An Implementation of the Spacetime Constraints Approach to the Synthesis of Realistic Motion

Pär Winzell

November 26, 2004

Abstract

This thesis presents a computer implementation of the *Spacetime Constraints* approach to synthesizing realistic-looking motion for creatures in a simulated physical environment.

Earlier work in this field has shown that using *efficiency* as the criterium for selecting among the physically valid ways a creature can accomplish a task yields visually pleasing results.

Creatures are constructed from rigid bodies linked through angular joints and given the power of motion through torque-producing muscles. The state of the system is described in Cartesian and angular coordinates, which are represented over time as piecewise cubic polynomials with $\mathcal{C}^1$ transitions, in the Hermite function basis. Time-varying muscle tensions are also explicitly represented, and physical validity of the motion is achieved through a finite-element formulation of the Euler-Lagrange equations.

Subject to these constraints, an objective function measuring the energy consumed by the creature's muscles is minimized over time. This is a traditional, highly non-linear, constrained optimization problem, to which is applied an SQP-style solver package. The current implementation outputs a graphical representation of the animation on Pixar's RenderMan$^{\text{TM}}$ format.

The implementation and the software packages it depends on may be freely redistributed and used for non-commercial purposes.

Contents

List of Figures

1 Acknowledgements

This thesis project has been a presence in my life for many years, in one form or other. Many thanks go to my thesis advisor Thomas Karlsson for his good humour and heroic patience ever since I first stumbled over this subject matter in 1993. My darling wife Karen has valiantly endured neglect in competition with a whole menagerie of sadly crippled animation subjects. All of my family has been supportive, as always. Several people proofread and commented the paper – thank you.

This implementation relies heavily on free software. It uses a package for non-linear optimization problems containing the solver *HQP* and the frontend *Omuses* for it. These programs were written by Rüdiger Franke at Technische Universität Ilmenau, and are distributed under a license that allows free redistribution of the source-code. This package relies in turn on software from a host of other authors, similarly available to the public.

Many of the figures in this paper were drawn using the Blue Moon Rendering Tools by Larry Gritz. The thesis was written in LaTeX, on a machine running the free operating system Linux. Much respect is due the authors of these systems and others for making their efforts available to the public.

2 Introduction

The field of Computer Graphics is a varied and rewarding one. There are any number of utterly fantastic goals to work towards, and no shortage of unsolved problems to tackle on the way. Progress in the field is rapip: a ceaseless explosion of hardware ability, improvements in fundamental algorithms, the development of new mathematical tools such as wavelets, the influx of diverse competence, all these things serve to widen and deepen the field. If the things we can do now are amazing, what we will be able to do tomorrow is simply breathtaking.

2.1 One Motivation

One of the many fantastic ideas that drive the Computer Graphics (CG) field on at such a pace is that of the full-blown virtual reality, self-contained and viewer-independent, bubbling with complex behaviour and artificial life. Seen as an artistic medium, virtual reality is something completely new. Skills which we have acquired drawing static images and making single-perspective movies will be useless in application to the canvas of "reality".

Even on a lesser scale than this, non-trivial three-dimensional spaces will need to be *populated*, densely and in detail, with objects and sub-systems controlled by computer algorithms, interacting with each other within the physical simulation of their virtual environment.

The overwhelming technical problem this represents is perhaps most naturally tackled through an abstraction approach: on the one hand, the encompassing framework of the environment, establishing the rules under which objects interact with their environment and each other; on the other, the design and implementation of the sub-systems and objects that fill the virtual void, and which may rely completely on the existence of the virtual environment.

A virtual world with virtual grass and virtual trees should be roamed by virtual creatures. How could one possibly begin implementing living beings? In years to come, this question will undoubtedly receive much attention. For the purposes of motivating this thesis, it suffices to note that the sub-problem of motor control is a central one.

Given computer code libraries that generate motion from sufficiently high-level control, the remaining problems would not seem nearly so daunting. Scripting behaviour for a little field-mouse would be next to trivial if there was at our disposal a template for a living, breathing virtual creature, needing only a will plugged into it to become itself. In the light of this realization, exploring the nature of motion seems warranted.

2.2 Means of Motion

If a rock is dropped on earth, it falls to the ground in a deterministic motion, trivially computed. We can build more complex mechanical systems and simulate their equally deterministic states forward in time using a computer (or a lot of patience). The evolution of such a system may be considered the inevitable result of the forces at work within it.

The rock above moves as it falls, but the verb *move* can mean something else when a *will* is involved. Living creatures tense their muscles, which apply torque to their joints, causing them to move. A significant portion of their brain sits between the urge to perform some physical action and the resulting motion, a portion dedicated to the task of motor control.

Whether completely accurate in neurological terms or not, it makes sense to explore this mind-body division further for our purpose, which is that of artificial motion synthesis. There is the high-level control (the will) with the task of evaluating, at whatever rate it is able, the ever-changing perceptions of the environment, updating its internal notions of the state of the world, all the while generating decisions.

In contrast, the task of motor-control is firmly grounded in the present, occupying itself largely with the relationships between velocity and acceleration. It is this latter functionality that we would ultimately wish to have available as an abstraction layer, for implementations of virtual creatures to rely on.

Intriguingly, the problem of generating realistic-looking motion satisfying high-level demands for a virtual creature can be considered basically solved.

2.3 The Spacetime Control Paradigm

At the SIGGRAPH conference of 1988, Andrew Witkin and Michael Kass presented the paper [1], "*Spacetime Constraints*", which showed in practice that visually pleasing behaviour can follow from fundamental concepts.

In their method, the system to be animated is parameterized in a set of degrees of freedom (DOF) – for example, a planar creature constructed from rigid bodies linked through angular joints. The DOF are discretized over an interval of time, yielding a finite set of variables. Each possible configuration of these variables then represents an animation of the system over the interval.

Next, forces are brought into play – the creature is given muscles – discretized similarly and represented with another set of variables. Equations of motion are constructed using symbolic algebra techniques, constraining the system so that the values of the time-varying torques produced by the mus-

cles and those of the the positional DOF are constrained by the requirement of conformance to physical law.

High-level control is achieved through constraints on the DOF and other relevant quantities at different instants of time. Most commonly, the DOF at the beginning of the animation are given, and their rate of change are constrained to be zero. Often, what we want to solve is a boundary-value problem, where we also fully constrain the system at the end of the animation. Such a scenario is akin to the keyframe interpolation problem that plays such a major part in contemporary computer animation software systems.

These *control problems* are generally very underconstrained; they have infinitely many solutions. To pick one solution out of many, we discriminate through the criterium of *efficiency*. Finding the most efficient solution is an optimization problem for which there are known algorithms.

The solution is then the explicit description of the chunk of spacetime representing the most efficient way for the creature to perform the high-level task defined by the constraints. There is no room for surprises in this paradigm; it is an offline computation given exact knowledge of the environment for the duration of the animation.

Since the solution includes the description of joint torques, it tells us not only what the optimal motion looks like, but also how it came about; exactly how the creature must tense its muscles at each point in time to move just so. We could later place the virtual creature in the physical simulation of a virtual reality. The computed optimal torques could be applied from moment to moment, and the creature would execute the correct motion – as long as the environment exactly matched that which was used in the Spacetime Control computation.

2.4 Another Computational Approach

Five years later, J. Thomas Ngo and Joe Marks presented [9], "Spacetime Constraints Revisited", at SIGGRAPH '93. As the title implies, this too is a method where control over the solution takes the form of constraints in space and time. The reported results of this paper and that of Witkin and Kass are also similar. Nevertheless, the two methods are very different. In fact, they actually solve different problems.

Instead of working directly on optimizing a portion of spacetime, Ngo and Marks use the indirect control method of creature *behaviour*. The output of their computer program is not a single animation, but a set of reflex-like stimulus-response reactions for the creature.

These reflexes are parameterized and applied to them is a *genetic algorithm*, which is another approach to solving optimization problems. The

algorithm constructs a "population" of different solutions and subjects them to forced evolution. Behaviour configurations that yield desirable results in physical simulation are considered superior, and are allowed to "mate" with high frequency.

The desirability criterium is encoded in a *fitness function*, and includes, among other things, any requirements on pose and position at different instants of time, i.e., the spacetime constraints. The population moves towards a state where its members optimize this fitness function and this state is taken to be the solution.

2.5 This Thesis

Where Witkin and Kass' method constructs a single sequence of motion for a creature. Ngo and Marks give the creature something of a mind. If placed in a virtual world, it does have the ability to act on its perceptions of the environment and respond to events. In practice, though, these behaviour controllers are so simple that they do very little but act out the desired motion.

As the discipline of artificial life progresses, behaviour-controller methods become increasingly proficient, and outgrow applications as simple as this. The paper [10] is worth study in this context.

What the methods have in common is the high-level control; a function discriminates bad motion from good, and it is the responsibility of the automatic solvers to satisfy this function as well as they can. An interesting idea for the future could be to generate a library of classified motions using the former method, and make this knowledge base available to behaviour modules evolved using the latter.

The rest of this paper will present an implementation similar to the original one by Witkin and Kass. We start by identifying and posing the problem, in preparation for the choice of a specific solution method. The implementation is then described, and finally some results are presented along with some discussion and a survey of what else has been done in this field.

2.6 Electronic Access

There is a web-site dedicated to this thesis project at

`http://alyx.com/~zell/exjobb`

where the paper and the computer implementation can be found along with pointers to the packages on which it depends, and possibly errata and updates. The author may be contacted at the email address `zell@alyx.com`.

3 Problem

3.1 Continuous Dynamics

In this thesis, we will consider the synthesis of motion of objects in an abstract physical reality. We make the requirement that the configuration of these objects (the *creatures*) can be described at all times using a finite set of m *generalized coordinates* $q_i(t)$. Thus the state of the system at time t is fully described by the $q_i(t), \dot{q}_i(t)$ and so to create a motion over an interval of time is to fully define the q_i on it.

In general, the q_i can represent virtually any physical magnitude and the associated *generalized force* Q_i then takes on a dimension such that $Q_i\, dq_i$ always constitutes *work*. See e.g. [16] or [17] for a full introduction to analytical mechanics. In this thesis, the q_i will be Cartesian and angular coordinates, and the Q_i, correspondingly, Cartesian forces and torques.

3.1.1 Equations of Motion

We restrict our attention to *physically valid* motion, to systems whose evolution over time is governed by Newton's laws of motion. These are generally ordinary differential equations of the second order. With the system defined in terms of generalized coordinates, we can impose the constraints of physics through *Lagrange's equations*,

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} - Q_i = 0, \qquad 1 \leq i \leq m, \tag{3.1}$$

which for now we take to be valid at all times. The function $L = T - V$, the *Lagrangian function*, is the difference between the system's total kinetic and potential energies. It encodes the physical structure of the creatures and their environment, as well as any conservative force-fields such as gravity. In general, $\frac{\partial V}{\partial \dot{q}_i}$ is zero, and $\frac{\partial V}{\partial q_i}$ represents conservative forces such as gravity, and is of dimension generalized force, like the Q_i. The latter are needed to represent forces that are not conservative, and so cannot be included as gradients of the scalar field V.

