\
2 \left\langle \begin{pmatrix} 0 \\ p_1 - {}_{L_1}X_{L_2} p_2 \end{pmatrix}, \hat{T}_{p_1} \hat{a}_1 - \hat{T}_{L_1 X_{L_2} p_2} {}_{L_1}\hat{X}_{L_2} \hat{a}_2 \right\rangle &= \nabla_{\hat{a}_1, \hat{a}_2} C(p_1, p_2, t)
\end{aligned} \tag{5.16}$$

We can rewrite ∇C as an operator defined in body local coordinates

$$\nabla C = \begin{pmatrix} \hat{T}_{p_1}^T \begin{pmatrix} 0 \\ p_1 - {}_{L_1}X_{L_2} p_2 \end{pmatrix} \\ (-\hat{T}_{L_1 X_{L_2} p_2} {}_{L_1}\hat{X}_{L_2})^T \begin{pmatrix} 0 \\ p_1 - {}_{L_1}X_{L_2} p_2 \end{pmatrix} \end{pmatrix}^T \tag{5.17}$$

which should be thought of as a sparse vector. Notice the number of common subterms which should be exploited to cut down the overall operations count.

The expression for $\nabla_{\hat{v}_1, \hat{v}_2} \nabla_{\hat{v}_1, \hat{v}_2} C$ is given in body local coordinates by

$$\langle \hat{T}_{p_1} \hat{v}_1 - \hat{T}_{L_1 X_{L_2} p_2} {}_{L_1}\hat{X}_{L_2} \hat{v}_2, \hat{T}_{p_1} \hat{v}_1 - \hat{T}_{L_1 X_{L_2} p_2} {}_{L_1}\hat{X}_{L_2} \hat{v}_2 \rangle \tag{5.18}$$

5.3.2 Point to Point constraint

This constraint is very similar to the distance constraint in that it constrains two points to have zero distance to one another. This could be implemented with the distance constraint for $dd = 0$, however there is a simpler way which avoids computing the dot product and its derivatives. Define

$$p_1 - p_2 = C(p_1, p_2) \stackrel{!}{=} \vec{0} \tag{5.19}$$

and with a simple derivation as above we get

$$\begin{aligned}
\hat{P}_2 (p_1 - {}_{L_1}X_{L_2} p_2) &= C(p_1, p_2, t) \\
\hat{P}_2 (\hat{T}_{p_1} \hat{v}_1 - \hat{T}_{L_1 X_{L_2} p_2} {}_{L_1}\hat{X}_{L_2} \hat{v}_2) &= \nabla_{\hat{v}_1, \hat{v}_2} C(p_1, p_2, t) \\
\hat{P}_2 (\hat{T}_{p_1} \hat{a}_1 - \hat{T}_{L_1 X_{L_2} p_2} {}_{L_1}\hat{X}_{L_2} \hat{a}_2) &= \nabla_{\hat{a}_1, \hat{a}_2} C(p_1, p_2, t)
\end{aligned} \tag{5.20}$$

Here again we think of ∇C as a sparse vector. Notice that in this case $\nabla \nabla C = 0$.

5.4 Implementation

In the code a particular constraint is derived from an abstract constraint class. Each constraint maintains its own data structures and only needs to define certain member functions. These include *right_hand_side*, which computes ϕ , and *dot*, which evaluates the left hand side of the matrix equation. In this way I never need to define and deal with sparse matrix data structures and can leave it up to the individual constraint to exploit common subexpressions and structure such as spatial orthogonality in a transparent and maximal fashion.

When the linear algebra module gets called, it only receives two lists, *rows*, which is the list of constraints, and *cols*, which is the list of bodies. In this fashion the LSQR algorithm need not know about sparsity or the particular dimensionality of a given member of the row list. A matrix multiply is a linear traversal of the row list, calling each list members' *right hand side* function.

5.5 Constraints via recursive propagation

Since we are interested in connected bodies (if they are not connected any forward integration for the individual body's equations of motion will do) one obvious approach is to parameterize one body in terms of another that it is connected to. It is this point of view that leads to a class of recursive formulations. Featherstone has shown [19] that for more than 9 DOFs in a linkage the most efficient forward simulation algorithm is the *articulated body method*. Based on this formulation Lathrop proposed an algorithm capable of propagating constraints along a linkage.

At first we consider only chains without branches, numbering the bodies from base to tip, $i = 0, \dots, n$. Quantities with script i refer to body i , and $\hat{\mathbf{f}}_i$ is the force exerted by body i onto body $i - 1$. For now the joints between bodies will be one DOF joints only. Joint i , designated $\hat{\mathbf{s}}_i$ connects body i and $i - 1$. Newton's equations can then be expressed as

$$\hat{\mathbf{f}}_i = \hat{\mathbf{l}}_i \hat{\mathbf{a}}_i + \hat{\mathbf{f}}_{i+1} + \hat{\mathbf{p}}_i^v \quad (5.21)$$

which relates $\hat{\mathbf{f}}_i$ to the derivative of the momentum of body i , plus the force that body $i + 1$ exerts on it ($\hat{\mathbf{p}}_i^v = \hat{\mathbf{v}}_i \hat{\times} \hat{\mathbf{l}}_i \hat{\mathbf{v}}_i$ is the bias force). Relating accelerations we get

$$\hat{\mathbf{a}}_i = \hat{\mathbf{a}}_{i-1} + \hat{\mathbf{s}}_i \hat{q} + \hat{\mathbf{v}}_i \hat{\times} \hat{\mathbf{s}}_i \hat{q} \quad (5.22)$$

expressing the acceleration of body i as the sum of the acceleration of body $i - 1$, the acceleration along the joint axis, and accelerations due to the velocity across the joint. Furthermore we have the joint constraint

$$Q_i = \hat{\mathbf{s}}_i^s \hat{\mathbf{f}}_i \quad (5.23)$$

which expresses the force acting along the joint axis. Given this numbering convention, $\hat{\mathbf{f}}_0$ is the force applied by the environment to the linkage and $\hat{\mathbf{f}}_{n+1}$ is the force applied by the tip to the environment. Sign, however, can be incorporated in the affine transform.

Each body has 6 motion DOFs. Given the spatial rigid body inertia tensor this is equivalent to the 6 force DOFs. We introduce another set of 6 abstract DOFs, which can be thought of as parameterizing the balance of force between the proximal (closer to the root) and distal (farther from the root) end of a body. Let these DOFs be represented by Ω_i at body i and assume that the endpoint force and acceleration are some known affine combination of these DOFs

$$\begin{aligned} \hat{\mathbf{a}}_0 &= \hat{\mathbf{M}}_0^a \bar{\Omega}_0 + \hat{\kappa}_0^a \\ \hat{\mathbf{f}}_0 &= \hat{\mathbf{M}}_0^f \bar{\Omega}_0 + \hat{\kappa}_0^f \\ \hat{\mathbf{a}}_n &= \hat{\mathbf{M}}_n^a \Omega_n + \hat{\kappa}_n^a \\ \hat{\mathbf{f}}_{n+1} &= \hat{\mathbf{M}}_{n+1}^f \Omega_n + \hat{\kappa}_{n+1}^f \end{aligned} \quad (5.24)$$

(quantities with an overbar refer to root constraints). Substituting in the expression gov-

erding the force that body n exerts on body $n - 1$ we get

$$\begin{aligned}\hat{\mathbf{f}}_n &= (\hat{\mathbf{l}}_n \hat{\mathbf{M}}_n^a + \hat{\mathbf{M}}_{n+1}^f) \Omega_n + (\hat{\mathbf{l}}_n \hat{\kappa}_n^a + \hat{\kappa}_{n+1}^f + \hat{\mathbf{p}}_n^v) \\ &= \hat{\mathbf{M}}_n^f \Omega_n + \hat{\kappa}_n^f\end{aligned}\tag{5.25}$$

So far we have made no assumptions about the dimensionality of Ω , although typically it will be six dimensional, but the development of the algorithm is independent of the dimensionality of Ω . This is important for the argument allowing loops to be broken. We now have a means to “work” up the chain. Since we really want to solve for the joint accelerations we need to use the entries of Ω to hold these. This can be accomplished by an algebraic manipulation which in effect reflects a relationship among the entries of Ω imposed by the joint constraints $Q_i = \hat{\mathbf{s}}_i^s \hat{\mathbf{f}}_i$. In order for this algebraic manipulation to succeed (see below) we need to ascertain that $\hat{\mathbf{M}}_i^f, i = 0, \dots, n$ has maximal rank. Indeed, if this were not so, we could have a body with non-zero acceleration yet with no force acting on it.

5.5.1 Examples of endpoint constraints

The basic mechanism to formulate an endpoint constraint is to write down the equations which describe the force exerted by the last link onto the environment, as well as the acceleration of the tip link. Forces that the environment exerts onto the tip can be encoded by a sign change, expressing them as an equal and opposite force exerted by the tip onto the environment.

Say, the last body in a linkage has no interaction with the environment. Then the force that it exerts on the environment (or the environment on it) must be zero

$$\begin{aligned}\hat{\mathbf{M}}_{n+1}^f &= 0 \\ \hat{\kappa}_{n+1}^f &= 0\end{aligned}$$

Hence, independent of the values of Ω_n no force is transmitted. Clearly we don't know what the acceleration of the body is. This fact is expressed by

$$\begin{aligned}\hat{\mathbf{M}}_n^a &= \begin{pmatrix} 0 & id \\ id & 0 \end{pmatrix} \\ \hat{\kappa}_n^a &= 0\end{aligned}$$

Now we have in effect used Ω_n to hold the variables directly determining acceleration since

$$\hat{\mathbf{a}}_n = \hat{\mathbf{M}}_n^a \Omega_n + \hat{\kappa}_n^a = \Omega_n^s$$

The spatial transpose arises because the coordinate locations of rotational and translational DOFs are reversed between motion and force type spatial quantities.

Another common constraint is for a body to be connected to the inertial frame. Assume

we want to connect the base of the linkage to the inertial frame. In this case the base body does not experience any acceleration

$$\begin{aligned}\hat{\mathbf{M}}_0^a &= 0 \\ \hat{\kappa}_n^a &= 0\end{aligned}$$

The force that the environment is exerting on body 0, which can be thought of as the reaction force, is not known however

$$\begin{aligned}\hat{\mathbf{M}}_0^f &= id \\ \hat{\kappa}_0^f &= 0\end{aligned}$$

Notice that in this case $\bar{\Omega}_0$ is used to hold the variables which directly determine the reaction that the inertial frame exerts on the base.

5.6 Propagating the constrained DOFs

The main process in going from one body to the next can be summarized as exploiting the joint constraint (equation 5.23) to eliminate one of the abstract DOFs (one of the coefficients in Ω_i) and replacing it by the new DOF \tilde{q}_i , the joint acceleration. This process is repeated recursively up the tree until we get to the root. At that point we can solve for the root DOFs and propagate these back out the tree.

Combining the joint force constraint at joint i with the parameterization of $\hat{\mathbf{f}}_i$ we get

$$Q_i = \hat{\mathbf{s}}_i^s \hat{\mathbf{f}}_i = \hat{\mathbf{s}}_i^s (\hat{\mathbf{M}}_i^f \Omega_i + \hat{\kappa}_i^f) \quad (5.26)$$

Since $\hat{\mathbf{M}}_i^f$ has maximal rank at least one of the coefficients of $\hat{\mathbf{s}}_i^s \hat{\mathbf{M}}_i^f$ must be non-zero, say the j th. Hence we can divide through by it and express the j th coefficient of Ω_i as a constant plus a linear combination of the other coefficients of Ω_i

$$[\Omega_i]_j = \frac{Q_i - \hat{\mathbf{s}}_i^s \hat{\kappa}_i^f}{[\hat{\mathbf{s}}_i^s \hat{\mathbf{M}}_i^f]_j} - \sum_{k \neq j} \frac{[\hat{\mathbf{s}}_i^s \hat{\mathbf{M}}_i^f]_k}{[\hat{\mathbf{s}}_i^s \hat{\mathbf{M}}_i^f]_j} [\Omega_i]_k \quad (5.27)$$

$[\hat{\mathbf{b}}]_j$ designates the j th coefficient of the vector $\hat{\mathbf{b}}$. We can rewrite the DOFs to reflect this

$$\Omega_i = \left(1 - \frac{e_j \hat{\mathbf{s}}_i^s \hat{\mathbf{M}}_i^f}{[\hat{\mathbf{s}}_i^s \hat{\mathbf{M}}_i^f]_j} \right) (1 - e_j e_j^T) \Omega_i + e_j \left(\frac{Q_i - \hat{\mathbf{s}}_i^s \hat{\kappa}_i^f}{[\hat{\mathbf{s}}_i^s \hat{\mathbf{M}}_i^f]_j} \right) \quad (5.28)$$

e_j designates a vector with all zeros but in the j th position. We define

$$\Omega_i^* = (1 - e_j e_j^T) \Omega_i$$

which is just Ω_i with the j th entry set to zero. Substituting the new expression for Ω_i into the parameterizations of force and acceleration at body i we get

$$\begin{aligned} \hat{\mathbf{a}}_i &= \left\{ \hat{\mathbf{M}}_i^a \left(1 - \frac{e_j \hat{\mathbf{s}}_i^a \hat{\mathbf{M}}_i^f}{[\hat{\mathbf{s}}_i^a \hat{\mathbf{M}}_i^f]_j} \right) \right\} \Omega_i^* + \left\{ \hat{\mathbf{M}}_i^a e_j \left(\frac{Q_i - \hat{\mathbf{s}}_i^a \hat{\kappa}_i^f}{[\hat{\mathbf{s}}_i^a \hat{\mathbf{M}}_i^f]_j} \right) + \hat{\kappa}_i^a \right\} \\ &= \hat{\mathbf{M}}_i^{a*} \Omega_i^* + \hat{\kappa}_i^{a*} \\ \hat{\mathbf{f}}_i &= \left\{ \hat{\mathbf{M}}_i^f \left(1 - \frac{e_j \hat{\mathbf{s}}_i^a \hat{\mathbf{M}}_i^f}{[\hat{\mathbf{s}}_i^a \hat{\mathbf{M}}_i^f]_j} \right) \right\} \Omega_i^* + \left\{ \hat{\mathbf{M}}_i^f e_j \left(\frac{Q_i - \hat{\mathbf{s}}_i^a \hat{\kappa}_i^f}{[\hat{\mathbf{s}}_i^a \hat{\mathbf{M}}_i^f]_j} \right) + \hat{\kappa}_i^f \right\} \\ &= \hat{\mathbf{M}}_i^{f*} \Omega_i^* + \hat{\kappa}_i^{f*} \end{aligned} \quad (5.29)$$

Since the j th column of $\left(1 - \frac{e_j \hat{\mathbf{s}}_i^a \hat{\mathbf{M}}_i^f}{[\hat{\mathbf{s}}_i^a \hat{\mathbf{M}}_i^f]_j} \right)$ is zero so is the j th column of $\hat{\mathbf{M}}_i^{f*}$ and $\hat{\mathbf{M}}_i^{a*}$. We exploit this to augment Ω_i^* with the unknown joint acceleration at joint i as our new DOF to be solved for

$$\begin{aligned} \Omega_{i-1} &= \Omega_i^* + e_j \ddot{q}_i \\ \hat{\mathbf{a}}_i &= \hat{\mathbf{M}}_i^{a*} \Omega_{i-1} + \hat{\kappa}_i^{a*} \\ \hat{\mathbf{f}}_i &= \hat{\mathbf{M}}_i^{f*} \Omega_{i-1} + \hat{\kappa}_i^{f*} \end{aligned} \quad (5.30)$$

Having rewritten $\hat{\mathbf{a}}_i$ and $\hat{\mathbf{f}}_i$ in this form as functions of Ω_{i-1} we can now substitute these in the recursive relationships for acceleration and force (equations 5.21, 5.22) to arrive at the parameterization of $\hat{\mathbf{a}}_{i-1}$ and $\hat{\mathbf{f}}_{i-1}$

$$\begin{aligned} \hat{\mathbf{a}}_{i-1} &= (\hat{\mathbf{M}}_i^{a*} - \hat{\mathbf{s}}_i e_j^T) \Omega_{i-1} + (\hat{\kappa}_i^{a*} - \hat{\mathbf{v}}_i \hat{\times} \hat{\mathbf{s}}_i \dot{q}_i) \\ &= \hat{\mathbf{M}}_{i-1}^a \Omega_{i-1} + \hat{\kappa}_{i-1}^a \\ \hat{\mathbf{f}}_{i-1} &= (\hat{\mathbf{l}}_{i-1} \hat{\mathbf{M}}_i^{a*} + \hat{\mathbf{M}}_i^{f*}) \Omega_{i-1} + (\hat{\mathbf{l}}_{i-1} \hat{\kappa}_i^a + \hat{\kappa}_i^{f*} + \hat{\mathbf{p}}_{i-1}^v) \\ &= \hat{\mathbf{M}}_{i-1}^f \Omega_{i-1} + \hat{\kappa}_{i-1}^f \end{aligned} \quad (5.31)$$

At the root then we have a parameterization of $\hat{\mathbf{a}}_0$ and $\hat{\mathbf{f}}_0$ in terms of the DOFs of the linkage itself and a prescribed parameterization due to the root constraint. Hence we can actually

solve for Ω_0 by considering the system

$$\begin{aligned}
\hat{\mathbf{a}}_0 &= \hat{\mathbf{M}}_0^a \Omega_0 + \hat{\kappa}_0^a \\
&= \hat{\mathbf{M}}_0^a \bar{\Omega}_0 + \hat{\kappa}_0^a \\
\hat{\mathbf{f}}_0 &= \hat{\mathbf{M}}_0^f \Omega_0 + \hat{\kappa}_0^f \\
&= \hat{\mathbf{M}}_0^f \bar{\Omega}_0 + \hat{\kappa}_0^f
\end{aligned} \tag{5.32}$$

This system might not have any solution, which is the case when the constraints placed on the linkage are incompatible with constraints placed on the root. It might have many solutions, which means that the force and acceleration are determined, but not all DOFs can be solved for. An example is a grip which holds a brick. The accelerations of the contact points can be solved for, namely zero, but the forces can not be uniquely solved for. My implementation uses the LSQR algorithm to solve the following equivalent system

$$\begin{pmatrix} \hat{\mathbf{M}}_0^a & -1 & 0 & 0 \\ \hat{\mathbf{M}}_0^f & 0 & 0 & -1 \\ 0 & -1 & \hat{\mathbf{M}}_0^a & 0 \\ 0 & 0 & \hat{\mathbf{M}}_0^f & -1 \end{pmatrix} \begin{pmatrix} \Omega_0 \\ \hat{\mathbf{a}}_0 \\ \bar{\Omega}_0 \\ \hat{\mathbf{f}}_0 \end{pmatrix} = \begin{pmatrix} -\hat{\kappa}_0^a \\ -\hat{\kappa}_0^f \\ -\hat{\kappa}_0^a \\ -\hat{\kappa}_0^f \end{pmatrix} \tag{5.33}$$

The solution for the root acceleration is used for updating the root body and the solution for Ω_0 is used to recur back down the chain. This is accomplished by using equation (5.27). Each body contains local storage holding the j value designating the entry that was freed up in the recursion up the chain to make room for the joint acceleration. Now that we have Ω_i we have

$$\ddot{\mathbf{q}}_i = [\Omega_i]_j \tag{5.34}$$

and remembering that Ω_i^* is just Ω_i with the j th entry zeroed

$$\Omega_{i+1} = \Omega_i^* + e_j \frac{Q_i - \hat{\mathbf{s}}_i^s \hat{\kappa}_i^f}{[\hat{\mathbf{s}}_i^s \hat{\mathbf{M}}_i^f]_j} - \frac{\hat{\mathbf{s}}_i^s \hat{\mathbf{M}}_i^f}{[\hat{\mathbf{s}}_i^s \hat{\mathbf{M}}_i^f]_j} \Omega_i^* \tag{5.35}$$

In the actual implementation all that happens is that the j th entry of Ω is recovered, set to zero and then filled with the above expression. Since all the elements of the above expression, but Ω_i^* , are subexpressions of earlier computations they are simply stored in the body data structure as well. This way the actual computation only involves the last dot product.

5.6.1 The recursion in local frames

So far I have ignored all questions of frames. When going to local frames the $\hat{\mathbf{a}}_i$ and $\hat{\mathbf{f}}_i$ have to be transformed incrementally from one body to the next. Let ${}_{i-1}\hat{\mathbf{X}}_i$ designate the spatial

transformation that takes us from the local frame of body i to body $j - 1$. The recursion then becomes

$$\begin{aligned}
\hat{M}_{i-1}^a &= {}_{i-1}\hat{X}_i \left(\hat{M}_i^{a*} - \hat{s}_i e_j^T \right) \\
\hat{\kappa}_{i-1}^a &= {}_{i-1}\hat{X}_i \left(\hat{\kappa}_i^{a*} - \hat{v}_i \hat{X} \hat{s}_i \dot{q}_i \right) \\
\hat{M}_{i-1}^f &= \hat{l}_{i-1} \hat{M}_{i-1}^a + {}_{i-1}\hat{X}_i \hat{M}_i^{f*} \\
\hat{\kappa}_{i-1}^f &= \hat{l}_{i-1} \hat{\kappa}_{i-1}^a + {}_{i-1}\hat{X}_i \hat{\kappa}_i^{f*} + \hat{\rho}_{i-1}^v
\end{aligned} \tag{5.36}$$

On the way down, when we recover the joint accelerations no transformations are necessary, since the Ω_i are not expressed with respect to any basis. The joint accelerations which are recovered from the Ω_i are scalars that are always relative to the constant joint axis and hence also do not need any transformation.

5.7 Branched kinematic trees

In order to deal with branched kinematic trees the original numbering scheme which parameterized the recursion has to be modified. While every body still has a unique parent, some bodies have multiple children. In the numbering of the bodies we only require that successor bodies (more distal) have numbers higher than their predecessors (more proximal). For body i the index of its predecessor is λ_i and μ_i is the set of its successors with $\mu_i(k)$ the k th element of that set. Joint i still connects body i and λ_i . The acceleration is unchanged and given by

$$\hat{a}_i = \hat{a}_{\lambda_i} + \hat{s}_i \ddot{q}_i + \hat{v}_i \hat{X} \hat{s}_i \dot{q}_i \tag{5.37}$$

The force of body i now has a contribution from all successor bodies

$$\hat{f}_i = \hat{l}_i \hat{a}_i + \hat{\rho}_i^v + \sum_{j \in \mu_i} \hat{f}_j$$

The acceleration of body i is parameterized in terms of quantities which arise from the successors of body i when going across the joints $k \in \mu_i$. Say we have m children of body i , then we have m different ways to express \hat{a}_i

$$\hat{a}_i = \hat{M}_{i_k}^a \Omega_{i_k} + \hat{\kappa}_{i_k}^a \tag{5.38}$$

In what follows I will refer to the elements of μ_i by the numbers $k = 1, \dots, m$ to simplify the discussion. We also have one equation expressing \hat{f}_i in terms of $\Omega_{i_k}^*$. Using the fact that $\hat{M}_j^{f*} \Omega_{i_j} = \hat{M}_j^{f*} \Omega_{i_j}^*$ and choosing one of the parameterizations of \hat{a}_i , say $j = 1$ to substitute into the recursive expression for the force (equation 5.21), we get

$$\hat{f}_i = (\hat{l}_i \hat{M}_{i_1}^a + \hat{M}_{i_1}^{f*}) \Omega_{i_1} + \sum_{j=2}^m \hat{M}_j^{f*} \Omega_{i_j} + (\hat{l}_i \hat{\kappa}_{i_1}^a + \sum_{j=1}^m \hat{\kappa}_j^{f*} + \hat{\rho}_i^v) \tag{5.39}$$

we now have a system of $m + 1$ equations relating the Ω_i to each other. We can exploit these to build a parameterization of force and acceleration at body i in terms of a new Ω_i .

Lathrop treats this case in an extension to his algorithm and shows that this system of equations can be solved in time linear in the number of distal joints of body i . Finding the new parameterization however is not entirely trivial since we cannot just solve for it directly as we don't yet actually know $\hat{\mathbf{a}}_i$ and $\hat{\mathbf{f}}_i$. For this I have chosen the following approach. For greater clarity I will rename the matrices of the force parameterization

$$\begin{aligned}\hat{\mathbf{A}}_1 &= \hat{\mathbf{l}}_i \hat{\mathbf{M}}_1^a + \hat{\mathbf{M}}_1^{f*} \\ \hat{\mathbf{A}}_j &= \hat{\mathbf{M}}_j^{f*}, j = 2, \dots, m\end{aligned}$$

We also collect the constant terms

$$\begin{aligned}\hat{\eta}_1 &= \hat{\kappa}_1 \\ \hat{\eta}_j &= \hat{\kappa}_1 - \hat{\kappa}_j, j = 2, \dots, m \\ \hat{\eta}_{m+1} &= \hat{\mathbf{l}}_i \hat{\kappa}_1^a + \sum_{j=1}^m \hat{\kappa}_j^{f*} + \hat{\mathbf{p}}_i^v\end{aligned}$$

This leaves us with the following matrix equation

$$\begin{pmatrix} \hat{\mathbf{a}}_i \\ \hat{\mathbf{0}} \\ \vdots \\ \hat{\mathbf{0}} \\ \vdots \\ \hat{\mathbf{0}} \\ \hat{\mathbf{f}}_i \end{pmatrix} = \begin{pmatrix} \hat{\mathbf{M}}_1^a & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ \hat{\mathbf{M}}_1^a & -\hat{\mathbf{M}}_2^a & 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & & \ddots & & & & \vdots \\ \hat{\mathbf{M}}_1^a & 0 & \cdots & 0 & -\hat{\mathbf{M}}_j^a & 0 & \cdots \\ \vdots & & & & & \ddots & \\ \hat{\mathbf{M}}_1^a & 0 & \cdots & \cdots & \cdots & 0 & -\hat{\mathbf{M}}_m^a \\ \hat{\mathbf{A}}_1 & \hat{\mathbf{A}}_2 & \cdots & \cdots & \hat{\mathbf{A}}_j & \cdots & \hat{\mathbf{A}}_m \end{pmatrix} \begin{pmatrix} \Omega_1 \\ \vdots \\ \Omega_j \\ \vdots \\ \Omega_m \end{pmatrix} + \begin{pmatrix} \hat{\eta}_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \hat{\eta}_{m+1} \end{pmatrix} \quad (5.40)$$

Since the left hand side of this equation has at most 12 independent scalars, the matrix on the right can at most have rank 12. But that means that by a suitable coordinate transformation we can rewrite the system as

$$\begin{bmatrix} \hat{\mathbf{a}}_i \\ \hat{\mathbf{f}}_i \end{bmatrix} = B\Omega_i + k \quad (5.41)$$

where the length of Ω_i is less than or equal to 12. Once we achieve this the first 6 rows of B will be the new $\hat{\mathbf{M}}_i^a$ and the bottom 6 rows will give us $\hat{\mathbf{M}}_i^f$ while the first 6 rows of k will be $\hat{\kappa}_i^a$ and the bottom 6 rows will be $\hat{\kappa}_i^f$.

5.7.1 Implementation of the matrix reduction

One possible way to do affect the change to the form outlined in equation (5.41) is to use Gaussian elimination on the branch equations (5.40), which is of linear time complexity in this case due to the special structure of the matrix. This is the approach suggested by Lathrop. It however suffers from numerical problems due to ill conditioning. Furthermore it is complicated by the fact that we can only do row operations on the rows corresponding to $j = 2, \dots, m$ and otherwise have to use column operations which makes it very difficult to recover the individual Ω_i from Ω_i when we come back down the tree to solve for the actual joint accelerations.

For $j = 2, \dots, m$ consider the equation

$$\hat{\mathbf{o}} = \hat{\mathbf{M}}_1^a \Omega_1 - \hat{\mathbf{M}}_j^a \Omega_j + \hat{\eta}_j$$

Using a SVD this equation can be partially solved for Ω_j in terms of Ω_1

$$\begin{bmatrix} u_1, \dots, u_{k_j} \end{bmatrix} \text{Diag}(\rho_1, \dots, \rho_{k_j}, 0_{k_j+1}, \dots, 0_{r_j}) \begin{bmatrix} v_1, \dots, v_{r_j} \end{bmatrix}^T \Omega_j = \hat{\mathbf{M}}_1^a \Omega_1 + \hat{\eta}_j$$

Here $u_l, l = 1, \dots, k_j$ is an orthogonal basis for the range of $\hat{\mathbf{M}}_j^a$, $\rho_l, l = 1, \dots, k_j$ are the associated singular values, and $v_l, l = 1, \dots, r_j$ is an orthogonal basis for the domain of Ω_j , where r_j is equal to the length of Ω_j . First we change the basis of Ω_j to the new orthogonal basis

$$\tilde{\Omega}_j = \begin{bmatrix} v_1, \dots, v_{r_j} \end{bmatrix}^T \Omega_j$$

We can now rewrite the first k_j coefficients of $\tilde{\Omega}_j$ in terms of Ω_1

$$\begin{aligned} \left[\tilde{\Omega}_j \right]_{1, \dots, k_j} &= \text{Diag}(\rho_1^{-1}, \dots, \rho_{k_j}^{-1}) \begin{bmatrix} u_1, \dots, u_{k_j} \end{bmatrix}^T \left(\hat{\mathbf{M}}_1^a \Omega_1 + \hat{\eta}_j \right) \\ &= \hat{\mathbf{N}}_j^a \Omega_1 + \hat{\nu}_j \end{aligned}$$

The same substitution is applied in the last row, first transforming to the new basis

$$\tilde{\tilde{\mathbf{A}}}_j = \hat{\mathbf{A}}_j \begin{bmatrix} v_1, \dots, v_{r_j} \end{bmatrix}$$

and then substituting the first k_j coefficients of $\tilde{\tilde{\mathbf{A}}}_j$ as functions of Ω_1 (to avoid yet another set of variable names I will use the C language operator “+ =”)

$$\begin{aligned} \hat{\mathbf{A}}_1 \quad + &= \left[\tilde{\tilde{\mathbf{A}}}_j \right]_{\text{Col}=1, \dots, k_j} \hat{\mathbf{N}}_j^a \\ \hat{\eta}_{m+1} \quad + &= \left[\tilde{\tilde{\mathbf{A}}}_j \right]_{\text{Col}=1, \dots, k_j} \hat{\nu}_j \end{aligned}$$

This way we have effectively used the equations $j = 2, \dots, m$ to eliminate as many DOFs

from the last row of equation (5.40) as possible. Remembering that $\hat{\mathbf{A}}_1$ and $\hat{\eta}_{m+1}$ have been modified above we are left with

$$\begin{aligned}\hat{\mathbf{a}}_i &= \hat{\mathbf{M}}_1^a \Omega_1 + \hat{\eta}_1 \\ \hat{\mathbf{f}}_i &= \hat{\mathbf{A}}_1 \Omega_1 + \left(\sum_{j=2}^m \left[\tilde{\hat{\mathbf{A}}}_j \right]_{\text{Col}=k_j+1, \dots, r_j} \left[\tilde{\Omega}_j \right]_{k_j+1, \dots, r_j} \right) + \hat{\eta}_{m+1}\end{aligned}$$

We are now down to the desired 12 rows but might still have more columns than the rank of the matrix we started with (equation 5.40). At this point the algorithm does a final SVD to remove all unnecessary DOFs leaving us with the desired system of equation (5.41).