3.1.2 Simulation

Considering L to be a fixed property of the system, (3.1) intertwines the q_i and the Q_i in accordance with Newtonian mechanics. The special case where the Q_i are known and the $q_i(t_0)$ and $\dot{q}_i(t_0)$ specified at some time t_0 is an *initial value* problem. For these, the (3.1) fully determine the exact future

state of the system, and solving these equations forwards in time is known as *physical simulation.*

To illustrate this, consider a ball falling to the ground from the roof of a tower. If the ball has mass $m > 0$ and the acceleration of gravity is G, the Lagrangian function is $L = T - V = m(\dot{y}^2/2 - G\,y)$ and the ball will be governed by

$$\frac{d(m\dot{y})}{dt} - mG = 0 \iff \ddot{y} = G \tag{3.2}$$

until it hits the ground, at which time other forces come into play. The height of the ball above the ground, $y(t)$, is known at the time t_0 when we drop it, as is its velocity, $\dot{y}(t_0) = 0$, so

$$\dot{y}(t) = G\,(t - t_0), \qquad y(t) = y(t_0) + G\frac{(t - t_0)^2}{2} \tag{3.3}$$

By solving this equation we have performed a tiny physical simulation, a trivial case with a closed-form solution. In problems only a little bigger, the equations are non-linear and recourse must be taken to numerical ODE solvers.

The quick and accurate simulation of complex physical systems is an important problem and the subject of much research, in the computer graphics field and in others. However, simulation does not *create* motion, it merely solves the equations of motion to calculate the inevitable deterministic effects of the Q_i accelerating the system just so at each point in time.

3.1.3 Control

Where in simulation problems the forces Q_i are given and we want to solve for the q_i, *control problems* typically arise when we treat the Q_i as variable and make demands on the q_i.

Let's expand the example above by keeping the initial conditions and adding a second condition that $\dot{y}(t_1) = 0$ for some given $t_1 > t_0$, i.e. a soft landing at a pre-determined time. Since we cannot make demands on gravity, such a condition can only be satisfied if the ball can be influenced in some other way. Let us assume that we have at our disposal a (somewhat strange) external force $F_y \cdot \hat{y} = k(t - t_0) \cdot \hat{y}$ that acts on the ball. Here, k is constant over the course of the motion, but from the point of view of the control problem, it is a variable for which to solve. The equation of motion is now

$$\frac{d(m\dot{y})}{dt} - mG = F_y \iff \ddot{y}(t) = G + \frac{k}{m}(t - t_0) \tag{3.4}$$

so

$$\dot{y}(t) = G\,(t - t_0) + \frac{k}{m}\frac{(t - t_0)^2}{2} \tag{3.5}$$

and

$$\dot{y}(t_1) = G\,(t_1 - t_0) + \frac{k}{m}\frac{(t_1 - t_0)^2}{2} = 0 \Rightarrow k = -\frac{2mG}{t_1 - t_0} \tag{3.6}$$

We have solved a control problem – computer, within a given parameterization, the force to apply to the system, for a requested system state to come about.

3.1.4 Variational Dynamics

The equations (3.1) are differential equations that must hold at each point in time. In the circumstances we shall be dealing with, a weaker formulation of these equations turns out to be equivalent. Let $\eta(t)$ be any arbitrary, well-behaved function defined on $[t_0, t_1]$ with the additional property that $\eta(t_0) = \eta(t_1) = 0$. Then

$$\int_{t_0}^{t_1} \left\{ \frac{d}{dt}\left(\frac{\partial L(t)}{\partial \dot{q}_i}\right) - \frac{\partial L(t)}{\partial q_i} - Q_i(t) \right\} \eta(t)\, dt = 0 \tag{3.7}$$

certainly holds when (3.1) does. Under certain regularity conditions, the reverse implication holds as well, and (3.7) holding true for all such $\eta(t)$ implies (3.1) holding true for all $t \in [t_0, t_1]$.

To see why, consider the possibility that (3.1) is non-zero for some $t \in [t_0, t_1]$. Since by regularity requirements this function is continuous, there must then exist an interval containing t where the function is strictly positive or negative. An $\eta(t)$ could always be constructed that exposed this fact, leading to a contradiction of (3.7). The result follows. The $\eta(t)$ are called *trial functions*.

Integrating (3.7) by parts, we have

$$\left[\frac{\partial L(t)}{\partial \dot{q}_i}\eta(t)\right]_{t_0}^{t_1} - \int_{t_0}^{t_1} \left\{ \frac{\partial L(t)}{\partial \dot{q}_i}\dot{\eta}(t) + \left(\frac{\partial L(t)}{\partial q_i} + Q_i(t)\right)\eta(t) \right\} dt = 0 \tag{3.8}$$

and $\eta(t_0) = \eta(t_1) = 0$ so

$$\int_{t_0}^{t_1} \left\{ \frac{\partial L(t)}{\partial \dot{q}_i}\dot{\eta}(t) + \left(\frac{\partial L(t)}{\partial q_i} + Q_i(t)\right)\eta(t) \right\} dt = 0 \tag{3.9}$$

Writing differential equations in this *variational* form leads us naturally towards the *Finite Element Method (FEM)* for solving them. Note that an intrinsic property of this integral formulation is to make statements that hold over *intervals* of time. In contrast, the (3.1) have no awareness of past and present, but are relationships, from moment to moment, between cause and effect.

3.1.5 Impact and Regularity Issues

In general, physically sensible Q_i are bounded, the $\dot{q}_i$ lie fully in $\mathcal{C}^0$ and the q_i in $\mathcal{C}^1$. However, situations such as *impact* arise, when the $\dot{q}_i$ must have discontinuities. For example, when two objects in the virtual world collide, they do so inelastically. There must be an instant effect, because they are rigid bodies and do not compress. This amounts to infinite acceleration over an infinitismal time period. To avoid tangling with this unpleasant singularity, consider a "hammer blow" of magnitude P and direction $\hat{s}$. The blow applies for a very brief period of time δt a force

$$\bar{F} = F \cdot \hat{s} = \frac{P}{\delta t} \cdot \hat{s} \tag{3.10}$$

on the system at a point $\bar{p}$. Recalling that $Q_i \, dq_i$ always has the dimension of work, the force $\bar{F}$ must influence the system according to the transformation

$$Q_i = \bar{F} \cdot \frac{\partial \bar{p}}{\partial q_i} = F \cdot \hat{s} \cdot \frac{\partial \bar{p}}{\partial q_i}. \tag{3.11}$$

Because the interval of time δt is small, the state of the system can be considered constant over it. As $\delta t \to 0$, $F \to \infty$, but P does not vary. The equations of motion (3.1) yield in the limit

$$\delta t \cdot Q_i = \delta t \cdot \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \delta t \cdot \frac{\partial L}{\partial q_i} \quad \to \quad \Delta\left(\frac{\partial L}{\partial \dot{q}_i}\right) \tag{3.12}$$

and application of (3.10) and (3.11)

$$\delta t \cdot Q_i = (\delta t \cdot F) \cdot \hat{s} \cdot \frac{\partial \bar{p}}{\partial q_i} = P \cdot \hat{s} \cdot \frac{\partial \bar{p}}{\partial q_i} \tag{3.13}$$

so

$$\Delta\left(\frac{\partial L}{\partial \dot{q}_i}\right) = P_i \tag{3.14}$$

where P_i is *generalized momentum*, and is distributed from $\bar{P}$ according to the same pattern (3.11) as dictates the Q_i from $\bar{F}$, i.e.

$$P_i = \bar{P} \cdot \frac{\partial \bar{p}}{\partial q_i} = P \cdot \hat{s} \cdot \frac{\partial \bar{p}}{\partial q_i}. \tag{3.15}$$

In other words, just as forces may be applied to points $\bar{p}$ on creatures over an interval of time, so may blows of pure momentum at discrete instants of time, with transition conditions given by (3.14).

For a simple example of when this may occur, consider a creature landing on a hard floor. At the moment of impact there is a jolt which travels instantly through its body parts and discontinuously changes the angular joint velocities. In a living creature, where this cannot happen, skeletal elasticity absorbs the shock up to the shattering point of bone.

For the remainder of this paper, we will consider the duration of the animation to be structured as sequential intervals $[t_0, t_1], [t_1, t_2], \ldots$, within which the Q_i are *bounded*, the $\dot{q}_i$ *continuous*, and (3.9) hold. These intervals we call *stages*. They may be separated by discontinuity blows, as outlined above, with transition conditions on the form

$$\frac{\partial L}{\partial \dot{q}_i}(t_j^+) - \frac{\partial L}{\partial \dot{q}_i}(t_j^-) = P_i, \qquad j = 1, \ldots \tag{3.16}$$

The variational equations of motion (3.9) are integrated over each stage independently of the others, and we can treat them as independent subproblems. Most of the rest of this work in this paper will assume an interval $[t_0, t_1]$, which we take to be a generic stage satisfying the continuity assumptions outlined above.

3.1.6 Realism

In the example control problem of the falling ball, a single additional discrete variable k was introduced and a single corresponding additional constraint was imposed. These cancel out, and there is a single correct solution.

In the general setting, there is not. Animation problems are underconstrained. For one thing, there are many control variables; the Q_i vary quite freely. Second, there are not many constraints. The whole point of the Spacetime paradigm is that control is sparse and high-level. Typically, boundary pose and position are constrained along with a few conditions such as "jump this high". There are infinitely many variations a creature may make in its precise muscle control, all of them resulting in satisfactory jumps.

So it is not enough to define a task to perform and to demand physical validity – there are still infinitely many possible motions. There must be a criterium for discriminating between these possibilities. We have discarded from our consideration all motion that violates physical law – leaving us with motion that is *real* – we now turn our attention to the *realistic*.

This is where we descend into the thoroughly subjective. We are trying to find some unifying, rational property of motion that appeals to the senses as realistic-looking. An obvious place to start this search is motion that is *truly* real; in this case, that exhibited by living, biological creatures. While

we touch this field only lightly, it may be that a basic understanding of physiology is as important as mathematical and computer-science skills are, in an endeavor such as this.

It seems reasonable to suppose that one of nature's fundamental criteria in discriminating between good and bad behaviours of motion is *efficiency*. Animals spend a lot of time gathering food for energy, and there is surely great evolutionary pressure on their motor control centers to perform without wasting such a precious resource. It is the great claim to fame of the Space-time Constraints method that it has shown in practice that realistic-looking motion does follow from this intuition.

Efficiency can be formalized well: given a system description and a well-defined physical task to perform, find motion that performs it while burning as little energy as possible, i.e.

$$\min W^{tot}(\bar{Q}, \dot{\bar{q}}, \bar{q}) \quad \text{subject to} \tag{3.17}$$

$$\{\bar{Q}, \dot{\bar{q}}, \bar{q}\} \text{ satisfy the laws of physics} \tag{3.18}$$

$$\{\bar{Q}, \dot{\bar{q}}, \bar{q}\} \text{ perform the appointed task} \tag{3.19}$$

where W^{tot} is a measure of the metabolic energy consumed in all the muscles of the creature over the duration of the motion. The first set of constraints, those enforcing the laws of physics, are expressed by the equations of motion (3.9).

3.2 Discrete Dynamics

The variables of this problem are real-valued functions defined on an interval of time $[t_0, t_1]$, elements of infinite-dimensional function spaces. For a numerical algorithm to be able to manipulate and reference these functions, they must have a finite description. By approximating them in a subspace spanned by a finite basis of functions $\phi_j(t)$, $t \in [t_0, t_1]$, functions in general become linear combinations, e.g.,

$$q_i(t) = \sum_{j=1}^{n} c_{i,j}\phi_j(t), \qquad \dot{q}_i(t) = \sum_{j=1}^{n} c_{i,j}\dot{\phi}_j(t), \quad \ldots \tag{3.20}$$

and are thus represented by a finite sequence of coefficients, such as these $c_{i,j}$. This allows for a discrete problem formulation.

The subspaces in which we approximate the true paths will be those of piecewise polynomials, specifically of degrees one and three. The discretization process transforms the domain of definition $[t_0, t_1]$ of the function onto the dimensionless $[0, N]$, and defines there N polynomials, e.g.

$$f(t) = \mathcal{P}_i^f(u), \quad u \in [0, 1], \quad 1 \leq i < N \tag{3.21}$$

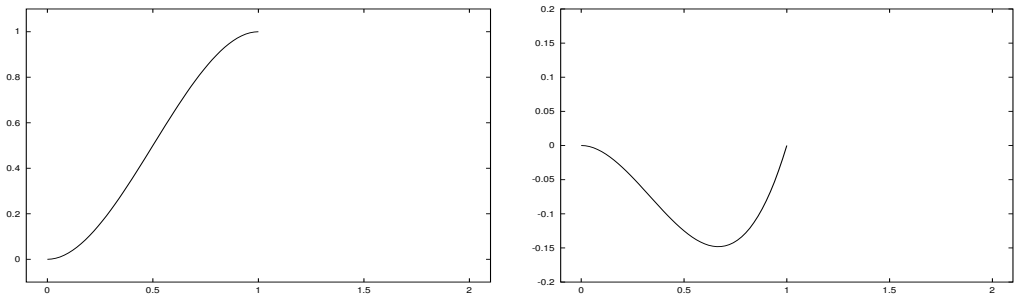

Figure 1: Basis functions $\phi^{(1)}$ and $\phi^{(2)}$

for some polynomials $\mathcal{P}_i^f(u)$.

In general, for degree L these N pieces are fully determined by $(L+1)N$ coefficients; $4N$ in the cubic case. Functions that represent muscle tension may be discontinuous, and so these $(L+1)N$-dimensional spaces are of value to us. However, the functions that represent *coordinate trajectories* are at least continuously differentiable at each stage, i.e. $f \in \mathcal{C}^1$; a condition which must be enforced over transitions from one piece to the other. Where one piece ends, the next must begin, and their tangents must match as well. From one piece to the next, then, only two degrees of freedom are introduced. This reduces the dimension of the function space from $4N$ to $2N+2$.

3.2.1 Hermite Cubics

This $2N+2$-dimensional space has a very useful basis, consisting of integer translates of functions $\phi^{(1)}(u)$ and $\phi^{(2)}(u)$, known as *Hermite cubics*. They are plotted in Figure 1.

These functions are piecewise cubic polynomials defined over two intervals (i.e. $u \in [i, i+2]$). By populating $[-1, N+1]$ with $2N+2$ integer translates of these functions, the desired basis is constructed. For any $t \in [t_0, t_1]$, at most four basis functions are non-zero. Perhaps the most striking property of these functions is their interpolatory nature;

$$\phi^{(1)}(1) = 1, \qquad \dot{\phi}^{(1)}(1) = 0, \tag{3.22}$$

$$\phi^{(2)}(1) = 0, \qquad \dot{\phi}^{(2)}(1) = 1, \tag{3.23}$$

i.e., the *value* of $f(t)$ at a knot is determined by a single coefficient, that of the associated $\phi^{(1)}(1)$, and the value of its *derivative* $\dot{f}$ by that of $\phi^{(2)}(1)$. This is illustrated in Figure 2 which depicts a single cubic on on $[0, 1]$ and the four weighted basis functions that define it.

Explicitly,

$$\phi^{(i)}(u) = \phi_j^{(i)}(u - j), \quad u \in [j, j+1] \tag{3.24}$$

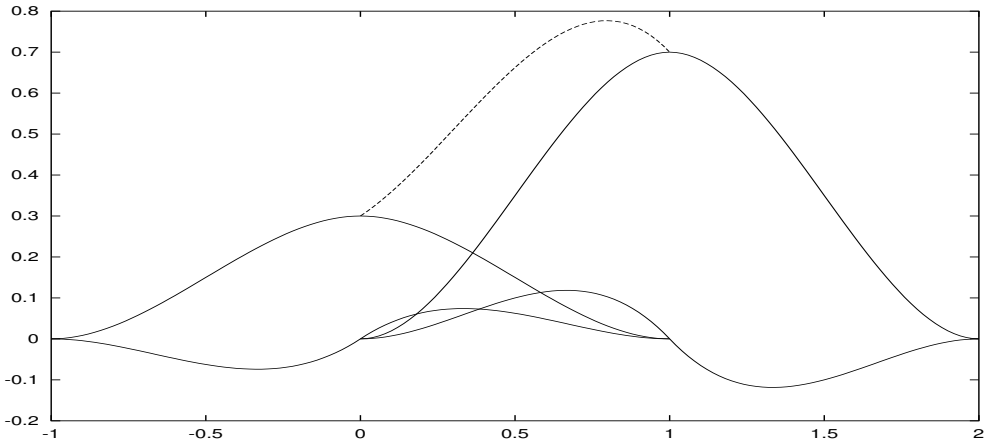

Figure 2: Scaled basis functions (solid) defining a cubic (dashed) on $[0, 1]$

with

$$\phi_0^{(1)}(v) = -2v^3 + 3v^2, \qquad \phi_1^{(1)}(v) = 1 + 2v^3 - 3v^2, \tag{3.25}$$

$$\phi_0^{(2)}(v) = v^3 - v^2, \qquad \phi_1^{(2)}(v) = v^3 - 2v^2 + v \tag{3.26}$$

The piecewise linear case is simpler, and the traditional basis for it consists of translates of the hat function

$$\phi(u) = \begin{cases} u, & u \in [0, 1] \\ 2 - u, & u \in [1, 2]. \end{cases} \tag{3.27}$$

3.2.2 Finite Element Method

With functions represented as finite linear combinations, differential equations like (3.1) must become statements about their coordinates in the finite-dimensional function spaces, i.e. discrete equations on the $c_{i,j}$ of (3.20).

In [1], Witkin and Kass sample the interval $[t_0, t_1]$ at discrete instants of time $t = 0, h, 2h, \ldots$ and functions are represented as piecewise constants around these points, so $f_i = f(t_i) = f(ih)$ are the control variables. The equations of motion are constructed through the method of *finite differences*, where derivatives in (3.1) are replaced with differences, e.g.

$$\dot{f}_i = \dot{f}(t_i) \approx \frac{f(t_i + h) - f(t_i)}{h} = \frac{f_{i+1} - f_i}{h} \tag{3.28}$$

and evaluated at each sample point t_i. The result is a set of difference functions on the f_i.

A different possibility follows from writing the differential function in variational form, as we did to get (3.9). Recall that in this form, the differential function is tested against an infinite set of trial functions $\eta(t)$, with one integral equation resulting from each test. The natural way to construct a discrete problem out of this form is to test against only a finite subset of trial functions $\eta(t)$. By choosing specifically the *basis* functions of (3.20) as trial functions, we have arrived at the *Finite Element Method*, the FEM.

Applying this procedure to (3.9), which is the equation relevant to this thesis, yields a set of discrete equations

$$\int_{t_0}^{t_1} \left\{ \frac{\partial L(t)}{\partial \dot{q}_i} \dot{\phi}_j(t) + \left(\frac{\partial L(t)}{\partial q_i} + Q_i(t) \right) \phi_j(t) \right\} dt = 0, \quad \forall j. \tag{3.29}$$

In simple situations, the functions $\frac{\partial L(t)}{\partial \dot{q}_i}$, $\frac{\partial L(t)}{\partial q_i}$ and $Q_i(t)$ are *linear*. The example problem of the falling ball employed earlier has just such a simple structure. Writing the acceleration law (3.4) in variational form,

$$m \int_{t_0}^{t_1} \left\{ \dot{y}(t)\dot{\eta}(t) + \left(G + \frac{F_y}{m} \right) \eta(t) \right\} dt = 0, \tag{3.30}$$

then expanding y as a sum of basis functions,

$$m \int_{t_0}^{t_1} \left\{ \sum_j c_{y,j} \dot{\phi}_j(t)\dot{\eta}(t) + \left(G + \frac{F_y}{m} \right) \eta(t) \right\} dt = 0, \tag{3.31}$$

and finally using those basis functions ϕ_i that are zero at the boundaries as trial functions – and rearranging – yields

$$\sum_j c_{y,j} \int_{t_0}^{t_1} \dot{\phi}_j(t)\dot{\phi}_i(t)\, dt + \left(G + \frac{F_y}{m} \right) \int_{t_0}^{t_1} \phi_i(t)\, dt = 0. \tag{3.32}$$

This may look to be more complex than what we started out with rather than a simplification, but note that with

$$a_{i,j} = \int_{t_0}^{t_1} \dot{\phi}_j(t)\dot{\phi}_i(t)\, dt, \tag{3.33}$$

$$f_i = \int_{t_0}^{t_1} \phi_i(t)\, dt, \tag{3.34}$$

it reduces to

$$\sum_j c_{y,j}\, a_{i,j} + \left(G + \frac{F_y}{m} \right) f_i = 0. \tag{3.35}$$

Here, $a_{i,j}$ and f_i are *static*, determined by the basis functions only, and so easily precomputed. The equation (3.35) is a linear system of equations in the $c_{y,j}$, which can be solved in a single step, using any of countless numerical methods.

This example constitutes a trivial yet illustrative application of the FEM for discretizing ODE's. Note that when the ϕ have narrow support, as they do both in the Hermite and the Hat bases, most of the $a_{i,j}$ are *zero*, so (3.35) is a *sparse* problem. This is essential for keeping the time-complexity of the problem managable.

In the general, highly non-linear case of multi-body equations of motion, the rearrangement performed above is not possible, and the solution cannot be found in a single step. In the next section, the equations of motion (3.29) will be constraints in an optimization problem, in which they are the major source of non-linear complications. To account for them we will be forced to take an iterative approach, where the functions $\frac{\partial L}{\partial \dot{q}_i}$ and $\frac{\partial L}{\partial q_i}$ are approximated with local Taylor expansions, and a sequence of equation systems like (3.35) are constructed. We will be using solutions to these approximative sub-problems as directions in which to search for the best next step towards the true solution.

3.3 Formulating the Problem

With functions discretized, the tentative problem formulation (3.17) can be restated in the standard form of a discrete optimization problem – to find

$$\min_{x \in \Re^n} f(\bar{x}), \tag{3.36}$$

subject to variable bounds

$$\bar{x}_l \leq \bar{x} \leq \bar{x}_u \tag{3.37}$$

and constraints

$$\bar{c}_l(\bar{x}) \leq \bar{c}(\bar{x}) \leq \bar{c}_u(\bar{x}). \tag{3.38}$$

The general case of this problem remains unsolved; methods generally can guarantee only to find local minima. Special cases exist with stronger results, e.g. when the objective function and/or the constraints are linear or quadratic. The quadratic case is important to us because in practice, solutions to nonlinear problems are often sought by repeatedly solving locally approximative quadratic subproblems.