When the recursion comes back down with the actual Ω_i we are faced with the task of reconstructing the original Ω_j to be sent down the child chains. All necessary quantities reside in local storage within the body data structure itself. We now have

$$\begin{aligned}\hat{\mathbf{M}}_i^a \Omega_i &= \hat{\mathbf{a}}_i - \hat{\kappa}_i^a \\ &= \hat{\mathbf{M}}_1^a \Omega_1\end{aligned}$$

As before I use a SVD on $\hat{\mathbf{M}}_1$ to recover Ω_1 . With Ω_1 recovered the algorithm uses the SVDs of equation (5.7.1) to build the first k_j entries of $\tilde{\Omega}_j$ for all $j = 2, \dots, m$ from which we can recover the actual Ω_j themselves

$$\Omega_j = [v_1, \dots, v_{k_j}] \left(\hat{\mathbf{N}}_j^a \Omega_1 + \hat{v}_j \right)$$

The recursion continues on every child chain with these Ω_j .

5.8 Loops

Loops are handled by breaking them. After that they can be treated as the endpoints of two linkages. The only difference is that constraints on the DOFs of one arm have to be related to the corresponding DOFs of the other arm. To simplify the discussion I will first consider breaking an ordinary chain between bodies $i + 1$ and i . From the point of view of body i the force $\hat{\mathbf{f}}_{i+1}$ can be thought of as the force that it exerts on the environment if it was the last body in a chain. Similarly for body $i + 1$ the force $\hat{\mathbf{f}}_{i+1}$ is what it experiences from the environment, or we can say the negative of what it exerts on the environment. We now set $j = i + 1$ and think of body i and body j as the last bodies in their respective linkages. I will indicate this by the indices j_{n+1} and i_{n+1} . This leads to

$$\begin{aligned}\hat{\mathbf{f}}_{j_{n+1}} &= -\hat{\mathbf{f}}_{i_{n+1}} \\ \hat{\mathbf{f}}_{i_{n+1}} &= \hat{\mathbf{f}}_{i+1}\end{aligned}$$

Using the same argument for accelerations we get

$$\begin{aligned}\hat{\mathbf{a}}_{j_n} &= \hat{\mathbf{a}}_i + \hat{\mathbf{s}}_{i+1} \ddot{q}_{i+1} + \hat{\mathbf{v}}_{i+1} \hat{\times} \hat{\mathbf{s}}_{i+1} \dot{q}_{i+1} \\ \hat{\mathbf{a}}_{i_n} &= \hat{\mathbf{a}}_i\end{aligned}$$

and the joint constraint becomes

$$Q_{i+1} = \hat{\mathbf{s}}_{i+1}^s \hat{\mathbf{f}}_{i+1}$$

Let the unknown DOFs be

$$\Omega_{ij} = (\hat{\mathbf{a}}_i^T, \hat{\mathbf{f}}_j^T, \ddot{q}_j)^T$$

We can now define the parameterization for the last body of each of these two chains as we would normally do for an ordinary chain

$$\begin{aligned}\hat{\mathbf{a}}_{i_n} &= (id, 0, \hat{\theta}) \Omega_{ij} + \hat{\theta} \\ &= \hat{\mathbf{M}}_{i_n}^a \Omega_{ij} + \hat{\kappa}_{i_n}^a \\ \hat{\mathbf{f}}_{i_n} &= (\hat{\mathbf{l}}_i, id, \hat{\theta}) \Omega_{ij} + \hat{\mathbf{p}}_{i_n}^v \\ &= \hat{\mathbf{M}}_{i_n}^f \Omega_{ij} + \hat{\kappa}_{i_n}^f \\ \hat{\mathbf{a}}_{j_n} &= (id, 0, \hat{s}_j) \Omega_{ij} + \hat{\mathbf{v}}_j \hat{\times} \hat{\mathbf{s}}_j \dot{q}_j \\ &= \hat{\mathbf{M}}_{j_n}^a + \hat{\kappa}_{j_n}^a \\ \hat{\mathbf{f}}_{j_n} &= (\hat{\mathbf{l}}_j, -id, \hat{\mathbf{l}}_j \hat{\mathbf{s}}_j) \Omega_{ij} + \hat{\mathbf{l}}_j (\hat{\mathbf{v}}_j \hat{\times} \hat{\mathbf{s}}_j \dot{q}_j) \\ &= \hat{\mathbf{M}}_{j_n}^f \Omega_{ij} + \hat{\kappa}_{j_n}^f\end{aligned}\tag{5.42}$$

Notice that so far we have 13 DOFs, but there are only 12 at the breaking point since one of the DOFs is constrained by the joint. Before we start the recursion I exploit this constraint as before (assume the coefficient l of \hat{s}_j^s is non-zero)

$$[\Omega_{ij}]_{7,\dots,12} = \left(1 - \frac{e_l \hat{s}_j^s}{[\hat{s}_j^s]_l}\right) (1 - e_l e_l^T) [\Omega_{ij}]_{7,\dots,12} + e_l \frac{Q_j}{[\hat{s}_j^s]_l}$$

This time however we use this fact to actually eliminate a column from the $\hat{\mathbf{M}}$ matrices above, bringing them all down to 12 columns. Now it is also clear why we wanted the development of the algorithm to be independent of the actual length of Ω .

5.8.1 Propagating shared DOFs

The DOFs arising from the breaking of a loop come in pairs. When the recursion moves up one of the arms, we encounter constraints on these DOFs. Any such constraint has to be communicated to the same DOFs off the other arm. This is accomplished by substituting the expression of equation (5.27) for the eliminated DOF on the other arm.

Chapter 6

The code

6.1 Introduction

This chapter gives an overview of the C++ code that went into the implementation of the VES. It introduces the class libraries which realize all the algebra, and the basic data structures for bodies, constraints, and forces. The system can be expanded on that level through the use of derivation from a few abstract base classes. Examples of this will be given. The other major part of the algorithm that is documented here with the help of actual program text, is the recursive constraint propagation algorithm. The documentation for the front end parser can be found at the end of this chapter. The actual files have been slightly edited from the ones actually used, wherever this increased readability. For example many inline functions defined in the class definition itself have been taken out.

Historical note: The class libraries were first implemented using AT&T cfront version 1.2.1. I have only ported these to 2.0 without taking advantage of a number of new features. For example the streams are still only oldstyle *stream.h*.

6.2 Linear algebra classes

A number of primitive data types are required when using linear algebra for dynamics. These include 3 vectors which come as column vectors and row vectors. I have used generic types to define these. The same goes for spatial vectors. Matrices operate on these, but more important is the class of affine transforms. This class is defined to act on spatial vectors like a spatial transform and on 3 vectors like an ordinary affine transform. This representation is possible due to the fact that spatial transformations follow the same rules for concatenation as do affine transformations. Not only does this allow for increased economy in terms of storage but also time, since executing a spatial transformation concatenation requires 6 times as many floating point operations as does the associated affine transforma-

tion concatenation (see [19, page 144]). The actual code that executes these operations is straightforward and thus not included here. It is interesting to note, though that I found the code to be considerably faster on the HP9000/800 when unrolling each and every loop.

6.2.1 Generic class of 3 vectors

This generic class is instantiated as **Cvector** for column vectors and **Rvector** for row vectors. I chose this distinction to use the compiler to flag errors such as crossing a row and a column vector, or multiplying a column vector by a matrix (as opposed to multiplying a matrix by a column vector).

```

class <T>vector{
protected:
    double  vec[3];
public:
    <T>vector() {}
    <T>vector( const int ) { vec[0] = vec[1] = vec[2] = 0.; }
    // useful for fortran
    <T>vector( const double *ar )
        { vec[0] = ar[0]; vec[1] = ar[1]; vec[2] = ar[2]; }
    <T>vector( const double, const double, const double );
    <T>vector( const <T>vector& );
    <T>vector& operator=( const <T>vector& );

    friend  ostream& operator<<( ostream&, const <T>vector& );

    <T>vector  operator-( const <T>vector& ) const;
    <T>vector  operator-() const;
    <T>vector  operator+( const <T>vector& ) const;
    <T>vector  operator*( const double ) const;
    friend  <T>vector  operator*( const double, const <T>vector& );
    <T>vector  operator/( const double ) const;

    <T>vector& operator-=( const <T>vector& );
    <T>vector& operator+=( const <T>vector& );
    <T>vector& operator*=( const double );
    <T>vector& operator/=( const double );

    // takes the cross product with its argument
    <T>vector  cross( const <T>vector& ) const;
    // takes the dot product with its argument
    double  T( const <T>vector& ) const;

```

```

// returns a unitized vector
<T>vector dir() const;
// returns its length
double len() const;
// could have indexing protection
double& operator[]( const int i ) const { return vec[i]; }
};

```

6.2.2 Generic class of spatial vectors

These vectors too, come as **Csvector** for column spatial vectors and as **Rsvector** for row spatial vectors.

```

class <T>svector{
protected:
    <T>vector upper, lower;
public:
    <T>svector() {}
    // for a spatial vector initialized to zero
    <T>svector( const int );
    <T>svector( const <T>vector&, const <T>vector& );
    <T>svector( const double* );
    <T>svector( const double*, const double* );
    <T>svector( const <T>svector& );
    <T>svector( const double, const double, const double,
        const double, const double, const double );
    <T>svector& operator=( const <T>svector& );

friend ostream& operator<<( ostream&, const <T>svector& );

    <T>svector operator-( const <T>svector& ) const;
    <T>svector operator-() const;
    <T>svector operator+( const <T>svector& ) const;
    <T>svector operator*( const double ) const;
friend <T>svector operator*( const double, const <T>svector& );
    <T>svector operator/( const double ) const;

    <T>svector& operator-=( const <T>svector& );
    <T>svector& operator+=( const <T>svector& );
    <T>svector& operator*=( const double );
    <T>svector& operator/=( const double );

```

```

// spatial cross product with its argument
<T>svector cross( const <T>svector& ) const;
// ordinary dot product with its argument
double T( const <T>svector& ) const;
// spatial dot product with its argument
double S( const <T>svector& ) const;
// returns the index of the absolutely largest element
// starting at the (possibly) specified index
int max( int = 0 ) const;
// to get the individual sub vectors
<T>vector& up() const { return upper; }
<T>vector& low() const { return lower; }
// for indices 0 — 5, could error check
double& operator[( const int i ) const
    { return ( i < 3 ? upper[i] : lower[i - 3] ); }
};

```

6.2.3 Orientations as quaternions

In order to facilitate using the most efficient representation for rotations given the context I not only built a quaternion class but various cast operators to and from matrices as well as the exponential map and its inverse (logarithm). This allows the user of this class to stay fairly detached from the various actual representations of rotations. Depending on context for example it might be much more intuitive to specify a rotation by an axis vector scaled by the amount of rotation. An exponentiation of the vector followed by an assignment to a matrix will do the right thing in a transparent fashion.

```

class quaternion : private Cvector{
protected:
    // actually cos( theta / 2 )
    double s;
public:
    quaternion() : Cvector() {}
    // identity
    quaternion( const int ) : Cvector( 0 ), s( 1.0 ) {}
    quaternion( const double, const double, const double, const double );
    quaternion( const double, const Cvector& );
    quaternion( const quaternion& );
    // initialization and cast operator
    quaternion( const matrix& );
    quaternion& operator=( const quaternion& );
    // assignment of matrix to quaternion

```

```

quaternion& operator=( const matrix& );

friend ostream& operator<<( ostream&, const quaternion& );

quaternion operator*( const quaternion& ) const; 20
quaternion& operator*=( const quaternion& );
// quaternion multiplication with the column 3 vector
// treated as a quaternion with zero scalar part
quaternion operator*( const Cvector& ) const;
quaternion& operator*=( const Cvector& );
quaternion operator/( const quaternion& ) const;

quaternion operator-( const quaternion& ) const;
quaternion operator+( const quaternion& ) const;
quaternion operator*( const double ) const; 30
friend quaternion operator*( const double, const quaternion& );
quaternion operator/( const double ) const;

quaternion& operator--=( const quaternion& );
quaternion& operator+=( const quaternion& );
quaternion& operator*=( const double )
quaternion& operator/=( const double );

// returns a unitized quaternion
quaternion dir() const; 40
// dot product with its argument
double dot( const quaternion& q ) const;
// returns the length
double len() const { return sqrt( dot( *this ) ); }
// inverts a quaternion without regard to it being unitized or not
quaternion inv() const;
// rotate a column vector by the rotation parameterized by the quaternion
Cvector rot( const Cvector& ) const;
// like taking the transpose of the matrix associated with
// this quaternion; actually conjugation 50
quaternion T() const { return quaternion( s, Cvector::operator-( ) ); }
// as if it was a four vector
double& operator[( const int i )
    { return ( ( i ) ? Cvector::operator[(i - 1) : s ]; }
// member selection
double& w() const { return s; }

```

```

    const Cvector& v() const    { return *this; }
};

```

```

// exponentiation
quaternion exp( const Cvector& );
// inverse of exponentiation
Cvector log( const quaternion& );

```

60

6.2.4 Matrix class for 3 by 3 transformations

This class does allow for non-orthogonal matrices, however I currently only use it to hold orthogonal transformations (rotations). Note that the matrix class could just as well have been the quaternions, since in my system they are able hold all the information (rotations) needed.

```

class matrix{
protected:
    double  mat[9];
public:
    matrix() {}
    // 0 for the zero matrix
    // 1 for the identity matrix
    matrix( const int );
    matrix( const double, const double, const double,
           const double, const double, const double,
           const double, const double, const double );
    matrix( const matrix& );
    // initialization with quaternion as well
    // as cast operator from quaternion to matrix
    matrix( const quaternion& );
    matrix& operator=( const matrix& );
    matrix& operator=( const quaternion& );

    friend ostream& operator<<( ostream&, const matrix& );

    matrix operator-( const matrix& ) const;
    matrix operator-() const;
    matrix operator+( const matrix& ) const;
    matrix operator/( const double ) const;
    matrix operator*( const matrix& ) const;
    matrix operator*( const double ) const;

```

10

20

```

    matrix& operator--( const matrix& );
    matrix& operator+=( const matrix& );
    matrix& operator*=( const double );
    matrix& operator*=( const matrix& );
    matrix& operator/=( const double );

    // returns the transpose of itself
    matrix T() const;
    // multiplies a column vector on the right
    Cvector operator*( const Cvector& ) const;
    // transpose of itself times column vector
    Cvector T( const Cvector& ) const;
    // could do error checking
    double& operator()( const int r, const int c ) const
        { return mat[r * 3 + c]; }
};

// matrix times the cross operator arising from the vector argument
matrix cross( const matrix&, const Cvector& );
// cross operator arising from the vector argument times matrix
matrix cross( const Cvector&, const matrix& );

```

6.2.5 Class of rigid body motions

A translation vector and a rotation specify a rigid body motion. In this case the rotation is encoded as a matrix. The original design used matrices, however it could just as well be logarithms of rotations. It would be interesting to consider carefully the operations counts involved for the current algorithm when replacing the matrix member of affine transformations by a quaternion. As pointed out above spatial transformations follow the same rules as affine transformations, but act differently on spatial vectors. Hence there is no separate spatial transformation class, but rather variously defined multiplication operators.

```

class affine{
protected:
    Cvector t; // translation
    matrix r; // rotation
public:
    affine() {}
    // 0 for the zero affine transform
    // 1 for the identity affine transform
    affine( const int );
    affine( const Cvector&, const matrix& );

```

```

    affine( const affine& );
    affine& operator=( const affine& x );

    friend ostream& operator<<( ostream&, const affine& );

    // concatenation
    affine operator*( const affine& ) const;
    affine& operator*=( const affine& );
    smatrix operator*( const smatrix& ) const;
    // affine transform of column 3 vector 20
    Cvector operator*( const Cvector& ) const;
    // spatial transform of column spatial vector
    Cvector operator*( const Cvector& ) const;
    // returns the inverse for an affine transform
    affine inv() const;
    // returns the inverse (spatial transpose) of a spatial transform
    affine S() const { return inv(); }
    // apply the inverse affine transform to a column 3 vector
    // without actually creating the inverse
    Cvector inv( const Cvector& ) const; 30
    // spatial transpose of spatial transform times spatial column
    // vector, again without actually forming the transpose
    Cvector S( const Cvector& c ) const;
    // the same for the ordinary transpose
    Cvector T( const Cvector& c ) const;
    // selection of members
    Cvector& trans() const { return t; }
    matrix& rot() const { return r; }
};

```

For the definition of `smatrix` see below.