4 Solution

4.1 The Optimizer

To solve the problem (3.36) we need a solver for non-linear optimization problems. We have chosen *HQP* by Rüdiger Franke [13], which seems to perform very well, and has a non-commercial license – an important consideration, since the stated goal of this project is to write a freely available implementation.

4.1.1 Sequential Quadratic Programming

HQP (the "H" is for "Huge"!) implements the SQP algorithm. It uses Newton's method on *another* Lagrangian function

$$\mathcal{L}(\bar{x}, \bar{\lambda}) = f(\bar{x}) - \sum_i \lambda_i c_i(\bar{x}), \tag{4.1}$$

quite different from the *kinetic potential* function L used in the equations of motion earlier in this paper. The function $\mathcal{L}$ combines the objective function with the effect of the constraints through the method of *Lagrangian multiplers*, the λ_i. When $\mathcal{L}$ is (at least locally) minimal,

$$\nabla\mathcal{L}(\bar{x}, \bar{\lambda}) = \bar{0} \tag{4.2}$$

i.e.

$$\nabla f(\bar{x}) = \sum_i \lambda_i \cdot \nabla c_i(\bar{x}), \tag{4.3}$$

$$c_i(\bar{x}) = 0, \tag{4.4}$$

then $f(\bar{x})$ enjoys a local extremum or a saddle point there *relative to the constraints*. It should be noted that this discussion is valid for equality constraints only. Inequality constraints are a fairly complex extension, and left out of this discussion for the sake of clarity.

Newton's method solves an unconstrained optimization problem by repeatedly expanding the objective function as a power series and solving the quadratic problem generated by truncating higher powers. The quadratic information is available through the Hessian, and applying the method to $\mathcal{L}$ yields for each iteration k an equation

$$\nabla^2\mathcal{L}(\bar{x}^{(k)}, \bar{\lambda}^{(k)}) \begin{pmatrix} \delta\bar{x}^{(k)} \\ \delta\bar{\lambda}^{(k)} \end{pmatrix} = -\nabla\mathcal{L}(\bar{x}^{(k)}, \bar{\lambda}^{(k)}), \tag{4.5}$$

solved for a direction

$$\bar{s} = \begin{pmatrix} \delta\bar{x}^{(k)} \\ \delta\bar{\lambda}^{(k)} \end{pmatrix} \tag{4.6}$$

along which a *line-search* is performed. The value of $\mathcal{L}$ is sampled along $\bar{s}$ to confirm the quality of the approximated solution. In this situation it is possible to prove convergence towards a *local* minimum, and a theoretical *quadratic* rate of convergence once the process comes "near enough" to such a minimum.

If the search-space is well-behaved in the vicinity of the current solution, the quadratic approximation works well and the solution to the sub-problem makes for a good next step in the iterative scheme. More commonly, the landscape is non-linear to the point of chaos, "near enough" occurs very late, and convergence is sluggish.

In practice, the analytical Hessian $\nabla^2\mathcal{L}$ is not used in (4.5). Instead, a numerical approximation is constructed step by step from the gradient information. There are several ways to perform this construction; HQP implements *BFGS* updates (see e.g. [18]).

HQP repeatedly evaluates not only the objective function $f(\bar{x})$ and the constraints $c_i(\bar{x})$, but also their gradients, for various $\bar{x}$. These values constitute the only perception the solver has of the search-space, and writing the computer code to evaluate them makes up the bulk of the implementation effort.

This data is fed to HQP through a C++ front-end called *Omuses* [14], which not only provides the interface between the core solver and our code, but also makes available several powerful tools to allow for a simplified problem description.

4.1.2 ADOL-C

The current implementation makes use of only one of these tools, namely *automatic differentiation* implemented by the ADOL-C [15] package. By using C++ operator overloading and *active variables*, ADOL-C builds up a run-time record (a *tape*) of the operations performed in the algorithms that evaluate e.g. $f(\bar{x})$. Using this information, it can then automatically compute the exact *analytical* gradients those same expressions.

For a trivial example, consider a function, something like

```
adouble KineticParticleEnergy(adouble Mass, adouble Velocity) {
    return Mass * Velocity * Velocity / 2;
}
```

This is a computer representation of a well-known physical formula; it evaluates $T = m\frac{v^2}{2}$. The *adouble* type it returns, however, represents not only the value of the expression, but also a record of how it was computed. As a consequence, the chain rule may be applied,

$$\nabla T = (\nabla m)\,\frac{v^2}{2} + m\,v\,\nabla v, \tag{4.7}$$

and since the input parameters v and m are also *adoubles*, this procedure can be applied to them as well, until the dependency of ∇T on the most primal variables in the system has been computed.

Such functionality is very useful for our purposes. It reduces the scope of the problem to providing a computer code for the *evaluation* of f and the c_i. The necessary derivative information is then automatically computed. Compared to having to calculate the derivatives ourselves, ADOL-C makes the implementation quite a bit easier, and also more secure. It is easy to make a mistake differentiating complex expressions.

4.2 Description of the Dynamics System

The Euler-Lagrange equations we rely on for physical validity are extremely general, and generalized coordinates may take on virtually any physical dimension. However, to establish specific algorithms for the evaluation of f and the various c_i, we first need to settle on an exact system formulation.

This implementation handles the planar case only. Creatures are built from rigid bodies, *limbs*, in a tree-like topology. There is a root limb, which has an absolute position in the plane and an absolute angle. At different points on the root limb, descendant limbs attach through angular joints. The absolute angle θ_j of a descendant j of limb i is represented by the relative angle α_i to its parent limb, i.e., $\theta_j = \theta_i + \alpha_j$.

Thus, if two thin rods are connected through a joint where $\alpha = 0$, the rods will be in alignment. In general, the state of an n-limbed creature is represented by two Cartesian and n angular coordinates.

We can introduce an ordering of the limbs according to a depth-first recursion scheme. If $S(i)$ is the set of limbs directly attached to limb i, and there are no loops, then recursively evaluating S constructs the set of all limbs that descend from limb i. We write $j \succ i$ for all limbs j in that set. Conversely, $j \prec i$ means limb j lies between limb i and the root (or *is* the root). We sometimes refer to limbs ordered in this fashion as lying *inboards* and *outboards*.

Limbs that are mutually independent, such as the two hands of a human, do not enter into this ordering scheme.

The absolute limb angles θ are simple sums,

$$\theta_i = \sum_{j \preceq i} \alpha_j \tag{4.8}$$

and clearly $\frac{\partial \theta_i}{\partial \alpha_j} = 1$ when $i \succeq j$ and 0 otherwise.

The inboards base of each limb i, where it attaches to its parent, is denoted $\bar{b}_i$. Modifying α_i clearly has the effect of rotating the system of limbs $j \succeq i$ around $\bar{b}_i$; thus the dependency of a vector $\bar{p}$ that lies outboards from i is expressed through a rotational operator

$$R(\alpha_i) = \begin{pmatrix} cos(\alpha_i) & -sin(\alpha_i) \\ sin(\alpha_i) & cos(\alpha_i) \end{pmatrix} \tag{4.9}$$

and through it alone. This in turn means that differentiating $\bar{p}$ by α_i affects this R only, i.e.

$$R'(\alpha_i) = \begin{pmatrix} -sin(\alpha_i) & -cos(\alpha_i) \\ cos(\alpha_i) & -sin(\alpha_i) \end{pmatrix}. \tag{4.10}$$

Noting finally that $R' = ER$ with

$$E = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \tag{4.11}$$

we have for $j \succeq i$

$$\frac{\partial \dot{\bar{r}}_j}{\partial \alpha_i} = E[\dot{\bar{r}}_j - \dot{\bar{b}}_i] \tag{4.12}$$

$$\frac{\partial \bar{r}_j}{\partial \alpha_i} = E[\bar{r}_j - \bar{b}_i] \tag{4.13}$$

where $\bar{r}_j$ is the *center of mass* of limb j, a quantity we will use a lot below.

Each rigid-body limb i has two intrisic physical properties we shall use, namely its mass m_i and its *radius of gyration* k_i, a geometric property that essentially measures how reluctant the body is to rotate. An expression for the kinetic energy of this limb is

$$T_i = \frac{m_i}{2}(\|\dot{\bar{r}}_i\|^2 + k_i^2 \dot{\theta}_i^2) \tag{4.14}$$

which also makes clear the nature of k_i.

The muscles of the creature sit at the joint where a limb j connects to its parent i. Whenever it applies a torque $\tau_{i,j}$ to limb i, by the law of action – reaction, it must counter-apply a torque of equal magnitude but opposite sense $-\tau_{i,j}$ working on limb j. Now since we have chosen the generalized

coordinate α_j to encode the angle *between* the two limbs, the associated generalized force Q_j corresponds precisely to a such a pair of opposite torques.

This means that a muscle between limbs i and j contributes to Q_j only and, conversely, the Q_j represents exactly this kind of relative torque. We will refer to muscle-tension values simple as τ_j.

Apart from muscles, the system is influenced by a set of Cartesian forces and impulses $\bar{F}_i$ and $\bar{J}_i$, that work directly on points defined on the creatures. The effect of these are given by (3.11) and (3.14). Finally, gravity is a conservative force-field and so we include it in the scalar potential part V of the Lagrangian $L = T - V$.

4.3 Designing a Solution

Further design issues may now be settled by deciding on how the user would go about setting up a control problem;

- The abstract environment is created and configured. The level of gravitation is set. Each creature is constructed from a root point by attaching limbs outwards, building the tree structure. Cartesians are explicitly allocated and associated with the position of the root; likewise angles with the limbs. At this point, no reference to time has been introduced.

- Next, stages are defined. The length of time of the stage is declared, as well as the number of polynomial pieces N into which to cut it.

- On each stage, representations must be declared for every DOF q_j allocated for creatures in the first item, above. All piecewise polynomials defined on a stage share the same N. These representations automatically become variables of the optimization problem. The two most common representations are possibly that using the Hermite basis, and the constant one, which allocates no optimization variables at all.

- Any muscles that the creatures should have on each stage are declared, and explicit representations are allocated for them, similar to the process above.

- Any task-defining constraints are declared for each stage. For example, in the first stage of the sequence, one would normally specify the full state and its derivative; all the $q_i(t_0)$ and the $\dot{q}_i(t_0)$.

- Any active *forces* acting on the stage are declared and represented explicitly with functions, just as above. The force is then applied to a specific point on a specific limb. Any momentum blows to occur at

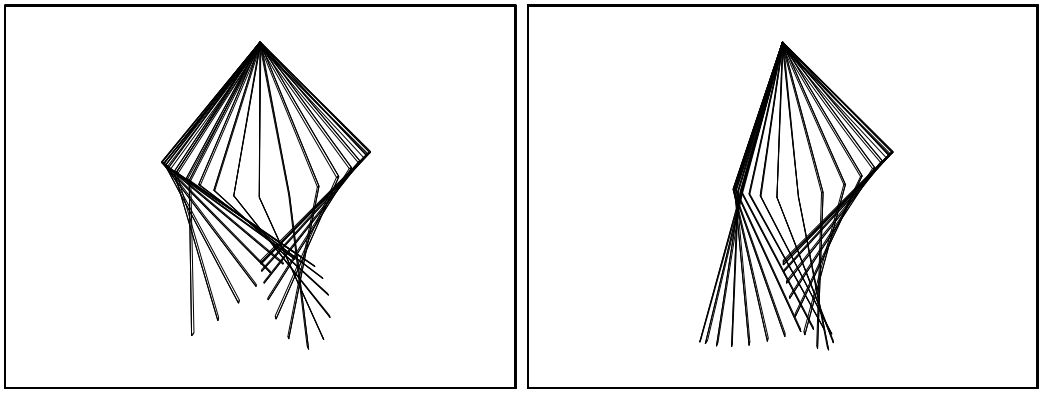

Figure 3: A double pendulum; simulation and control

the end of each stage are similarly declared and directed. These can be constant, or more commonly, variables in the optimization problem.

4.3.1 An Example Scenario

For an illustrative example, consider a short simulation of a double pendulum. The world is given a basic gravity of $-9.81\,m/s^2$. Two long thin rods A and B created, one meter long each, and angular DOF α and β represent their positional state. Cartesians X and Y are allocated and rod A is attached to the root (X, Y). Rod B is attached to the other end of rod A.

Next, a single stage is defined of duration 1 second, with $N = 6$. Because the pendulum is supposed to hang, not fall, we represent X and Y with constants zero, but α and β are made fully Hermite. Finally, constrain

$$\alpha(0) = -\frac{\pi}{4}, \qquad \dot{\alpha}(0) = 0, \tag{4.15}$$

$$\beta(0) = -\frac{\pi}{2}, \qquad \dot{\beta}(0) = 0, \tag{4.16}$$

and the problem is fully determined. The result can be seen in Figure 3. This is an initial-value simulation without any control over the motion.

It is illustrative to analyze this example further. After the discretization, the functions $\alpha(t)$ and $\beta(t)$ are each represented as sums of $2N + 2$ basis functions, yielding a total of $4N + 4 = 28$ free variables in the optimization. The equations of motion (3.9) are evaluated for α and β respectively using the $2N$ basis functions that are zero at the boundaries of the stage interval, for a total of $4N = 24$ constraints. Finally adding the four initial-value constraints total up to 28, matching the number of variables to solve for. The problem is uniquely determined, as one would hope! The objective function never comes into play, here. There is only one solution.

This can be made a control problem by adding a muscle in the joint between the rods and,for example, demanding that at the end of the motion, the two rods be aligned and their relative angular velocities zero, i.e.

$$\beta(t_1) = 0,\ \dot{\beta}(t_1) = 0. \tag{4.17}$$

If the muscle is represented in the Hermite basis, another $2N+2$ variables are needed to represent it, but only two additional constraints are introduced. So, the problem is now underconstrained, and some objective function must come into play to drive the solution towards minimal effort.

One solution is displayed in Figure 3. As mentioned earlier, this is only guaranteed to be a *local* minimum. There may well be some other way to accomplish this task with even less effort.

4.3.2 Objective Functions

Earlier in this paper we discussed how to discriminate between the infinity of valid motions our control problem generates, in an attempt to pick a plausible, realistic-looking one. It was argued that a sensible heuristic would be to attempt to minimize some measure of the metabolic energy consumed in the creature's muscle. The time has come to pick a good measurement.

One possibility that springs to mind is to take the basic physical definition $dW = \sum_j \tau_j \, d\alpha_j$ and integrate to get

$$f = \Delta W = \sum_j \int_t \tau_j(t) \, \dot{\alpha}_j(t) \, dt, \tag{4.18}$$

the amount of energy added to the system by the muscles.

Unfortunately, by this f, there is no cost associated with an energy-conserving action such as standing up and then sitting down. Muscles do not really work that way. If they did, nobody would get tired from doing push-ups or lifting heavy weights, so it seems (4.18) is not a very good measurement of metabolic energy consumption.

Another possibility would be to minimize the (square of the) norm of the physical power consumption $\frac{dW}{dt}$, i.e.

$$f = \left\| \frac{dW}{dt} \right\|_2^2 = \sum_j \int_t (\tau_j(t) \, \dot{\alpha}_j(t))^2 \, dt \tag{4.19}$$

which seems to be the objective function used in [1]. However, this one is also a bit suspect. By this function, push-ups do tire a person, but holding a piano over one's head does not.

The third and last possibility we consider is

$$f = \sum_j \|\tau_j\|_2^2 = \sum_j \int_t \tau_j^2(t)\, dt, \tag{4.20}$$

i.e. minimize the torque magnitudes, the muscle-tensions, over the motion. This function is used in [5] and the authors refer to the paper [19] to provide factual support for using this measurement. It is the function we choose to implement, and it was used to produce Figure 3.

It is useful to remember that in the end, the correct definition of f is an entirely subjective matter. The stated desire is visual realism, and that is not a well-defined notion.

4.3.3 Physics-enforcing Constraints

The majority of the work involved in this implementation is spent on the physical constraints intertwining forces and trajectories. The FEM has already been presented, and the remaining task is to identify and analyze the physical values that must be combined into the equations (3.29).

The three quantities in (3.9) break down as follows,

- $\frac{\partial L}{\partial \dot{q}_i}$, generalized momentum, is entirely due to the kinetic energy T component of the $L = T - V$, because V does not depend on velocities.
- $\frac{\partial L}{\partial q_i}$ has the dimension of generalized force. From V there is the contribution of gravitational force, and from T, there are inertial terms due to the curvature of the coordinate system.
- Q_i, finally, collects forces other than gravitation, i.e. the creature's muscles and the effect of the Cartesian forces working on it.

To compile these equations of motion, then, requires computer-code that evaluates the derivatives of T and V relative to the q_i and those of T to $\dot{q}_i$, and also distributes the effects of forces according to (3.11).

4.3.4 The Dynamics Algorithm

Consider a creature and one of its bodyparts i. Allow joint-angle α_i to vary, but fix all other DOF. This is the situation in which partial differentiation takes place. It is clear that only the portion of the creature that lies outboard from i is dependent on α_i. The kinetic energy of this portion is

$$T_i^{\Sigma} = \frac{1}{2} \sum_{j \succeq i} m_j \left\{ \|\dot{\bar{r}}_j\|^2 + k_j^2 \dot{\theta}_j^2 \right\}. \tag{4.21}$$

Differentiation with respect to joint-angles and joint-velocities yields

$$\frac{\partial T}{\partial \alpha_i} = \frac{\partial T_i^\Sigma}{\partial \alpha_i} = \sum_{j \succeq i} m_j \dot{\bar{r}}_j \frac{\partial \dot{\bar{r}}_j}{\partial \alpha_i} \tag{4.22}$$

$$\begin{aligned} \frac{\partial T}{\partial \dot{\alpha}_i} = \frac{\partial T_i^\Sigma}{\partial \dot{\alpha}_i} &= \sum_{j \succeq i} m_j \left\{ \dot{\bar{r}}_j \frac{\partial \dot{\bar{r}}_j}{\partial \dot{\alpha}_i} + k_j^2 \frac{\partial \dot{\theta}_j}{\partial \dot{\alpha}_i} \dot{\alpha}_j \right\} = \\ &= \sum_{j \succeq i} m_j \left\{ \dot{\bar{r}}_j \frac{\partial \bar{r}_j}{\partial \alpha_i} + k_j^2 \dot{\alpha}_j \right\} \end{aligned} \tag{4.23}$$

where we have used the identity $\frac{\partial \dot{\bar{r}}_j}{\partial \dot{\alpha}_i} = \frac{\partial \bar{r}_j}{\partial \alpha_i}$ which we do not motivate but which holds true for this $\bar{r}$.

Our task is now to derive recursive formulations for these values. This will allow an algorithm to traverse the tree-structure of the system's bodies and compute all the values needed in the equations of motion (3.9) in time linearly proportional to the number of bodies in the system[1].

Using (4.12) we rewrite (4.22)

$$\begin{aligned} \sum_{j \succeq i} m_j \dot{\bar{r}}_j \frac{\partial \dot{\bar{r}}_j}{\partial \alpha_i} &= \sum_{j \succeq i} m_j \dot{\bar{r}}_j \, E[\dot{\bar{r}}_j - \dot{\bar{b}}_i] = \\ &= \sum_{j \succeq i} m_j \dot{\bar{r}}_j \, E[\dot{\bar{r}}_j] - E[\dot{\bar{b}}_i] \sum_{j \succeq i} m_j \dot{\bar{r}}_j = -E[\dot{\bar{b}}_i] \, \dot{\bar{K}}_i^\Sigma \end{aligned} \tag{4.24}$$

where we have used the fact that $E[\bar{v}] \cdot \bar{v} = 0$ in general, and introduced the vector quantity $\bar{K}_i^\Sigma$, which has a very simple recursive update,

$$\bar{K}_i^\Sigma = \sum_{j \succeq i} m_j \bar{r}_j = m_i \bar{r}_i + \bar{K}_{S(i)}^\Sigma, \tag{4.25}$$

and an equally simple physical interpration; namely the centre of mass of the system of bodies $j \succeq i$, multipled by its mass,

$$M_i^\Sigma = \sum_{j \succeq i} m_j = m_i + M_{S(i)}^\Sigma. \tag{4.26}$$

Noting finally that the velocities $\dot{\bar{r}}_i$ never depend on the Cartesian root position of the creature, and thus $\frac{\partial T}{\partial x} = \frac{\partial T}{\partial y} = 0$ completes the recursive

[1] While the algorithm does run in linear time, it is obviously not possible to construct a dense Jacobian in less than quadratic time. However, in this implementation, that task falls to ADOL-C.

formulation of (4.22). Next, (4.23) decomposes similarly;

$$\sum_{j\succeq i} m_j \left\{ \dot{\bar{r}}_j \frac{\partial \bar{r}_j}{\partial \alpha_i} + k_j^2 \dot{\alpha}_j \right\} =$$

$$= \sum_{j\succeq i} m_j \left\{ \dot{\bar{r}}_j \cdot E[\bar{r}_j - \bar{b}_i] + k_j^2 \dot{\alpha}_j \right\} =$$

$$= \sum_{j\succeq i} m_j \left\{ \dot{\bar{r}}_j \cdot E[\bar{r}_j] + k_j^2 \dot{\alpha}_j \right\} - E[\bar{b}_i] \sum_{j\succeq i} m_j \dot{\bar{r}}_j =$$

$$= U_i^\Sigma + P_i^\Sigma - E[\bar{b}_i] \cdot \dot{\bar{K}}_i^\Sigma \quad (4.27)$$

where

$$U_i^\Sigma = \sum_{j\succeq i} m_j \dot{\bar{r}}_j \, E[\bar{r}_j] = m_i \dot{\bar{r}}_i \, E[\bar{r}_i] + U_{S(i)}^\Sigma \quad (4.28)$$

$$P_i^\Sigma = \sum_{j\succeq i} m_j k_j^2 \dot{\theta}_j = m_i k_i^2 \dot{\theta}_i + P_{S(i)}^\Sigma. \quad (4.29)$$