6.2.6 Variable length vectors for Ω -DOF vectors

For the recursive constraint propagation we need vectors whose length will be between 6 and 12. Since the length is bounded above by 12 these vectors are of fixed length 12 and maintain a member indicating the actual length. All operators are optimized for length 6, since that is almost always the length of an Ω vector. When conflicts as to length arise in a computation (which should never occur under ordinary circumstances), all operators will print a diagnostic message and assume the vector to be padded with zeros beyond its currently active range.

```
class omega{
```



```

private:
    double  vec[12];
    int     active;    // the entries currently valid
public:
    omega() : active( 6 ) {}
    // for a vector of length 6 initialized to zero
    omega( const int );
    omega( const double, const double, const double,
          const double, const double, const double );
    omega( const omega& );
    omega& operator=( const omega& );

friend    ostream& operator<<( ostream&, const omega& );

    omega operator-( const omega& ) const;
    omega operator-() const;
    omega operator+( const omega& ) const;
    omega operator*( const double ) const;
friend    omega operator*( const double, const omega& );
    omega operator/( const double ) const;

    omega& operator--=( const omega& );
    omega& operator+=( const omega& );
    omega& operator*=( const double );
    omega& operator/=( const double );

    // returns dot product with its argument
    double  T( const omega& ) const;
    // extend the currently active length by zero padding
    void fill( int, const int );
    // the actual current length (between 6 and 12)
    int length() const { return active; }
    // only for those who know what they are doing
    void set_active( const int a ) { active = a; }
    // index of largest element
    int max( int = 0 ) const;
    // when accessing a member beyond the current range,
    // the range is extended and padded with zeros;
    double& operator[]( const int i )
        { if( i >= active ) fill( active, i ); return vec[i]; }
    // access without check, only for those who know what they are doing

```

```

    double& ele( const int i ) const    { return vec[i]; }
};

```

6.2.7 Variable length matrices with 6 rows for \hat{M}

The various \hat{M} matrices used to parameterize the body state in terms of the Ω vectors of DOFs, will always have 6 rows but from 6 to 12 columns. Again it contains the maximal number of columns and a variable describing its current state. Notice that in this case the internal representation in column major form (transparent to the programmer) made these matrices well suited for using fortran SVD algorithms on them.

```

class smatrix{
private:
    Cvector cols[12];
    int active; // the columns currently valid
public:
    smatrix() : active( 6 ) {}
    // 0 for a 6 by 6 zero matrix
    // 1 for a 6 by 6 identity matrix
    smatrix( const int );
    smatrix( const Cvector&, const Cvector&, const Cvector&,
            const Cvector&, const Cvector&, const Cvector& );
    // upper left, upper right, lower left, lower right
    // of a 6 by 6 smatrix
    smatrix( const matrix&, const matrix&, const matrix&, const matrix& );
    smatrix( const smatrix& );
    smatrix& operator=( const smatrix& );

    friend ostream& operator<<( ostream&, const smatrix& );

    smatrix operator*( const smatrix& ) const;
    smatrix& operator*=( const smatrix& );
    smatrix operator+( const smatrix& ) const;
    smatrix& operator+=( const smatrix& );

    // a row spatial vector times a smatrix yields an omega vector
    friend omega operator*( const Rvector&, const smatrix& );
    // smatrix times an omega vector yields a column spatial vector
    Cvector operator*( const omega& ) const;
    // transpose smatrix times column spatial vector yields omega vector
    omega T( const Cvector& ) const;
    // extend the currently active set of columns with zero padding

```

```

void fill( int, const int );
// how many columns are in this smatrix (currently active)
int length() const { return active; }
// only if you know what you are doing
void set_active( const int a ) { active = a; }
// select a column with access check and possible
// zero padding for columns beyond the currently active range
Csvector& col( const int i )
    { if( i >= active ) fill( active, i + 1 ); return cols[i]; }
// no checking, only if you know what you are doing
Csvector& col_no_fill( const int i ) const { return cols[i]; }
// access an element, if beyond the currently
// active range, zero pad first
double& operator()( const int i, const int j )
    { if( j >= active ) fill( active, j + 1 ); return cols[j][i]; }
// no checking, only if you know what you are doing
double& ele( const int i, const int j ) const { return cols[j][i]; }
};

```

40

6.2.8 Inertial tensor in principal axes of inertia frame

A interesting change that I have not explored is to use a joint centered coordinate frame. In that case the inertial tensors would need to be represented by a matrix, a scalar, and a vector. We would only need to redefine the appropriate operators in this class. This would cut down on the number of full spatial transform concatenations in traversing the tree, when maintaining the local body coordinate frame.

```

class inertia : private Cvector{
protected:
    double mass;
public:
    inertia() : Cvector() {}
    // sort of an identity
    inertia( const int ) : Cvector( 1, 1, 1 ), mass( 1. ) {}
    inertia( const double, const double, const double, const double );
    inertia( const double, const Cvector& );
    inertia( const inertia& );
    inertia& operator=( const inertia& );

friend ostream& operator<<( ostream&, const inertia& );

    // inertia times spatial column vector

```

10

```

    Cvector operator*( const Cvector& cs ) const;
    // inverse inertia times spatial column vector
    Cvector inv( const Cvector& cs ) const;
    // inertia times smatrix
    smatrix operator*( const smatrix& ) const;
    // member selection
    double& m() const { return mass; }
    const Cvector& i() const { return *this; }
};

```

20

6.3 Forces

In order to facilitate the extensibility of the system forces are defined as derivations from an abstract class. The base class for all forces defines a number of virtual functions that need to be redefined for any actual force class. This design has the advantage that a force can maintain whatever state it needs in a transparent fashion and various pieces of the code that use forces need not know anything about the actual computations involved. This code was written when pure virtual functions were not available yet. Hence the functions in the base class print a diagnostic should they ever get executed.

```

class Force{
protected:
    char nomen[64];
    int active;
public:
    Force() {}
    Force( const char* );
    Force( const Force& );
    Force& operator=( const Force& );

friend ostream& operator<<( ostream&, const Force& );

    // += its contribution to the body force field
    virtual void apply( Body*, const double ) const;
    // used by the parser
    virtual int set_parameters( int, char**, char* );
    // for commands such as "print all forces"
    virtual ostream& print( ostream& ) const;
    // to communicate the state to the graphics front end
    virtual void draw( Body* ) const;

```

10

20

```

    char *name() const { return nomen; }
    int  toggle()  { if( active ) active = 0; else active = 1; return active; }
    int  state() const { return active; }
};

```

Example of a derived class

To give a better understanding of how these functions are realized in a particular force I will give the example of the derived class *motor*.

```

class MForce : public Force{
private:
    matrix    E;    // orientation of the local frame
    Cvector  loc;  // where in the parent body does it act
    Cvector  dir;  // the force itself
public:
    MForce();
    MForce( const char* );
    MForce( const MForce& );

    MForce& operator=( const MForce& );

    virtual void apply( Body*, const double ) const;
    virtual int  set_parameters( int, char**, char* );
    virtual ostream& print( ostream& ) const;
};

void
MForce::apply( Body *b, const double ) const
{
    if( state() ){
        b->cforce() += affine( loc, E ) * dir;
    }
}

char  motusage[] =
    "\t\t-f %F %F %F %F %F %F (the actual force applied)\n\
\t\t-t %F %F %F (where to apply it in local space)\n\
\t\t-E %F %F %F %F (a quaternions specification of orientation)\n";

int
MForce::set_parameters( int argc, char **argv, char *usage )

```

```

{
    Cvector newloc = loc;
    Cvector newdir = dir;
    quaternion newE = E; // notice implicit conversion matrix —> quaternion

    // scan the input
    int i = args( 0, argc, argv,
        "-f %F %F %F %F %F %F",
        &( newdir[0] ), &( newdir[1] ), &( newdir[2] ),
        &( newdir[3] ), &( newdir[4] ), &( newdir[5] ),
        "-t %F %F %F",
        &( newloc[0] ), &( newloc[1] ), &( newloc[2] ),
        "-E %F %F %F %F",
        &( newE.w() ), &( newE.v()[0] ), &( newE.v()[1] ), &( newE.v()[2] ),
        ( char* )0 );

    if( i < 0 ){
        cerr << usage << motusage;
    }else{
        E = matrix( newE );
        dir = newdir;
        loc = newloc;
        integrator_dirty = 1; // restart the integrator
    }
    return i;
}

```

The “set parameter” functions is noteworthy. By making it virtual it is possible to the front end parser to be ignorant as to what types of arguments an individual force takes. Rather it just calls the member function in a transparent way.

6.4 Support provided by other libraries

The VES code takes advantage of the capabilities of a set of libraries which include *btools* and *bobjs* for the parser, *args* for argument option scanning, *lsode* for the integration, and *linpack* and *blas* for the SVD code as well as for *lsode*.

6.5 The recursion

In order to give an impression of how the above classes have helped simplify the code I include here a slightly edited version of the core functions which execute the Lathrop

recursion. The last function *lathrop* of the following segment of code is the actual entry point.

```

// compute all the starred intermediate variables, also some body local
// variables which are needed for the down recursion
void
Body::compute_star()
{
    // here are the body local variables
    // first the matrix which expresses the constraint introduced by the joint
    sM() = S( s() ) * Mf();
    // get the element which is largest in the absolut sense
    j_offset() = sM().max();
    scale() = ( Q() - s().S( kf() ) ) / sM()[j_offset()];
    // normalize
    sM() /= sM()[j_offset()];
    // and here are the starred M matrices and kappa vectors
    // accelerations
    for( int i = 0; i < sM().length(); i++ ){
        Mastar().col( i ) = Ma().col( i ) - sM()[i] * Ma().col( j_offset() );
    }
    // we also subtract the joint axis right away
    Mastar().col( j_offset() ) -= s();

    kstar() = Ma().col( j_offset() ) * scale();
    kstar() += ka();
    kstar() -= vel().cross( s() ) * qdot();

    // forces
    for( i = 0; i < sM().length(); i++ ){
        Mfstar().col( i ) = Mf().col( i ) - sM()[i] * Mf().col( j_offset() );
    }
    kfstar() = Mf().col( j_offset() ) * scale();
    kfstar() += kf();
}

void
Body::acel_up( Body *child )
{
    // percolate up, the recursive step
    Ma() = child->pXc() * child->Mastar();
    ka() = child->pXc() * child->kstar();
}

```

```

}
40

void
Body::force_up( Body *child )
{
     $Mf() = I() * Ma();$ 
    if( child ){
         $Mf() += child->pXc() * child->Mfstar();$ 
    }else{
        // if we are at the end of a chain we need to incorporate the
        // endpoint constraint which is stored in Mfstar directly
         $Mf() += Mfstar();$ 
        50
    }

     $kf() = I() * ka();$ 
     $kf() += pv();$ 
    if( child ){
         $kf() += child->pXc() * child->kfstar();$ 
    }else{
        // if we are at the end of a chain we need to incorporate the
        // endpoint constraint which is stored in kfstar directly
         $kf() += kfstar();$ 
        60
    }
}

void
Body::extract_dof( omega& om )
{
    // the j-th entry was the joint acceleration
     $qddot() = om[j\_offset()];$ 
    // the next omega is the same as the current one but for the j-th entry
     $om[j\_offset()] = 0.;$ 
     $om[j\_offset()] = scale() - sM().T( om );$ 
    70
}

void
decompose_Ma( Body *b )
{
     $smatrix\& ma = b->Ma();$ 
    // go for the svd:  $ma = u * d * v^T$ 
    const int cols = ma.length();
    80
}

```



```

// linpack SVD function
dsvdc( &ma.ele(0,0), ldm, rows, cols, &b->svalues().ele(0),
      &b->evalues().ele(0), &ma.ele(0,0), ldu, &( b->v()[0] ), ldv,
      scratch, job, info );

// let's see how many good singular values we have
if( info > 0 ){
  cerr << "decompose_Ma: error in dsvdc(): info " << info << "\n";
}

// svalues[0] is the largest singular value (they are all positive!)
const double limit = 1e-10 * b->svalues()[0];
b->no_svalues() = 0;
int& ns = b->no_svalues();
while( limit < b->svalues()[ns] ){ ns++; }
}

smatrix
change_basis( const smatrix& m, const double *v )
{
  smatrix res( 0 );
  res.fill( res.length(), m.length() );
  // since we are dealing with fortran the matrix v is column major
  for( int i = 0; i < res.length(); i++ ){
    for( int j = 0; j < res.length(); j++ ){
      res.col_no_fill( i ) += m.col_no_fill( j ) * v[ldv * i + j];
    }
  }
  return res;
}

smatrix
dyad( const smatrix& a, const omega *r, const int n )
{
  smatrix res( 0 );
  res.fill( res.length(), r[0].length() );
  // a sum of dyads, really
  for( int i = 0; i < n; i++ ){
    for( int c = 0; c < r[i].length(); c++ ){
      res.col_no_fill( c ) += a.col_no_fill( i ) * r[i].ele( c );
    }
  }
}

```

90

100

110

120

```

    }
    return res;
}

// the function which reduces the big branching matrix
int
eliminate( Body *b1, Body *bj )
{
    // we are processing the j-th row
    Cvector& ka = bj->ka();
    ka = b1->ka() - bj->ka(); // build the constant term
    smatrix& ma1 = b1->Ma(); // the matrix in the first column
    smatrix& maj = bj->Ma(); // the matrix in the jth column
    decompose_Ma( bj ); // do a SVD of maj
    // apply the partial inversion to ma1 and ka: svalues^-1 * u^T ( ma1 + ka )
    for( int i = 0; i < bj->no_svalues(); i++ ){
        bj->res( i ) = ma1.T( maj.col( i ) ) / bj->svalues()[i];
        bj->resk( i ) = maj.col( i ).T( ka ) / bj->svalues()[i];
    }
    smatrix& aj = bj->Mf(); // the j-th matrix in the bottom row
    // bring the child's Mfstar up, and express it in the new
    // basis of right singular vectors
    aj = change_basis( bj->pXc() * bj->Mfstar(), bj->v() );
    // the constant term of the last row of the big matrix
    Cvector& kf = b1->kf();
    // and the same for kfstar which will be added into kf
    kf += bj->pXc() * bj->kfstar();
    // since the first no_svalues entries of the changed basis
    // omegaj are already available as a function of omega1
    // we'll add that contribution of A_1
    b1->Mf() += dyad( aj, bj->res(), bj->no_svalues() );
    for( i = 0; i < bj->no_svalues(); i++ ){
        kf += aj.col( i ) * bj->resk( i );
    }
    // return the number of left over dofs
    // which are not constrained by this row of eq 30
    return maj.length() - bj->no_svalues();
}

// we are down to 12 rows but might have many columns left