The dependency of T on the root *velocity* is seen to be

$$\frac{\partial T}{\partial \dot{x}} = \sum m_i \dot{\bar{r}}_i \frac{\partial \dot{\bar{r}}_i}{\partial \dot{x}} = \sum m_i \dot{\bar{r}}_i \cdot \hat{x} = \hat{x} \cdot \bar{K}_{\text{root}}^\Sigma \quad (4.30)$$

and likewise for $\hat{y}$, completing the differentiation of T. To account for gravity,

$$\frac{\partial V}{\partial \alpha_i} = G \cdot \frac{\partial}{\partial \alpha_i} \left\{ \hat{y} \cdot \sum m_j \bar{r}_j \right\} =$$

$$= G \cdot \hat{y} \cdot \sum_{j\succeq i} m_j E[\bar{r}_j - \bar{b}_i] = G \cdot \hat{y} \cdot E \left[\sum_{j\succeq i} m_j \bar{r}_j - b_i \sum_{j\succeq i} m_j \right] =$$

$$= G \cdot \hat{y} \cdot E \left[\bar{K}_i^\Sigma - b_i M_i^\Sigma \right] \quad (4.31)$$

Finally, the Cartesian forces $\bar{F}_i$ working on points $\bar{p}_i$ transform into the generalized system of coordinates according to (3.11), i.e.

$$Q_i = \sum_j \bar{F}_j \cdot \frac{\partial \bar{p}_j}{\partial q_i} \quad (4.32)$$

which need be written on a recursive formmat similar to those above in order for the linear-time traversal algorithm to compute it. Expanding again

by (4.12) and reordering yields

$$Q_i = \sum_{j \succeq i} \bar{F}_j \cdot E[\bar{p}_j - \bar{b}_i] =$$
$$= \sum_{j \succeq i} \bar{F}_j \cdot E[\bar{p}_j] - E[\bar{b}_i] \cdot \sum_{j \succeq i} \bar{F}_j =$$
$$= \bar{\mathcal{F}}_i^{\Sigma} - E[\bar{b}_i] \cdot \bar{F}_i^{\Sigma} \quad (4.33)$$

with

$$\bar{\mathcal{F}}_i^{\Sigma} = \sum_{j \succeq i} \bar{F}_j \cdot E[\bar{p}_j] = \bar{F}_i \cdot E[\bar{p}_i] + \bar{\mathcal{F}}_{S(i)}^{\Sigma} \quad (4.34)$$

$$\bar{F}_i^{\Sigma} = \sum_{j \succeq i} \bar{F}_j = \bar{F}_i + \bar{F}_{S(i)}^{\Sigma}. \quad (4.35)$$

It should be mentioned that we have taken some liberties here with the ordering. The $j \succeq i$ notation was defined for limbs, and there may be several forces acting on different points on a limb. This means that there is no exact order of forces. However, in the two recursive formulations above, only the order of forces *within* a limb is undefined, and that order does not matter to the computation of the values.

The algorithm now proceeds by traversing each creature from the root out to the tips, building the θ_i, the $\bar{b}_i$ and other kinematic quantities, then synthesizing the more complex, recursive expressions back towards the root, generating the target quantities for (3.9) in the process.

4.4 Assembling the Solution

Clearly, numerical integration techniques must be applied to equations of motion (3.9) and the objective function (4.20).

For simplicity, we will be using Simpson's rule. Consider a function f defined on $[a, b]$. Divide this interval into n pieces and write $f_i = f(a + ih)$ where $h = (b - a)/n$. Then

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left(f_0 + 4f_1 + 2f_2 + 4f_3 + \ldots + 2f_{n-2} + 4f_{n-1} + f_n\right) \quad (4.36)$$

approximates the integral with an error that behaves like h^4 as $h \to 0$. In our implementation, the n is hardcoded in the program.

When the optimizer calls us to fill in its values, the following takes place. The objective function f and the c_i are all initialized to zero, and we loop

over each stage, where all the work is done. Each stage then loops both over the N polynomial pieces into which it is split, and inside this loop, over the n sampling points dictated by Simpson's rule above. At each of these $n \cdot N$ sampling points t, a vast hierarchy of expressions is constructed.

First, the q_i, $\dot{q}_i$, the muscle tensions and the magnitude of active forces are explicitly evaluated from their representations at t. Then, the dynamics algorithm outlined above is executed, recursing through the creatures, generating the kinematic entities on its way out and synthesizing the kinetics coming back.

We now have a snapshot of the system, at instant t. While the recursive algorithm ran and expressions were evaluated, the automatic differentiation package *ADOL-C* built an exact record of how each final expression was constructed and how it depends on each input. This information is included in the snapshot.

Numerical integration next proceeds by adding the contribution of the system at t, according to the snapshot we just took. Like everything else, this addition takes place under the supervision of ADOL-C, so the integral values slowly accumulate an enormous tree of dependency information.

Finally, remaining constraints are evaluated, those that are not equations of motion. These are usually simple, governing a single value at a single instant of time. An exception are the constraints that govern the transitions from one stage to the next. These enforce the continuity of the coordinate trajectories and also of their generalized momentums, taking into account also the effects of any *impulses* active at the end of the current stage.

4.4.1 Variable Stage Lengths

In all of the above, the length T_k of some stage k has been considered static and part of the problem formulation, but this need not be so: T_k can be made variable as well, solved for along with everything else in the optimization problem.

In fact, when there is a definite task to be performed, such as "kneeling", it seems pointlessly restrictive to decree that the movement take exactly 0.3 seconds. Humans who are asked to kneel will do so at a sensible pace. Too quickly, and the downward velocity becomes large and muscles must work very hard to stop the motion fluently. However, a very slow pace is no better, because one would spend several seconds with knees painfully half-bent, supporting all one's body weight.

In this example, then, the objective function is *convex* as a function of T_k, punishing any motion with too long or too short a timeframe. This gives a stable problem, and convergence is obtained. However, there are scenarios

where a variable T_k leads to trouble.

In any pure simulation problem, for example, the length of simulation is certainly a defining parameter that should be given explicitly. If it is made variable, the problem becomes under-constrained, and there is no objective function to compensate. In fact, this is technically a singular problem, although in practice the solver usually does not encounter this singularity and the iterations do converge.

Another difficulty is when gravity is turned off. Generally, gravity acts as something of a clock in these problems, and without it there is nothing to measure time against. An astronaut performing some task in space would, if energy were at a premium, move exceedingly slowly. Depending on the objective function used, this could occur in abstract reality as well and cause the optimization process to drive the stage length towards infinity; an unstable problem without solution.

In practice, even in well-defined problems, we want to place bounds on the stage-length. Not only do we want to avoid having to consider what happens when $T_k \to 0$ or $T_k \to \infty$, but we also have an idea of approximately what the motion should look like, if it should take a tenth of a second, or a half.

As far as implementation goes, allowing T_k to vary is not difficult, especially using automatic differentiation. In fact, it is sufficient to flag T_k as an active variable, and it is immediately included in all dependency considerations. Stages may be set up either with a fixed or with a variable length in this implementation.

Finally, it should be mentioned that this topic has been explored and implemented previously, in the paper [7] (and probably countless others).

4.4.2 Bounding the DOF

Most of the animation problems we want to solve require bounds on the possible values that the coordinate trajectories can take. Sometimes the bounds are absolutely essential. Natural-looking movement evolved around the biological limits of joints, and we have no chance of emulating it if we cannot enforce similar boundaries in our problems.

Unfortunately, this is non-trivial for the Hermite basis. Recall that within any given polynomial piece, a function represented in this basis is the sum of four non-zero functions. We want to bound this sum not only at one point, or two, but continuously, over the whole stage;

$$f_{\min} \leq \mathcal{P}_i^f(u) \leq f_{\max}, \quad u \in [0,1], \quad 1 \leq i < N. \tag{4.37}$$

Recall Figure 2 and the properties (3.23). Bounding the function at $u = \{0,1\}$ is simple; two basis functions interpolate the function there so $f_{\min}$ and

$f_{\max}$ can be used as discrete bounds on the two corresponding coefficients.

Bounding the interior is trickier, and in fact the solution we give for this thesis is less than completely satisfactory. We sample (4.37) at $u = \{\frac{1}{3}, \frac{2}{3}\}$ and enforce the bounds there through explicit constraints. A cubic can attain at most one maximum and one minimum, and is very regular. Being bounded at four points, it is difficult for the function being represented to violate the boundaries to any large degree *between* those points. Thus, the solution *is* largely successful. Nevertheless, it is ad-hoc and the bounds *can* be violated to some degree.

This is unfortunate, because the optimizer is very good at exploiting just this kind of weakness. If violating the joint-boundaries lowers the value of the objective function a great deal, and the only way to violate the boundaries is, say, to oscillate severely, then severe oscillation is exactly what we will get. In practice, it is not difficult to spot this problem when it occurs in an animation. In these cases, one increases the number of sample points to e.g. $\{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}\}$, tightening the bound further at some cost in execution time.

4.4.3 Muscles: Implicit vs Explicit

In the process outlined so far, muscles are explicitly represented in the optimization problem as DOF in their own right. The laws of mechanics intertwine the q_i and the Q_i in the form of constraints, and the optimization objective becomes a function of the Q_i. While this is a sensible way to think about the situation, for computational purposes it could be considered slack. Introducing a slew of variables Q_i that are allowed to vary with complete freedom and then constraining them utterly relative to the q_i, more than doubles the size of the optimization problem.

The fact is, essentially the same equations that constrain the q_i and the Q_i can be used to *calculate* the Q_i from the q_i. Thus, instead of an objective function that depends on variables Q_i that in turn depend completely on q_i, the former could be entirely eliminated. Can there be any justification for explicitly representing the Q_i as we do?

The answer is not clear. If the Q_i are eliminated, the size of the problem certainly decreases dramatically, but its *density* increases correspondingly. In fact, cutting out the Q_i does not so much reduce the problem as compact it. At the heart of HQP sits a sparse linear equation solver, the running time of which depends on the number of non-zero matrix elements rather than the actual size of the matrix.

The question of whether large and sparse is better than small and dense is common to many technical applications like this one, and the answer often depends on the computing hardware and numerical algorithms at hand.

Using ADOL-C, an implicit-muscle implementation is surprisingly simple. The means of computing Q_i are in the basic Euler-Lagrange functions (3.1) themselves. In the case of the angles α_i, substituting the recursive relationships already derived above cancels out beautifully to

$$\begin{aligned} Q_i = \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\alpha}_i}\right) - \frac{\partial L}{\partial \alpha_i} = \\ = \frac{d}{dt}\left(U_i^\Sigma + P_i^\Sigma - E[\bar{b}_i]\cdot \dot{\bar{K}}_i^\Sigma\right) - \left(-E[\dot{\bar{b}}_i]\,\dot{\bar{K}}_i^\Sigma\right) + \frac{\partial V}{\partial \alpha_i} = \\ = \dot{U}_i^\Sigma + \dot{P}_i^\Sigma - E[\bar{b}_i]\cdot \ddot{\bar{K}}_i^\Sigma + G\cdot\hat{y}\cdot E\left[\bar{K}_i^\Sigma - b_i M_i^\Sigma\right] \end{aligned} \quad (4.38)$$

where $\dot{U}_i^\Sigma$, $\dot{P}_i^\Sigma$ and $\ddot{\bar{K}}_i^\Sigma$ are new and must be calculated. Their recursive definitions are trivial, as is the case of the Cartesians.