```

130

140

150

160

```

// this will do the final removal of unnecessary DOFs
void
col_reduce( Body *p, int total_cols )
{
    // we are now down to 12 rows and possibly tons of
    // columns from all the bodies;
    if( total_cols == 6 ){
        // nothing to do
        p->Ma() = p->child()->Ma();
        p->Mf() = p->child()->Mf();
        p->ka() = p->child()->ka();
        p->kf() = p->child()->kf();
        // indicate this for later
        p->no_svalues() = -1;
        return;
    }

    // we always have the columns corresponding to omega1 in this final rank
    // reduction we'll start out with considering the first 12 by length()
    // matrix due to the first body
    Body *b1 = p->child();
    smatrix& ma1 = b1->Ma();
    smatrix& a1 = b1->Mf();
    int i, j, k, l;
    for( i = 0; i < ma1.length(); i++ ){
        // we have two block rows of 6 rows each
        for( j = 0; j < 6; j++ ){
            // matreduce is a global array
            matreduce[i * ldB + j] = ma1.ele(j,i);
            matreduce[i * ldB + 6 + j] = a1.ele(j,i);
        }
    }
    total_cols -= i;
    decompose_Ma( b1 );
    // maybe there are more, but usually there are not...
    if( total_cols ){
        // get the others in order, pack them...
        int no_ch = p->no_children();
        for( j = 1; j < no_ch; j++ ){
            Body *bj = p->child( j );
            // start after the svalues

```

170

180

190

200

```

        for( k = bj->no_svalues(); k < bj->Ma().length();
            k++, i++, total_cols-- ){
            // the top 6 rows are always zero now
            for( l = 0; l < 6; l++ ){
                matreduce[i * ldB + l] = 0.;
                matreduce[i * ldB + 6 + l] = bj->Mf().ele(l,k);
            }
        }
    }
    if( total_cols ){
        // this better be zero now...
        cerr << "col_reduce: something wrong with looping " <<
            total_cols << "\n";
    }
}

int job = 20; // we don't care about the right singular values
dsdvc( matreduce, ldB, ldB, i, &p->svalues().ele(0), &p->evalues().ele(0),
        matreduce, ldB, scratch, ldv, scratch, job, info );

// let's see how many good singular values we have
if( info > 0 ){
    cerr << "error in dsdvc(): info " << info << "\n";
}
const double limit = 1e-10 * p->svalues()[0];
p->no_svalues() = 0;
int& ns = p->no_svalues();
while( limit < p->svalues()[ns] )    ns++;

// for sanity we will always leave with at least
// 6 columns
if( ns < 6 ){
    i = ns;
    // pad with zero columns
    p->Ma().fill( ns, 6 );
    p->Mf().fill( ns, 6 );
}
else{
    p->Ma().set_active( ns );
    p->Mf().set_active( ns );
}

```

210

220

230

240

```

// finally build Ma and Mf for p
for( i = 0; i < ns; i++ ){
    for( j = 0; j < 6; j++ ){
        // load Ma and Mf with u * D
        p->Ma().ele(j,i) = matreduce[i * ldB + j] * p->svalues()[i];
        p->Mf().ele(j,i) = matreduce[i * ldB + 6 + j] * p->svalues()[i]; 250
    }
}
p->ka() = b1->ka();
p->kf() = b1->kf();
}

// take a string of bodies and propagate all constraints from the end
// up to the beginning
void
branch_up( Body *const start, Body *body ) 260
{
    // compute the final M matrices and kappa vectors from their starred brethren
    while( start != body ){
        // now compute the intermediate starred quantities
        body->compute_star();
        body->parent()->accel_up( body );
        body->parent()->force_up( body );
        body = body->parent();
    }
} 270

// to do branched kinematic trees
void
tree_up( Body *body )
{
    Body *const start = body;
    // find the first that has more then one child
    while( body ){
        int sib = body->no_children();
        if( sib == 1 ){
            // one child, keep looking for the end of this string
            body = body->child();
        }else{
            if( sib > 1 ){
                int total_cols = 0;
            }
        }
    }
} 280

```

```

// more then one child, recurse
for( int i = 0; i < sib; i++ ){
    Body *sibling = body->child( i );
    // bring the child chains up to one body
    // below this body via recursion
    tree_up( sibling );
    sibling->compute_star(); // get the starred quantities
    // lift them up from the child
    sibling->Ma() = sibling->pXc() * sibling->Mastar();
    sibling->ka() = sibling->pXc() * sibling->kastar();
    // Now bring Ma for 1 <= i < sib to diagonal form
    if( i ){
        // this will simultaneously do the current row
        // in the big matrix and the last row of that matrix
        total_cols += eliminate( body->child(), sibling );
    }else{
        sibling->Mf() = body->I() * sibling->Ma();
        sibling->Mf() += sibling->pXc() * sibling->Mfstar();
        sibling->kf() = body->I() * sibling->ka();
        sibling->kf() += sibling->pXc() * sibling->kfstar();
        sibling->kf() += body->pv();
        total_cols = sibling->Mf().length();
    }
}
// now build the actual Ma, ka, Mf, kf used for 'body'
// do the final rank reduction
col_reduce( body, total_cols );
}else{ // sib == 0
    body->force_up( body->child() );
}
branch_up( start, body ); // move everything up
body = NULL; // terminate the while
}
}
}

omega
partial_invert( Body *b, const Cvector& rhs )
{
    omega res( 0 );
    // make it long enough if necessary

```

290

300

310

320

```

    res.fill( res.length(), b->Ma().length() );
    // compute the solution to  $v^T res = d^{-1} * u^T rhs$ 
    for( int i = 0; i < b->no_svalues(); i++ ){
        double coeff = b->Ma().col( i ).T( rhs ) / b->svalues()[i];
        for( int j = 0; j < res.length(); j++ ){
            res.ele(j) += coeff * b->v()[i * ldv + j];
        }
    }
    return res;
}

void
tree_down( Body *body, omega& om )
{
    Body *start = body;
    while( body ){
        int sib = body->no_children();
        if( sib == 1 ){
            body = body->child();
            body->extract_dof( om );
        }else{
            if( sib > 1 ){
                Body *sibling = body->child();
                // with om as we have it right now we have
                // body->acel() = body->Ma() * om + body->ka()
                // = body->child()->Ma() * om1 + body->child()->ka()
                const Cvector Ma_om1 = body->Ma() * om;
                omega om1;
                if( body->no_svalues() == -1 ){
                    om1 = om; // we never did a SVD in the final col_reduce
                    sibling->extract_dof( om1 );
                    tree_down( sibling, om1 );
                }else{
                    // now we exploit the fact that we have a SVD of sibling->Ma()
                    om1 = partial_invert( sibling, Ma_om1 );
                    sibling->extract_dof( om1 );
                    tree_down( sibling, om1 );
                }
            }
            // for all children we have that their respective omega can be
            // gotten from the fact that child->Ma() * omegaj =
            // Ma_om1 + child->ka() (this of course takes advantage of the fact

```

```

// that child->ka() = b1->ka() - bj->ka() !) since each child
// holds a SVD of its Ma() we now recover omegaj easily
for( int i = 1; i < sib; i++ ){
    sibling = body->child( i );
    om1 = partial_invert( sibling, Ma_om1 + sibling->ka() );
    sibling->extract_dof( om1 );
    tree_down( sibling, om1 );
}
}
body = NULL;
}
}
}
}
}

static  omega om;
static  omega om_bar;
static  omega root_acel;
static  omega root_force;
// these are the ones we actually solve for
static  omega *x[4] = { &om, &root_acel, &om_bar, &root_force };
static  smatrix *m[4] = { NULL, NULL, NULL, NULL };
static  Cvector *b[4] = { NULL, NULL, NULL, NULL };
static  const Cvector zero = 0;

void
lathrop( BodyList& body_list, double )
{
    Body *body;
    for( int i = 0, l = body_list.size(); i < l; i++ ){
        if( ( body = body_list.get_nth( i ) )->child() ){
            // this is a chain
            tree_up( body );
            // load the base constraints
            body->sfb_prep( m, x, b );
            solve_for_base( m, x, b );
            body->cacel() = root_acel;
            // back substitute
            tree_down( body, om );
        }
    }
}
}

```

370

380

390

400

6.6 Front end commands

At the current time the set of commands the front end parser understands consists of the following commands

- **ib** <localname> Instance a body with the unique name *localname*. Creates a generically initialized body on the list of bodies.
- **il** <rootname> <localname> Instance a link with the (unique) name *localname* as a daughter of *rootname*. The joint connecting the two is initialized to a rotational joint about the local z axis.
- **ilc** <body> <type> Instance a Lathrop constraint on *body* of type *type*. Currently *type* can be *free*, *nail*, and *move_to_point*.
- **pb** [*localname*] Print body with an optional named body as argument. Else it prints all bodies.
- **sb** <body> Set body parameter. The list of optional parameters and their arguments are
 1. **-vl** %F %F %F Velocity linear
 2. **-vr** %F %F %F Velocity rotational
 3. **-t** %F %F %F Translation
 4. **-E** %F %F %F %F Orientation as a quaternion (**E** is the letter used throughout the literature for orientation)
 5. **-in** %F %F %F Inertia along principal axes
 6. **-m** %F Mass
 7. **-if** <type> <name> Instance force of *type* with *name* to refer to it later, where *type* can be any of
 - (a) **gravity** A linear force that acts at the center of mass in the negative global z direction
 - (b) **motor** A spatial force that acts on a body with a local frame of its own
 - (c) **damping** A spatial viscous damping force which acts at the origin of the body local frame
 - (d) **support** A parallel spring/damper combination with a support point that it tries to keep from going below global $z == 0$
 - (e) **brake** A linear viscous damping force that has a local frame associated with it
 8. **-f** <name> Force acting on this body with *name*, toggling it on and off

- **sl** <parent> <child> Set parameters of a link. With the following optional arguments
 1. **-s** %F %F %F %F %F %F An explicit spatial joint axis for the proximal joint in the child (**s** is the notation used for spatial joint axes in spatial algebra)
 2. **-q** %F To set joint position (a DOF generally abbreviated with the letter **q**)
 3. **-qdot** %F The derivative of the joint variable
 4. **-pjt** %F %F %F Proximal joint translation in the child frame
 5. **-pje** %F %F %F %F Orientation of the joint axis in the child frame
 6. **-djt** %F %F %F %F Distal joint translation in the parent frame
 7. **-dje** %F %F %F %F Orientation of the joint axis in the parent frame
- **step** %d Take a specified (integer) multiple of the current step size forward in time, drawing after every step. The integrator internally takes whatever number of internal steps necessary to realize these.
- **rtime** Read **time** of the internal simulation clock
- **stime** Set **time** of the internal simulation clock
- **sacc** Set **accuracy** goal of the integrator. Has the optional arguments
 1. **-r** %F Relative accuracy
 2. **-a** %F Absolute accuracy
- **rstepsize** Read **stepsize** returns the current increment of time from one frame to the next
- **sstepsize** Set **stepsize** sets the stepsize between frames
- **rlsode** Read **lsode** prints statistics of the integrator such as
 1. last (internal) stepsize used
 2. next (internal) stepsize to attempt
 3. number of steps taken since the last restart
 4. number of calls to the dynamics algorithm since the last restart
 5. method order last used
 6. absolute accuracy goal
 7. relative accuracy goal
- **ir** <localname> <body1> <body2> Instance a rubber string, *localname*, between *body1* and *body2* with the optional arguments

1. **-k %F** The spring constant
 2. **-l %F** The rest length
 3. **-e %F** The coefficient of restitution (**e** is the letter used in the literature)
 4. **-lone %F %F %F** Location of attachment point in body **one**
 5. **-ltwo %F %F %F** Location of attachment point in body **two**
- **sf <body> <force>** Set force acting on *body* by *name*. This command takes optional parameters whose validity and interpretation is based on the type of the force.
 1. Gravity
 - (a) **-g %F** To set the gravitational constant
 2. Motor
 - (a) **-f %F %F %F %F %F %F** The spatial force to apply
 - (b) **-t %F %F %F** Where to apply in the *body*
 - (c) **-E %F %F %F %F** With what orientation
 3. Damping
 - (a) **-d %F** To set the damping constant
 4. Support
 - (a) **-d %F** To set the damping constant
 - (b) **-k %F** To set the spring constant
 - (c) **-s %F %F %F** To set the point of support in the *body* local frame
 5. Brake
 - (a) **-d %F** To set the damping constant
 - (b) **-t %F %F %F** To set the point of action
 - **pf [body [force]]** Print Force(s) associated with an optional *body*
 - **draw <bolio | corpus <filename> %d | null>** Select the convention for drawing objects used by the invoking program. (Currently supported are *bolio* [9], *corpus* [12], and *null*, i.e. no drawing at all.)
 - **?** and **help** to print usages messages for all commands available

Chapter 7

Results

7.1 Introduction

Over the past 2 months I have been designing various testcases and demonstration simulations to use and prove the capabilities of the VES. Some of these I will describe in a more detailed fashion in this chapter. I will also use this chance to discuss some of the things I learned when I did these simulations.

7.2 A tensegrity structure

In one of the scenarios of use I had given the example of the construction of a tensegrity structure. To actually build such a structure I implemented a force called *rubber*, which is modeled as a spring that does not exert a force when it becomes shorter than the rest length. It also incorporates a coefficient of restitution idea ¹, namely that the force exerted when the endpoints move towards each other is scaled down from what it would have been had the endpoints been moving away from each other. This way energy is effectively absorbed, allowing the tensegrity structure to come to a rest state.

With the above model there exists a jump discontinuity in the force applied when the relative velocity of the endpoints goes through zero. It turned out that the integrator was very sensitive to this discontinuity. Fixing this problem by restarting the integrator was not really an option since the worst such behavior occurred when the structure had assumed its final position. In that constellation a small bit of noise would continually move the structure back and forth over the discontinuity bringing the integrator to a halt. Using the idea of restarting the integrator to deal with this would have forced me to continually restart. Instead I used a smoothing polynomial of third order to remove the discontinuity. Even though the change is still rather steep the fact that it is now smooth solved the problem

¹This suggestion is due to Michael McKenna.

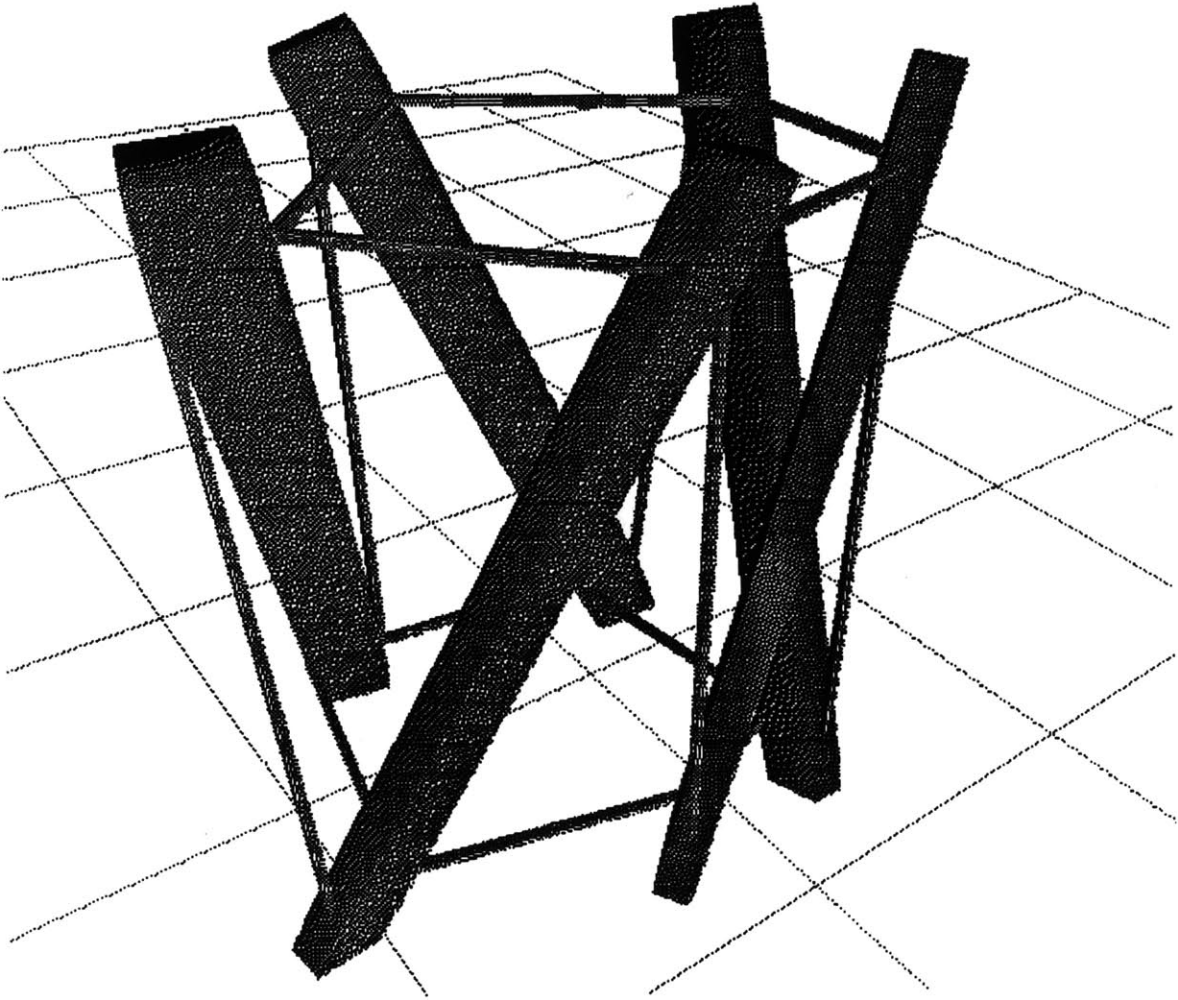


Figure 7-1: A model of a 5 stick tensegrity structure snapping together under the tension of springs. This image shows the structure after 0.5 seconds of simulation time.

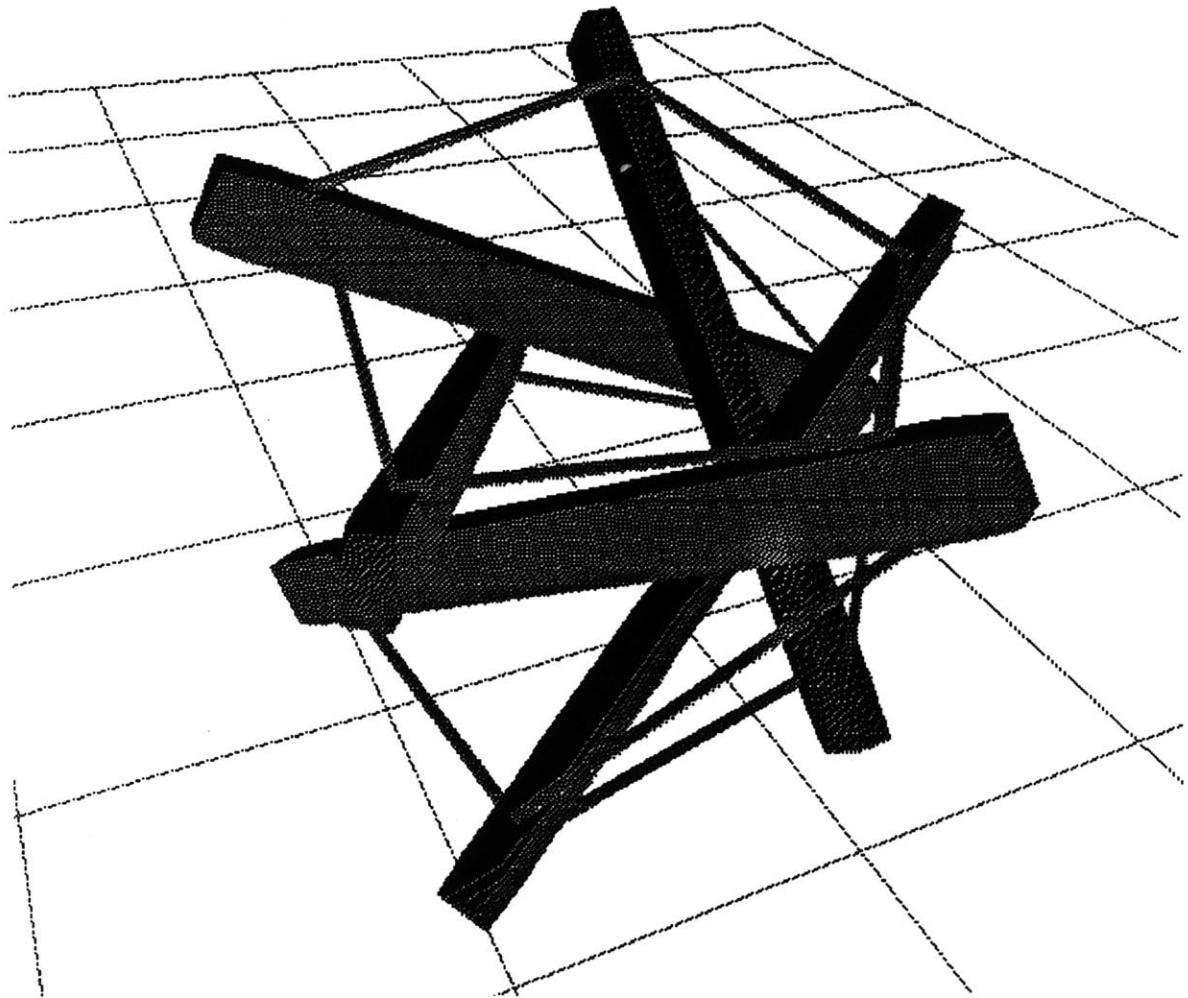


Figure 7-2: The same structure having reached its final state after 8 seconds of simulation time.

with the integrator.

The tensegrity structure that I actually built uses 5 sticks and 15 rubber strings to connect the sticks. The following set of commands instances the structure in an initial configuration.

```
;instance the bodies
ib stick0
ib stick1
ib stick2
ib stick3
ib stick4
;set inertia and initial position and orientation
sb stick0 -in 0.104167 8.35417 8.41667 -E .5 0 .5 0 -t 6 -15 10
sb stick1 -in 0.104167 8.35417 8.41667 -E .5 0 .5 0 -t 0 -12 10
sb stick2 -in 0.104167 8.35417 8.41667 -E .5 0 .5 0 -t -1 -17 10
sb stick3 -in 0.104167 8.35417 8.41667 -E .5 0 .5 0 -t 0 -18 10
sb stick4 -in 0.104167 8.35417 8.41667 -E .5 0 .5 0 -t 4 -18 10
;to make things settle faster instance damping for the bodies
sb stick0 -if damping damping
sb stick1 -if damping damping
sb stick2 -if damping damping
sb stick3 -if damping damping
sb stick4 -if damping damping
;set the damping constant
sf stick0 damping -d 10
sf stick1 damping -d 10
sf stick2 damping -d 10
sf stick3 damping -d 10
sf stick4 damping -d 10
;instance the rubber strings connecting the top
;spring constant 20, rest length 4, coefficient of restitution .1
ir stringB1 stick0 stick1 -lone -4 -.5 0 -ltwo -4 -.5 0 -k 20 -l 4 -e .1
ir stringB2 stick1 stick2 -lone -4 -.5 0 -ltwo -4 -.5 0 -k 20 -l 4 -e .1
ir stringB3 stick2 stick3 -lone -4 -.5 0 -ltwo -4 -.5 0 -k 20 -l 4 -e .1
ir stringB4 stick3 stick4 -lone -4 -.5 0 -ltwo -4 -.5 0 -k 20 -l 4 -e .1
ir stringB5 stick4 stick0 -lone -4 -.5 0 -ltwo -4 -.5 0 -k 20 -l 4 -e .1
;instance the strings connecting across the middle
ir stringM1 stick0 stick4 -lone -4 .5 0 -ltwo 4 -.5 0 -k 20 -l 4 -e .1
ir stringM2 stick1 stick0 -lone -4 .5 0 -ltwo 4 -.5 0 -k 20 -l 4 -e .1
ir stringM3 stick2 stick1 -lone -4 .5 0 -ltwo 4 -.5 0 -k 20 -l 4 -e .1
ir stringM4 stick3 stick2 -lone -4 .5 0 -ltwo 4 -.5 0 -k 20 -l 4 -e .1
ir stringM5 stick4 stick3 -lone -4 .5 0 -ltwo 4 -.5 0 -k 20 -l 4 -e .1
```

```

;instance the strings connecting the bottom
ir stringT1 stick4 stick0 -lone 4 .5 0 -ltwo 4 .5 0 -k 20 -l 4 -e .1
ir stringT2 stick0 stick1 -lone 4 .5 0 -ltwo 4 .5 0 -k 20 -l 4 -e .1
ir stringT3 stick1 stick2 -lone 4 .5 0 -ltwo 4 .5 0 -k 20 -l 4 -e .1
ir stringT4 stick2 stick3 -lone 4 .5 0 -ltwo 4 .5 0 -k 20 -l 4 -e .1
ir stringT5 stick3 stick4 -lone 4 .5 0 -ltwo 4 .5 0 -k 20 -l 4 -e .1

```

This assembly of bodies has 30 DOFs and 30 forces which are dependent on the states of the bodies. In the following table times are in seconds. “d” is the damping constant used.

rel. acc.	abs. acc.	d	fn. evals.	steps	fn. evals / step	total time	time / step
10^{-6}	10^{-7}	10	23337	14368	1.62	195	0.01357
10^{-6}	10^{-7}	1	16192	14992	1.08	165	0.01101
10^{-4}	10^{-3}	10	20949	11603	1.81	156	0.01344
10^{-4}	10^{-3}	1	16612	9245	1.80	125	0.01352

When changing the spring constants from 20 to 40 the following data results

rel. acc.	abs. acc.	d	fn. evals.	steps	fn. evals / step	total time	time / step
10^{-7}	10^{-6}	10	41947	40064	1.05	427	0.01066
10^{-7}	10^{-6}	1	34196	31725	1.08	347	0.01094
10^{-4}	10^{-3}	10	48726	26737	1.82	361	0.01350
10^{-4}	10^{-3}	1	38257	21147	1.81	285	0.01348

While lowering the accuracy requirements speeds up the simulation somewhat the main cause for small stepsizes in this case are the relatively numerous and strong springs. Also notice the ratio of function calls to steps which in all cases is lower then 2. It is possible to set the accuracy requirements so low (rel. acc $\geq 10^{-2}$) that the assembly goes unstable. However the integrator is mostly led by the modes that it detects, since for lesser accuracy requirements it still takes essentially the same number of function evaluations, but larger internal steps, increasing the ratio of function evaluations per step.

7.3 The policecar scene in “Grinning Evil Death”

The forthcoming movie “Grinning Evil Death” [24] contains a scene in which a police car can be seen driving down a street. The driver abruptly brakes the car as a consequence of which the car slides at a 100 degree angle into the crossing. The task I tried to solve was

to do the dynamic simulation for the car. To this end I modeled the mass of the car as a parallelepiped of 2000 kg. The dimensions of the car were taken directly from the animation placing the front wheels 1.95 m ahead and the back wheels 1.521 m behind the center of mass. The axle width was 1.794 m. The suspension was modeled as a spring damper combination. This is a primitive force in the VES called *support*. The spring constants were set to 50000 for all wheels and the damping constants to 2500 and 3500 for the front and back respectively. The car had a front wheel drive which was modeled as a linear force at the point of contact of the front wheels with the ground, oriented in the direction of the wheel (the primitive force *motor* with its local frame aligned with the wheel). Braking was accomplished with velocity damping at the point of contact of all four wheels with the ground. The brakes in the front were able to exert linear damping with a constant of 1000 and in the back with a constant of 500.

The “practice runs” for the “stunt” were performed by constructing a menu of actions. These included *motor toggle*, *brake toggle*, *turn steering*, *straighten steering*, *reset*. All of these would also print a time stamp when activated by a mouse click. This allowed me to reproduce later the best version. I then proceeded to go through a number of iterations, accelerating the car, turning the steering, and braking. Needless to say I overturned the car a few times. When I had a slide that was almost perfect as far as satisfying the final position requirement, I turned the actions for that run into a script. For this is used the time stamps printed on the console and “tweaked” the parameters slightly to make the braking action as dramatic as possible without having the car overturn.

Originally I had set the time intervals at which the VES would update the positions of the associated objects in bolio to 1/30th of a second. At this rate the graphical simulation was running at about 8 Hz. Most of this time however is due to communication and rendering overhead as evidenced by the fact that increasing the update time interval (which is distinct from the stepsizes that the integrator takes) to 1/4 second only lowered the speed to 6 Hz, effectively 1.5 times real time. This of course is no surprise since the model was very simple. The main point about this example is that it showed that for an animation task such as this film provided, the interactiveness of the simulator was good enough to allow for the necessary repeated trial and error runs.

7.4 Four ways to describe a simple linkage and some of the results

As a sample test and also a way to verify the code I simulated a simple 3 bar linkage 4 different ways. Consider 3 beams, each 10 m long. They are connected by a revolute joint at 4 m out from the center of mass with the joint axis perpendicular to the longitudinal axis of the beams. There are four distinct ways these can be treated when simulating them with the algorithm that I implemented. The linkage can be treated as a simple chain with the root at one end or the other, or as a body with two children, one to the left, one to the

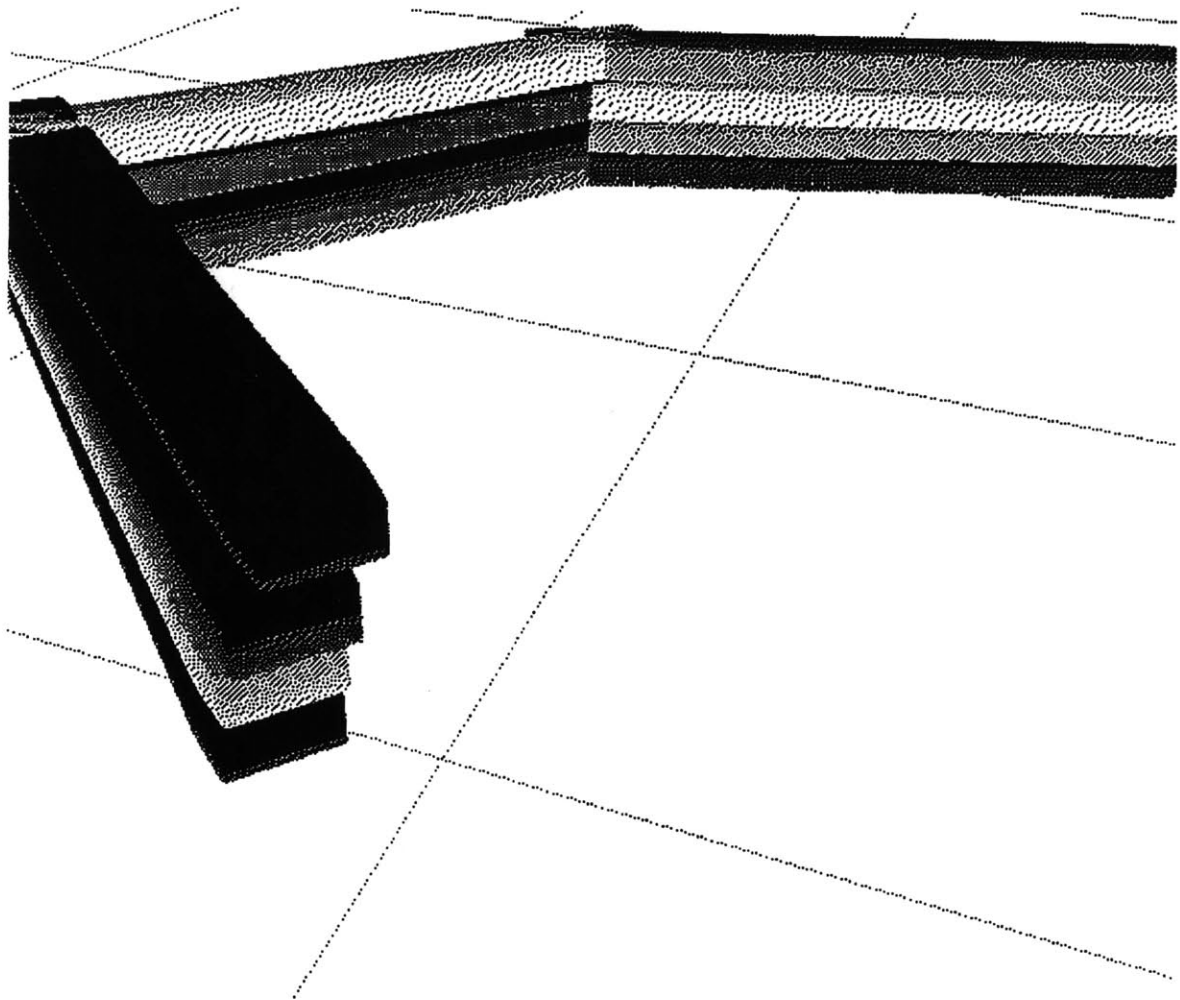


Figure 7-3: Four different parameterizations of a 3 bar linkage after 53 seconds of simulation time. The top linkage has noticeably drifted from its original alignment with the other three.

right. Since in the branching code I arbitrarily pick the first body on the list of children to represent the first column in the branching matrix, there are 2 distinct ways of simulating this case as well, the left child being first or second on the list of children (see 7-3). I started the simulation by having the linkage aligned with the x axis, the joint axes pointing up, and initially every body at rest but one of the endbodies, which was given an initial velocity about its joint axis of 0.1 radians per second.