If the Q_i are eliminated, what happens to the equations of motion? Their role is largely to constrain the relationship between the q_i and the Q_i, and now there are no Q_i to constrain. Getting rid of the equations of motion seems odd, but it is not really. A creature with muscles in every joint has perfect control over the movements of its limbs relative each other, and there *is* no physically invalid motion. There is no need to constrain the q_i.

The only relevant matter then is how much work the muscles must perform for some specific motion, i.e., the calculation of the objective function from the q_i. This was previously a trivial function of the τ_j, but now contains all the complexity of the Lagrange equations.

The only real difficulty in writing such an implementation arises when there is some joint i where there should *not* be a muscle, where the connecting limb should dangle freely. This "lameness" must then be explicitly enforced in a constraint, i.e.

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\alpha}_i}\right) - \frac{\partial L}{\partial \alpha_i} = 0. \quad (4.39)$$

There was not time to include this in the implementation. However, it would make for an interesting comparison, and is something to consider as future work.

4.4.4 Visualization

When the solution is found, the true output consists of values for the optimization variables. However, since this is not much to look at and besides, the program already knows what the values represent and how the creatures are constructed, it makes sense to take advantage of this by also generating output with more structure.

There are many ways of doing this, for this implementation, we choose to output scenery information on the RenderMan™ format designed by Pixar Inc. This is a standard language for describing 3D scenes, frame by frame. It was designed for computer animation. The Blue Moon Rendering Tools by Larry Gritz, with a non-commercial license, accepts RenderMan™ output and renders it on-screen or in image files. Such files can later be compressed into movies.

There are other output formats one could choose. The Virtual Reality Modeling Language (VRML), for example, would certainly be up to the task. It should not be difficult to write additional modules for generating different kinds of output.

5 Implementation

Writing the actual code follows quite easily from what has come before. We choose to work in C++. This is the language the HQP interface Omuses was designed for, also, the problem structure works well with object-oriented (OO) design.

Throughout this project, a pragmatic relationship to computer science has been maintained. The implementation is not beautiful. It violates many basic tenents of object-oriented programming: objects are used largely as data dumps. The classes are not designed as "black boxes". Most of their variables are public, and objects write in each other's data spaces with little hesitation.

Thus the structure of the problem is not mirrored in the design of the C++ classes as much as it could be. Nevertheless, there *is* discipline in how the code behaves and a well-defined order in which things are evaluated. The automatically differentiated expressions are built layer by layer, from optimizer input to optimizer output.

5.1 Tree of C++ Classes

- Class *World* is a global nexus. It is the first object created when a new problem is set up. Subsequently, it is where *Creatures*, *DOFs*, and *Stages* automatically register upon creation. The level of gravitation is kept here. This is also where all other objects request allocation of space in the optimization-variable and constraint arrays. In the main update loop, class *World* calls upon the *Stages* to do the real work, and then calls the *Visualizer* module to generate visual output.

- Class *Stage* holds lists of *Muscles*, *Forces*, and *Constraints* active over each stage, *Impulses* to apply at the end of it, and *Funs* that do the actual representation of time-varying functions on the stage. These all register with the *Stage* when they come into being. It also defines the length T of the stage, the number of polynomial pieces N and the handy quantity $h = T/N$. In the update loop, class *Stage* performs the necessary calls to all the external objects listed above, and performs the numerical integration of the results. This class registers with class *World* in its constructor.

- Class *RigidBody* implements the dynamics algorithm of this paper. Practically all the quantities calculated in the recursive scheme are class variables, rather than local to the recursion. This allows us to treat them as constrainable quantities. The body also keeps a list of the *BodyPoints* defined on it, and a pointer to the DOF which holds its angle α relative to its parent body. Finally, it has a mass and a (square of the) radius of gyration.

  - Class *Sphere* is a convenient *RigidBody*.
  - Class *ThinRod* is a convenient *RigidBody*.
  - Class *Disk* is a convenient *RigidBody*.
  - Class *Cylinder* is a convenient *RigidBody*.

- Class *AnchorPoint* represents a point in the plane, a vector quantity that is a function of the DOF q_i. It can be used as an expression in constraints, but exists mainly as an anchoring place where *RigidBodies* connect to each other, through the use of sub-class...

  - Class *BodyPoints* is an *AnchorPoint* that also represents a point on a *RigidBody*. As such it has a position (x, y) defined within the local coordinate system of the body on which it is defined. *RigidBodies* connect to *AnchorPoints* through *Bodypoints*.
  - Class *Creature* is fundamental. It implements the root of any creature and as such is the only entity that references DOF X and Y. It is an *AnchorPoint* without a body, so for the creature to have a real physical existance, at least one *RigidBody* must attach to this *Creature* point. This class registers with class *World* in its constructor.

- Class *DOF* represents a generalized coordinate. On each stage it must have a *Fun* representating it. It defines the variables q, $\dot{q}$, $\ddot{q}$, $\frac{\partial T}{\partial \dot{q}}$, $\frac{\partial T}{\partial q}$, Q,

and P (from (3.15)), which are calculated during the snapshot process and written into the *DOF* instances. This class registers with class *World* in its constructor.

- Class *Muscle* simply coordinates a *Fun* with an angular *DOF* and lets the former apply a torque to the latter. This class registers with class *Stage* in its constructor.

- Class *Force* applies a force to an *AnchorPoint* in a predefined direction. The magnitude may be flagged as a variable of the optimization problem. This class registers with class *Stage* in its constructor.

- Class *Constraint* is an abstract class. Its reason for existing is to guarantee to whatever references it that it will be able to evaluate the constraint it represents.

  - Class *ValConstraint* implements class *Constraint*. It accepts a reference to an ADOL-C-supervised variable and constraints it according to however the class is configured. The value it watches can represent virtually any quantity computed during the snapshot process. Simple examples include DOF q_i, the positions of *AnchorPoints*, and impulse magnitudes. This class registers with class *Stage* in its constructor.

  - Class *VecConstraint* implements class *Constraint*. Its operation is much like *ValConstraint*, but it works on vector quantities (of size two).

  - Class *Link* represents the transition between two stages A and B. For each *DOF* in these stages, it enforces continuity of trajectory (q) and generalized momentum ($\frac{\partial T}{\partial \dot{q}}$) unless the representations of the *DOF* are constant on both stages. This class registers with class *Stage* in its constructor.

- Class *Fun* is an abstraction of a real-valued function defined on a stage. Its value for each point in time t is constructed during the snapshot process and may be the target object of *Constraints*. It can be used to represent *Muscles*, *Forces*, and *DOFs*. This class registers with class *DOF* or class *Stage* in its constructor.

  - Class *Hermite* implements class *Fun*. It uses the Hermite basis that we have covered in some detail. Upon creation it automatically allocates $2N + 2$ optimization variables, possibly $2N$ constraints for the FEM equations of motions, if it is representing a *DOF*,

and possibly $2N$ constraints for the bounds, if it is bounded. This class registers with class *Stage* in its constructor.

  - Class *Hat* implements class *Fun*. It is the hat basis of (3.27). It works much like *Hermite*, but it is simpler. For example, it does not need the extra sample constraints for the bounding; constraining the coefficients themselves suffices.
  - Class *Single* implements class *Fun*. It allocates a single optimization variable, which holds its value over the stage. Thus it is constant, seen as as a function of time, but constant at a level that is variable from the optimizer's perspective. The value k in the trivial control-example in the second chapter of this thesis would be represented with an instance of this class.
  - Class *Constant* implements class *Fun*. This is a truly constant function, its value set when it is created.

- Class *Visualizer* is called to generate structured output, currently on the RenderMan™ format. It performs a sequence of snapshots, evenly spaced in time across the length of the interval and over the stages (which may be of variable length) and generates an animation frame for each such point in time by recursing over the *Creatures* in the system.

6 Results

The aim of this thesis was to write an implementation of the Spacetime Constraints paradigm and to make it available to the public. The code and the theory behind it – as described above – constitute the main result and accomplishment of the endeavour. However, in the process of writing it, some example outputs have also been collected that could be considered results, and these are described and presented here.

6.1 The Link Chain

The simplest example of a multibody problem is probably the single chain of links, anchored somewhere in the plane. The case of two links, for example, is known as a double pendulum and is a classic problem in mechanics.

Figure 4 depicts a triple-linked chain with muscles both at the shoulder and at each joint. The chain starts out hanging straight down and we ask that it straightens to the right in half a second. The total length of the chain is two meters, normal gravity is active, and joint angles are restricted to lie within $[-\pi, \pi]$. This is a simple problem and a solution is found fairly

quickly. Even so, starting the iterative solution process at slightly different initial guesses yields two very different solutions.

The requirements made on the two solutions displayed are identical, but the optimizer is started off with different initial values for its variables, and so it initially sees different portions of the search-space. It follows a minimizing path as best it can using its linear/quadratic approximations, but lacks any global perspective; it cannot see over search-space ridges even if there is a great abyss on the other side, and so can easily get stuck in little potholes.

The initial variable configuration of the first run corresponds to a sequence where the chain swings at a constant angular velocity, without bending at the joints, from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$. In the second, the root link moves instead from $-\frac{\pi}{2}$ to 0, and the two outer links curve slowly upwards. In both solutions, the creature attempts to mimimize its torque at the root by curling up before extending, but in the second, it comes up with an even more clever use of its resources. The value of the objective function in the second solution is substantially lower than in the first. We will let this illustrate the problem of global optimization.

6.2 The Ubiquitous Desk Lamp

In 1986, Pixar turned heads with the movie "Luxo Jr.", starring an animated desk-lamp. In their 1988 paper, Witkin and Kass again used this model for demonstrating their method. The leaping desk-lamp has become an archetype, and this paper would not feel complete if it did not reproduce their result.