When running the simulation it can be noticed that the 4 separate linkages start to drift apart after a few seconds of simulation time. Examination of the actual accelerations computed revealed that these initially only disagreed on the order of 10^{-16} to 10^{-14} , the best that can be expected given double precision arithmetic. However due to feedback these differences soon grow.

In all of the following simulations the requested local precision was 10^{-7} absolute and 10^{-6} relative. When only simulating the two linkages with the root in the middle and children to the left and right the velocities after one second were:

rotational velocity	linear velocity
$(0, 0, 3.58176 \cdot 10^{-4})$	$(1.40613 \cdot 10^{-2}, 1.031 \cdot 10^{-3}, 0)$
$(0, 0, 3.58294 \cdot 10^{-4})$	$(1.40613 \cdot 10^{-2}, 1.03091 \cdot 10^{-3}, 0)$

At this point it is important to note that the error estimation in the integrator uses a weighted l_2 norm, not an l_∞ norm. So rather than trying to keep the worst channel of the integration accurate to the requested degree, the integrator attempts to keep a weighted least squares estimation below the requested accuracy. This means for example that when introducing another body into the simulation that does nothing the integrator will believe the achieved accuracy is higher, because the extra body at rest incurs no errors whatsoever but increases the number of equations integrated. These are the results of the same simulation as above with such a null body included in the world:

rotational velocity	linear velocity
$(0, 0, 3.48003 \cdot 10^{-4})$	$(1.38485 \cdot 10^{-2}, 1.00062 \cdot 10^{-3}, 0)$
$(0, 0, 3.46133 \cdot 10^{-4})$	$(1.38486 \cdot 10^{-2}, 0.998151 \cdot 10^{-3}, 0)$

It comes as no surprise then that when I added one of the “root at one end” linkages to the simulation the values were again different. On the last run I simulated all four together with the following results for the velocity of the middle body:

rotational velocity	linear velocity
$(0, 0, 3.63908 \cdot 10^{-4})$	$(1.41807 \cdot 10^{-2}, 1.04783 \cdot 10^{-3}, 0)$
$(0, 0, 3.63802 \cdot 10^{-4})$	$(1.41805 \cdot 10^{-2}, 1.04814 \cdot 10^{-3}, 0)$
$(0, 0, 3.649 \cdot 10^{-4})$	$(1.41806 \cdot 10^{-2}, 1.04916 \cdot 10^{-3}, 0)$
$(0, 0, 3.649 \cdot 10^{-4})$	$(1.41734 \cdot 10^{-2}, 1.42664 \cdot 10^{-3}, 0)$

Notice that all of these simulations were run without any damping at all and I did leave one of them running for more than 9 simulation minutes. The bodies had not picked up any energy but had clearly drifted apart. It was also apparent that they were still following the same motion only separated in space. In other words locally they were following the same history, but not globally. Since there was no damping present, something one would not find in any simulation attempting to model the real world, these differences had time to accumulate out of floating arithmetic noise into noticeable differences. For any system modeling the real world with terms that lose energy these differences are likely not to

show up. These experiments do point out though that even under ideal conditions we can ultimately only hope to get behavior qualitatively correct (which is good enough).

From this set of experiments and many more along the same lines, varying different parameters, I have drawn the following conclusions. All ways of parameterizing a given linkage yield qualitatively the same simulation. The closer the center of mass of the linkage as a whole is to the root body the less aberration. Small changes in the time step history (see the introduction of the null body above) will result in quantitative changes but qualitatively the behavior is identical.

7.5 Endpoint constraints and their numerical behavior

Another set of experiments concerned the use of endpoint or root constraints that depended on reaction forces or other forces due to a geometric constraint. The setup was similar the the above mentioned experiments, only this time I used a 4 bar linkage with the root at one end. The immediate child of the root had a pure couple acting about its center of mass. This linkage was instanced 3 times, once with no geometric constraint on either end, once with the root immovably connected to the inertial frame, and once with the end-body immovably connected to the inertial frame (see 7-4 and 7-5). After 15 seconds of simulation time the displacements and velocities respectively of the bodies constrained to the inertial frame were:

rotational velocity	linear velocity	displacement
$(0, 0, -2.50395 \cdot 10^{-3})$	$(-3.33889 \cdot 10^{-4}, -2.50178 \cdot 10^{-4}, 0)$	$(-4.9 \cdot 10^{-3}, -3.5 \cdot 10^{-3}, 0)$
$(0, 0, 0)$	$(3.91406 \cdot 10^{-3}, 6.7218 \cdot 10^{-4}, 0)$	$(1.26 \cdot 10^{-2}, -1.46 \cdot 10^{-2}, 0)$

Again inspection of the actual values as they were computed showed that the initial accelerations that were constrained to be zero were only off by floating point arithmetic noise.

While I consider these results excellent from a numerical point of view (1 cm or less displacement out of a 40 m linkage after 15 seconds of simulation time), they do point out that in order to maintain constraints for long periods of time it is necessary to add feedback terms which penalize errors in the appropriate way. One such possibility is to add a weak damper/spring combination between the body locked to the inertial frame and the inertial frame itself.

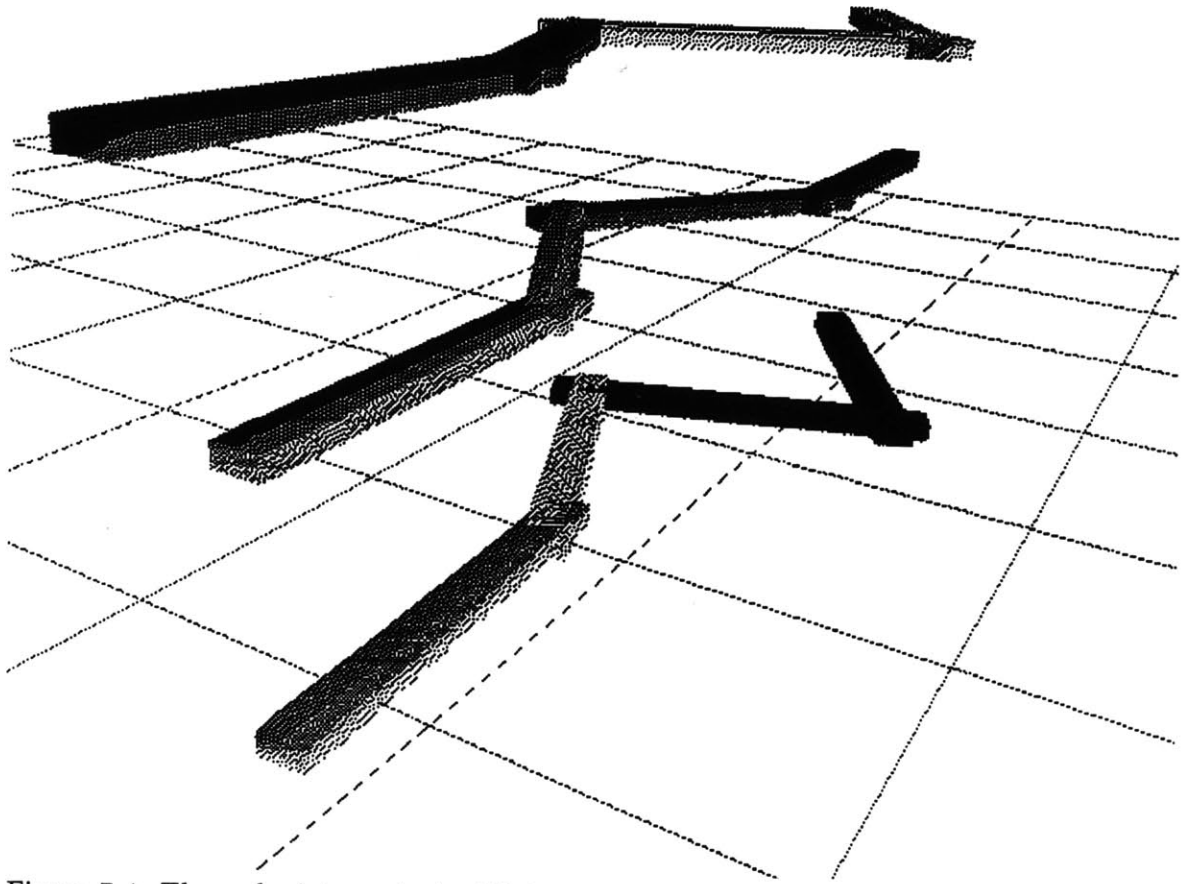


Figure 7-4: The endpoint constrained linkages after 5 seconds of simulation time. The root bodies are at the far end. The bottom linkage is moving freely while the middle linkage has a motionless root and the top linkage is constrained to have a motionless endlink.

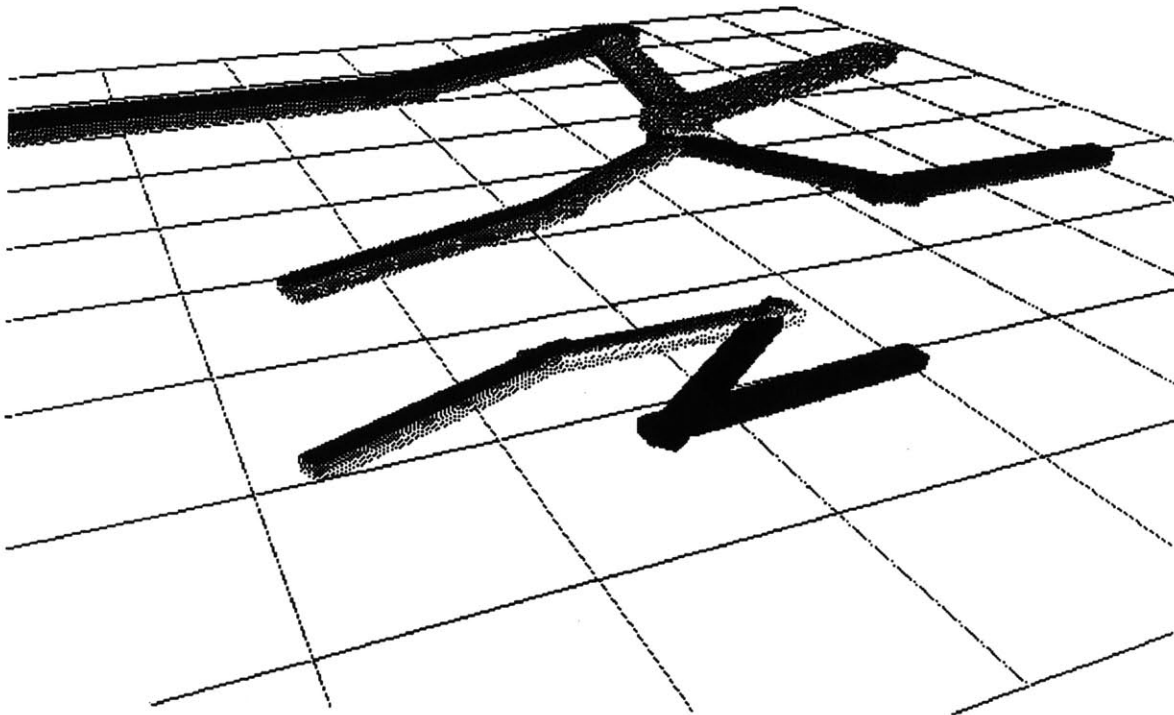


Figure 7-5: The same three linkages after 25 simulation seconds.

Chapter 8

Conclusion and future work

In this thesis I have presented a recursive linear time algorithm for the dynamic simulation of multibody systems in the presence of constraints. The algorithm as implemented is useful for interactive graphical simulation work and allows for a wide variety of effects to be modeled. The use of object oriented programming techniques and the C++ programming language has greatly facilitated the implementation without sacrificing much in terms of efficiency. During the implementation¹ I have filled in many of the details—such as the code to deal with branches—which only received cursory treatment in the original article. I also addressed a number of numerical issues and uncovered more that need to be addressed by future work in this area. Another contribution of this work is the extension of spatial algebra notation through an explicit representation of *spatial position* which is numerically advantageous.

Interesting areas for future work include the following

- The development of the algorithm was motivated purely by mathematical considerations. This leaves us with the \hat{M} matrices without understanding much of the structure of these matrices. I believe more structure can be extracted from the various \hat{M} matrices. For example consider an linkage with freely moving end bodies. In this case the algorithm should be equivalent to Featherstone’s articulated body method. This would mean that various expressions can be simplified or computed more efficiently. Another question is the invertability of the \hat{M} matrices. More understanding on this issue would allow for parameterizing the state of individual bodies with a smaller set of equations, possibly resulting in further efficiency gains.
- Featherstone [19] can do branches with exactly 6 DOFs. Hence an interesting hypothesis to examine would be to see if the B matrix of equation (5.41) must always be of rank exactly 6. This can not be argued from the mathematics of the equation alone,

¹To my knowledge, and that of Richard Lathrop, mine is the first actual implementation of the algorithm that he had proposed.

but possibly by finding the mapping to the articulated body method (so far all cases that I have studied have indeed been of rank 6).

- The relationship between motion and force constraints as expressed by the endpoint \hat{M} matrices is not clear and could be elucidated further using physical arguments rather than pure mathematical ones.
- The integrator in the current implementation was a modified version of LSODE [23] which is written in Fortran. It would be valuable to translate this code to C or better yet C++ to be able to take finer control over its inner workings. This would include
 - the accommodation of lists rather than arrays.
 - finer tuning of the error checking to treat the various channels of the integration differently, and actually separate the integration of each linkage in a simulation with several linkages.
 - accommodate an option for immediate return from an attempted evaluation of the dynamics to the top level, if an exception is raised (such as a collision).
- More work should be invested in analyzing closely the interaction between the integrator and the dynamics algorithm. Specifically more sophisticated techniques for dealing with discontinuities should be explored.
- Another interesting analysis would concern the condition of the matrices involved and possible approaches to preconditioning these to have better error behavior and thus help the integrator.
- Experiments with endpoint constraints have shown that errors can accumulate from floating arithmetic noise—due to feedback—to quite noticeable levels in very long simulations. While numerically no surprise and certainly in line it would be desirable to devise techniques that counteract the feedback. As mentioned a simple such a approach would be dampers and springs in parallel with the constraint to be maintained.
- It seems now clear to me that there is no need at all anymore for the linear algebra based constraint force approach, as the endpoint constraints satisfy all needs for which I had originally envisioned the use of constraint forces. If it is still felt necessary to include a stage which treats constraints as a set of equations whose roots are to be found, I would strongly encourage the implementation of such a stage using standard root finding techniques, which are much better conditioned than the Barzel [1] approach.
- More primitives, such as forces and joints should be incorporated into the implementation. The current set of joints are all one DOF joints. While multi DOF joints can be built from these, it would be a nice addition to the algorithm to accommodate

these joints directly by turning various scalar quantities into vector quantities (Q, q, \dot{q}, \ddot{q}). This does not change the algorithm but only impacts the code itself. It is hoped that such a change is actually very simple due to the use of data abstraction in the current implementation. The set of endpoint constraints should be extended as well. Many of these desirable new derived classes will only become apparent through the continued use of the system.

- The VES itself does not contain a front end aside from a rather terse command parser. Some tools already exist in bolio to use menus for invoking some of the commands of the VES. However the interaction could be improved by developing these tools for bolio further. Notice that the interactive front end was not part of the scope of this work.
- The C++ source code could be improved by taking advantage of new features of the latest language definition, most notably the *iostream* class, *virtual base classes*, *pure virtual* functions, and *multiple inheritance*.
- By far the most important and the most intriguing extension would be a linear time recursive collision response algorithm. To my knowledge there exists no such algorithm in the literature. The reason why I believe that it should be possible is the following. When we write down a matrix formulation for the dynamics of a given linkage the sparsity pattern in this matrix is purely a function of the connectivity of the linkage. The same holds true for the matrix associated with the preservation of momentum equations of the same linkage. Hence the Lathrop recursive algorithm corresponds to a certain traversal of the graph associated with the sparsity pattern of the matrix. The same traversal should also work for the preservation of momentum equations, giving a linear recursive algorithm for collision response.

Appendix A

Spatial Algebra

In [25], [19] Roy Featherstone introduced the use of spatial algebra for the analysis and modeling of newtonian dynamics. We will give a short introduction to its use and major equalities and refer the reader interested in a more thorough treatment to the original sources in [25], [19], and [26]

A.1 Line vectors

The six DOF's of a rigid body can be described as a point on the manifold $\mathbb{R}^3 \times S^3$. Differentiating spatial position we get elements in the tangent space to $\mathbb{R}^3 \times S^3$, namely in $\mathbb{R}^3 \times \mathbb{R}^3$. Differentiating once more for acceleration we stay in the latter space ($\mathbb{R}^3 \times \mathbb{R}^3$ has the structure of \mathbb{R}^6 and thus is isomorphic to its own tangent space). Thus velocities, accelerations and forces have the structure of a line which is parameterized uniquely by its so called Plücker pair, $(\mathbf{a}^T, \mathbf{a}_O^T)^T$

$$\begin{aligned} \mathbb{R}^3 \ni \mathbf{r} \in \text{line} \\ \iff (\mathbf{r} - \vec{OA}) \times \mathbf{a} &= \mathbf{0} \\ \iff \mathbf{r} \times \mathbf{a} &= \mathbf{a}_O \\ &\mathbf{a}_O = \vec{OA} \times \mathbf{a} \end{aligned} \tag{A.1}$$

where O is the origin of the current frame and A is a point on the line. This notation is closely related to screw calculus [26].

A.1.1 Spatial velocity and acceleration

We can now define spatial velocity $\hat{\mathbf{v}} = (\omega^T, v_O^T)^T$ and spatial acceleration $\hat{\mathbf{a}} = (\alpha^T, a_O^T)^T$.

Since we need to take cross products we define

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \times = \begin{pmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{pmatrix} \quad (\text{A.2})$$

with this now we can formulate the shifting rule

$$\hat{\mathbf{a}}_P = \begin{pmatrix} id & 0 \\ \vec{PO} \times & id \end{pmatrix} \hat{\mathbf{a}}_O \quad (\text{A.3})$$

To convince herself that this is the right definition, the reader should consider finding the linear velocity of a point P on a body given the body's linear and angular velocity at e.g. the center of mass O .

A.1.2 Change of coordinate frame

In order to execute coordinate changes we also need to define how to change the orientation of a spatial quantity. Again we motivate the definition via the properties we desire and the reader may convince himself that this is the correct definition by considering the spatial velocity of a body expressed in coordinate frame O and how to derive the appropriate linear and angular velocity in coordinate frame P

$$\hat{\mathbf{a}}_P = \begin{pmatrix} E & 0 \\ 0 & E \end{pmatrix} \begin{pmatrix} id & 0 \\ r \times^T & id \end{pmatrix} \hat{\mathbf{a}}_O = {}_P\hat{\chi}_O \hat{\mathbf{a}}_O \quad (\text{A.4})$$

where $r = \vec{OP}$ and E is the rotation matrix which takes us from the orientation of frame O to the orientation of frame P , i.e. its columns are the coordinates of the O frame expressed in the P frame.

A.1.3 Spatial transpose

We define the following notion of a transpose for spatial transforms such as the ones which arise from a change of coordinate frame

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^S = \begin{pmatrix} D^T & B^T \\ C^T & A^T \end{pmatrix} \quad (\text{A.5})$$

Notice that with this definition we have the identity

$${}_O\hat{\chi}_P = {}_P\hat{\chi}_O^{-1} = {}_P\hat{\chi}_O^S \quad (\text{A.6})$$

A.2 Differentiation in moving coordinates

Since the time behavior of a rigid body is a curve on our manifold $\mathfrak{R}^3 \times S^3$ we need to take this into account when differentiating. Let P be a possibly moving coordinate frame and O a stationary one then we get for $\hat{a} \in \mathfrak{R}^3 \times S^3$ arbitrary

$$\begin{aligned} D\hat{a} &= {}_P\hat{\chi}_O \frac{d}{dt}({}_O\hat{\chi}_P \hat{a}) \\ &= \frac{d}{dt}\hat{a} + {}_P\hat{\chi}_O \left(\frac{d}{dt} {}_O\hat{\chi}_P \right) \hat{a} \end{aligned} \quad (\text{A.7})$$

The constituent derivatives can be resolved as follows

$$\frac{d}{dt} {}_O\hat{\chi}_P = \begin{pmatrix} \frac{d}{dt} E^T & 0 \\ \left(\frac{d}{dt} r \right) \times E^T + r \times \frac{d}{dt} E^T & \frac{d}{dt} E^T \end{pmatrix} \quad (\text{A.8})$$

If the P frame has velocity \hat{v} expressed in the frame O as $\hat{v}_O = (\omega^T, v_O^T)^T$ we have $\frac{d}{dt} r = v_P = v_O - r \times \omega$ and $\frac{d}{dt} E = \omega \times E$. With this we can simplify the above expression into

$$\begin{aligned} \frac{d}{dt} {}_O\hat{\chi}_P &= \begin{pmatrix} \omega \times E^T & 0 \\ v_O \times E^T + \omega \times r \times E^T & \omega \times E^T \end{pmatrix} \\ &= \begin{pmatrix} \omega \times & 0 \\ v_O \times & \omega \times \end{pmatrix} \begin{pmatrix} E^T & 0 \\ r \times E^T & E^T \end{pmatrix} \\ &= \begin{pmatrix} \omega \times & 0 \\ v_O \times & \omega \times \end{pmatrix} {}_O\hat{\chi}_P \end{aligned} \quad (\text{A.9})$$

where we have used the identity $(a \times b) \times = a \times b \times - b \times a \times$.

By introducing the spatial cross operator

$$\begin{pmatrix} a \\ a_O \end{pmatrix} \hat{\chi} = \begin{pmatrix} a \times & 0 \\ a_O \times & a \times \end{pmatrix} \quad (\text{A.10})$$

we can simplify equation A.9 to

$$\frac{d}{dt} {}_O\hat{\chi}_P = \hat{v}_O \hat{\chi} {}_O\hat{\chi}_P \quad (\text{A.11})$$

Let \hat{v}_P be the velocity of frame P in P coordinates. Using the tensor transformation rule $\hat{M}_P = {}_P\hat{\chi}_O \hat{M}_{OO} \hat{\chi}_P$ we can state equation A.7 as

$$D\hat{a} = \frac{d}{dt}\hat{a} + \hat{v}_P \hat{\chi} \hat{a} \quad (\text{A.12})$$

by applying this rule to $\hat{v}_O \hat{x}$.

A.2.1 Spatial rigid body inertia

Let O be the frame centered at the body's center of mass we may then define the *spatial rigid body inertia* as

$$\hat{I}_O = \begin{pmatrix} 0 & m \text{ id} \\ I^* & 0 \end{pmatrix} \quad (\text{A.13})$$

where I^* is the ordinary inertia tensor. This tensor follows the ordinary transformation rules for tensors.

Let P be a coordinate frame moving with velocity \hat{v}_P as expressed in P coordinates. We can then derive an expression for $D\hat{I}$ with the same approach used in equation A.7 by writing

$$D\hat{I}_P = {}_P\hat{X}_O \left(\frac{d}{dt} \hat{I}_O \right) {}_O\hat{X}_P \quad (\text{A.14})$$

where

$$\begin{aligned} \frac{d}{dt} \hat{I}_O &= \frac{d}{dt} ({}_O\hat{X}_P \hat{I}_{PP} \hat{X}_O) \\ &= ({}_O\hat{X}_P \hat{v}_P \hat{x}) \hat{I}_{PP} \hat{X}_O + {}_O\hat{X}_P \left(\frac{d}{dt} \hat{I}_P \right) {}_P\hat{X}_O - {}_O\hat{X}_P \hat{I}_P (\hat{v}_P \hat{x} {}_P\hat{X}_O) \end{aligned} \quad (\text{A.15})$$

where we used

$$\begin{aligned} D_P \hat{X}_O &= D_O \hat{X}_P^S \\ &= (\hat{v}_O \hat{x} {}_O\hat{X}_P)^S \\ &= {}_P\hat{X}_O (\hat{v}_O \hat{x})^S \\ &= (\hat{v}_P \hat{x})^S {}_P\hat{X}_O \\ &= -\hat{v}_P \hat{x} {}_P\hat{X}_O \end{aligned} \quad (\text{A.16})$$

which leads to

$$D\hat{I}_P = \frac{d}{dt} \hat{I}_P + \hat{v}_P \hat{x} \hat{I} - \hat{I}_P \hat{v}_P \hat{x} \quad (\text{A.17})$$

Thus a rigid body moving with absolute velocity \hat{v} and inertia \hat{I} gives rise to

$$\begin{aligned} D\hat{I} &= \hat{v} \hat{x} \hat{I} - \hat{I} \hat{v} \hat{x} \\ &= [\hat{v} \hat{x}, \hat{I}] \end{aligned} \quad (\text{A.18})$$

where $[,]$ stands for the Lie bracket (also known as 'commutator').

A.2.2 Spatial force

We are now equipped to express Newton's equations of motion in spatial notation

$$\begin{aligned}\hat{\mathbf{f}} &= D(\hat{\mathbf{v}}) \\ &= [\hat{\mathbf{v}} \hat{\times}, \hat{\mathbf{l}}] \hat{\mathbf{v}} + \hat{\mathbf{l}} \hat{\mathbf{a}} \\ &= \hat{\mathbf{l}} \hat{\mathbf{a}} + \hat{\mathbf{v}} \hat{\times} \hat{\mathbf{l}} \hat{\mathbf{v}}\end{aligned}\tag{A.19}$$

Special attention should be directed to the so called *bias force* term $\hat{\mathbf{v}} \hat{\times} \hat{\mathbf{l}} \hat{\mathbf{v}}$ which accounts for all velocity dependent terms and falls out from first principles (!).

References

- [1] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constraints. *Computer Graphics*, 22(4):179–188, August 1988.
- [2] William W. Armstrong and Mark W. Green. The dynamics of articulated rigid bodies for purposes of animation. *Visual Computer*, pages 231–240, 1985.
- [3] Paul M. Isaacs and Michael F. Cohen. Controlling dynamic simulation with kinematic constraints, behavior functions, and inverse dynamics. In *Proceedings SIGGRAPH*, pages 215–224, 1987.
- [4] Jane Wilhelms and Brian Barsky. Using dynamics analysis to animate articulated bodies such as humans and robots. In *Graphics Interface*, pages 97–115, 1985.
- [5] Andrew Witkin and Mickael Kass. Spacetime constraints. In *Proceedings SIGGRAPH*, pages 159–168, 1988.
- [6] Mathew Moore and Jane Wilhelms. Collision detection and response for computer animation. In *Proceedings SIGGRAPH*, pages 289–298, 1988.
- [7] James K. Hahn. Realistic animation of rigid bodies. In *Proceedings SIGGRAPH*, pages 299–308, 1988.
- [8] Cliff Brett, Steve Pieper, and David Zeltzer. Putting it all together: An integrated package for viewing and editing 3d microworlds. In *Proc. 4th Usenix Computer Graphics Workshop*, pages 3–12, October 1987.
- [9] David Zeltzer, Steve Pieper, and David J. Sturman. An integrated graphical simulation platform. In *Proc. Graphics Interface*, pages 266–274, 1989.
- [10] Richard H. Lathrop. Constrained (closed-loop) robot simulation by local constraint propagation. In *Robotics and Automation*, pages 689–694. IEEE Council on Robotics and Automation, 1986.
- [11] Andrew Witkin, Michael Gleicher, and William Welch. Interactive dynamics. In *these proceedings*, 1990.

- [12] Michael McKenna. A dynamic model of locomotion for computer animation. Master's thesis, Massachusetts Institute of Technology, 1990.
- [13] Paul M. Isaacs and Michael F. Cohen. Mixed methods for complex kinematic constraints in dynamic figure animation. *The Visual Computer*, 4(6):296–305, 1988.
- [14] Jane Wilhelms. Dynamic experiences. In David Zeltzer, Norman Badler, and Brian Barsky, editors, *Making Them Move: Mechanics, Control and Animation of Articulated Figures*. Morgan Kaufman, 1990. in preparation.
- [15] Linda Petzold and Per Lotstedt. Numerical solution of nonlinear differential equations with algebraic constraints 2: Practical implications. *SIAM J. Sci. Stat. Comput.*, 7(3):720–733, July 1986.
- [16] Dirk Ferus. Lecture notes: Differential geometry. From a course given at the Technical University of Berlin, Department of Mathematics., 1985.
- [17] Ronen Barzel and Alan H. Barr. Controlling rigid bodies with dynamic constraints. In *Course Notes: Developments in Physically Based Modelling*, chapter E. ACM Siggraph, 1988.
- [18] Christopher C. Paige and Michael A. Saunders. Lsqr: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. on Math. Software*, 8(1):43–71, March 1982.
- [19] Roy Featherstone. *Robot Dynamics Algorithms*. Kluwer Academic Publishers, 1987.
- [20] C. William Gear. *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice-Hall, 1971.
- [21] Per Lotstedt. Numerical simulation of time-dependent contact and friction problems in rigid body mechanics. *SIAM J. Sci. Stat. Comput.*, 5(2):370–393, June 1984.
- [22] Per Lotstedt. Mechanical systems of rigid bodies subject to unilateral constraints. *SIAM J. Appl. Math.*, 42:281–296, 1982.
- [23] Alan C. Hindmarsh. Odepack, a systemized collection of ode solvers. In R. S. Stapleman et al., editor, *Scientific Computing*, pages 55–64. North-Holland, 1983.
- [24] Michael McKenna and Bob Sabiston. Grinning evil death, 1990. Computer animation.
- [25] Roy Featherstone. The calculation of robot dynamics using articulated body inertias. *The Intl. J. of Robotics Research*, 2(1):13–30, Spring 1983.
- [26] R. S. Ball. *A Treatise on the Theory of Screws*. Cambridge University Press, 1900.