Figure 5 outlines the animation. Luxo is constructed from simple cylinders and a sphere, made from materials of various densities. Joint bounds

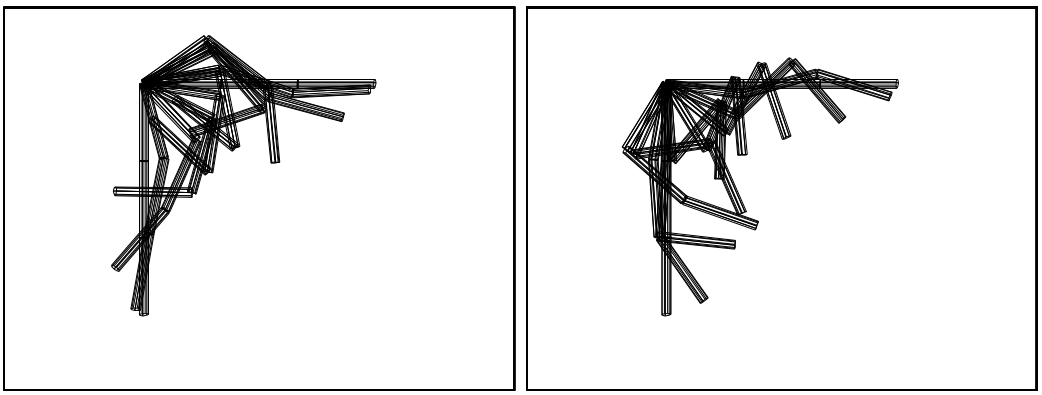

Figure 4: A three-limbed arm stretching: two local minima

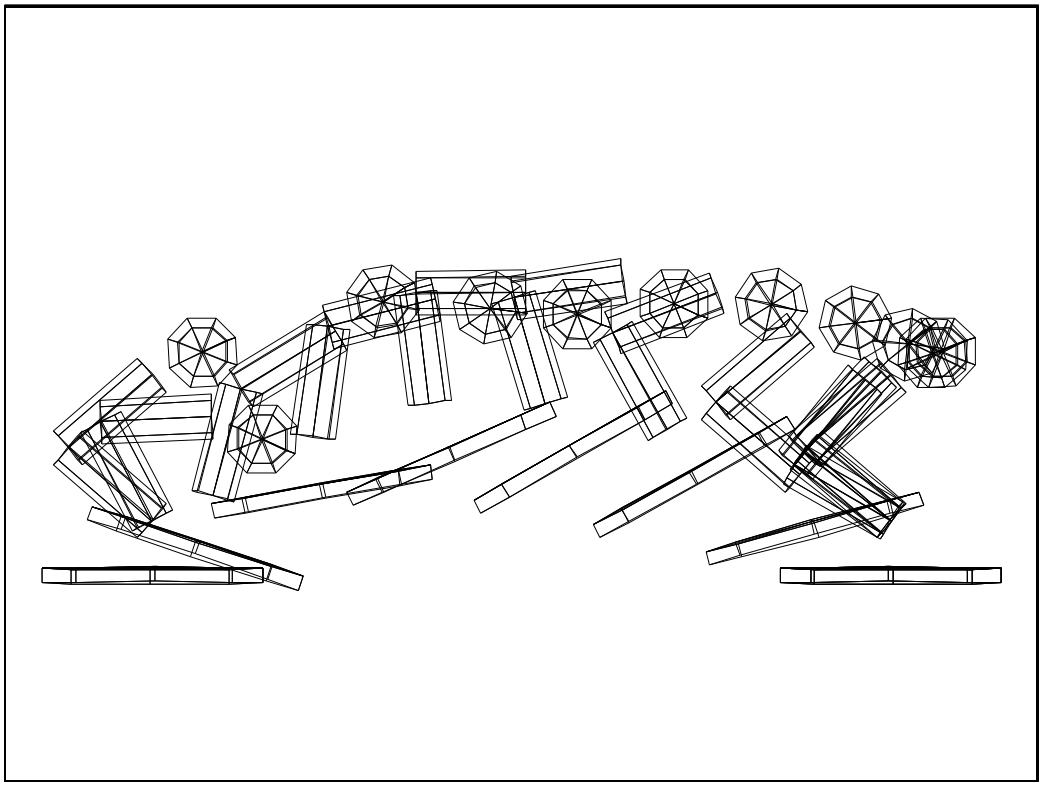

Figure 5: A leaping desk lamp

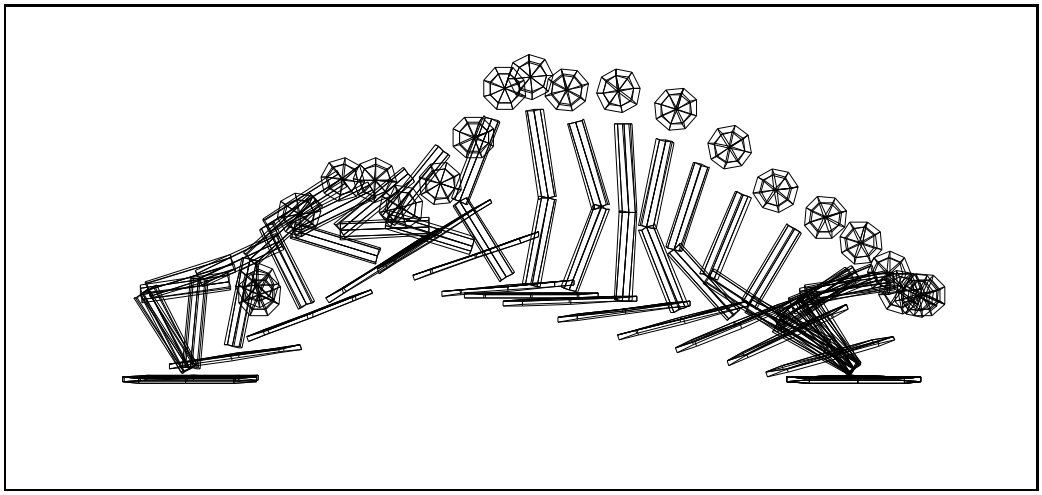

Figure 6: A longer leap

are very important in this problem, especially if one wants to maintain an anthromorphic attitude towards the lamp. For example, unless forced not to, Luxo will bend his "spine" back without compunction, and this seems to grate on the human eye.

Luxo's jump is a three-stage problem; in the first stage, the base is nailed to the ground by representing the position and the attitude of the base as constants. This lets Luxo gather momentum in his upper body. In the second stage, Luxo flies, and all DOF are fully Hermite. The third stage is much like the first, except that the base is nailed down at a different place, forcing him to make the jump to get there. Luxo's pose is constrained at the beginning of the first stage and at the end of the third.

Between the second and third stages, three impact vectors are applied to the base, to represent the effect of Luxo slamming inelastically into the table. One acts along the $\hat{x}$ axis, representing instant friction. The other two are applied to the front and back edge of the base and are directed along $\hat{y}$, these work together to stop any part of Luxo from falling through the ground. The magnitude of these impulses are variable, solved for along with the rest of the problem.

These variable impact vectors are vital. They introduce freedom over the stage transition. Without them, $\mathcal{C}^1$ continuity would be enforced and Luxo would have to maneuver in mid-flight in such a way as to set the base down smoothly, without any impact. This would completely change the nature of the animation. Of course, it is a simple matter to constrain the magnitude of the impulse so that Luxo has to perform a gentle touchdown. Likewise, forcing him to come down hard will probably cause him to jump higher first.

Figure 6 shows a longer jump, from earlier in the implementation effort, where a flaw in the joint-angle bounds lets Luxo get away with an awkward-looking bend in his "spine".

6.3 Field Survey and Comparison

In 1992, Michael F. Cohen published [2], which reproduces and extends the original work of Witkin and Kass. Cohen represents DOF as piecewise cubic polynomials, in a *B-spline* basis. This basis spans a smoother function space than does the Hermite one we use, and its functions have wider support, but it is otherwise similar.

Perhaps the most attractive feature of Cohen's code is the integration with a visual, interactive environment, where results are displayed as iteration progresses and the animator stays in control throughout the process.

In comparison, the implementation we present in this thesis allows no control once the optimization process starts. There is no integrated visual

environment, although output is generated at every iteration, which can be displayed simultaneously using some graphical program.

6.3.1 Symbolic Algebra

Like Witkin and Kass, Cohen uses symbolic algebra techniques to generate most of the complex expressions that are computed implicitly by the dynamics algorithm in this thesis. Such symbolic methods essentially compile, from a problem description, explicit expressions in the problem's defining variables for all the values that will need to be computed during the optimization process. Typically, symbolic compilation happens once, perhaps as a separate step performed before the actual solver code is run.

In contrast, automatic differentiation as performed by ADOL-C is entirely a run-time affair, consisting essentially of a string of multiplications by numerically computed Jacobians. There are more similarities than differences between these two methods, however. Both take over the task of computing derivate information, allowing the system implementor to concentrate on defining the problem correctly in terms of its governing equations. Both can exact a heavy toll in decreased efficiency.

The efficiency problem is redundancy – automatic methods must be explicitly programmed to perform algebraic simplifications that humans do without much meditation. These can range from trivial, such as $x \cdot 1 = x$, to the kind of complex manipulation mathematicians thrive on. Without these reductions, expressions may grow very quickly and become costly to evaluate during the optimization process.

The paper [3], presented at SIGGRAPH '94, further matures Cohen's work by alleviating some of its efficiency problems. The authors borrow from the field of compiler design optimization techniques such as that of *common sub-expression elimination*. By evaluating equivalent expressions only once and reusing the computed value, redundancy is decreased. The authors report greatly improved efficiency.

Nevertheless, the paper [4] from 1995 points out that the basic complexity problem remains even after common sub-expressions are eliminated. The authors present a symbolic language in which joint torques and states, limb positions and angles, and other high-level physical attributes of the system may be referenced as first class variables. This is clearly a good language in which to express e.g. constraints and objective functions.

The language encodes assumptions about the structure of the mechanical system, and the underlying implementation is thus able to use a dynamics algorithm not entirely unlike that presented in this thesis to quickly compute the relevant quantities. The structure assumed in the paper is a tree topology

much like ours, but in a full 3D environment rather than the (very much simpler) planar case with which we have worked.

With this paper, the computation of the Jacobian of the dynamics expressions in a symbolic-algebra implementation achieves the theoretically optimal time complexity of $O(\mathrm{DOF}^2)$ – which our implementation would, as well, were ADOL-C not used.

6.3.2 Hierarchical Basis

The paper [3] introduces another enhancement as well: a *wavelet* transformation of the B-spline basis is performed, with the result that the optimization code is able to affect coarse, sweeping changes on one hand and and precise, local ones on the other, through tweaking different basis coefficients.

The authors report dramatically improved convergence properties after this transformation. A few stabs at wavelet support were made for this implementation, but the time-frame was far too limited. The wavelet transformation becomes useful when the number of piecewise polynomials N used to represent a stage grows large.

6.3.3 Other Spacetime Work

Another exploration of the spacetime control thread is [6]. In this paper, the authors analyze one specific problem (human diving) and make simplifications to the governing equations, in an attempt to avoid the full complexity from which the full space-time formulation suffers but still produce essentially valid motion.

This is in some contrast to the other works, which are very general in their application. The endeavour is successful, resulting in the full somersault sequence being calculated in less than a second.

Martin Preston has implemented a parallel Spacetime solver in [8] where he reports good results.

6.3.4 Decision Trees and Databases

Using a method related to Spacetime Constraints, Pedro S. Huang and Michiel van de Panne apply in [11] classical Artificial Intelligence techniques to the problem of motion control.

Rather than build discrete equations of motion that hold over time, they perform physical simulation forward in time. A *decision tree* is constructed, where nodes represent system states and edges represent control, such as e.g. some configuration of joint torques. Thus from any node at time $t = t_0$, traversing some edge means tensing muscles in some corresponding way for

a discrete period of time. The state represented by the child node is then determinstically computed using physical simulation.

The size of this tree is hugely exponential in the number of time steps. By evaluating nodes according to some fitness criterium, the authors search it for desired motion. Because of the enormity of the tree, pruning methods and heuristics are used to make the search managable. Letting edges represent low-level control such as joint torque, the authors solve highly unstable control problems using this technique, such as that of a single-jointed, two-limbed "acrobat" not only balancing properly, but managing a flip as well.

The edges can equally well represent higher-level control. Thus each decision, each edge-traversal can represent not a simple tensing of some muscle, but a complex combination over a longer period of time. For example, the choice could be between one of many jumps possible from the current state. Searching such a tree constitutes *motion planning*. The authors have Luxo bouncing across rough terrain with peaks and chasms, where the ability of the tree-search algorithm to plan several jumps ahead may be essential.

In the paper [12], Alexis Lamouret and Michiel van de Panne present a related work, revolving around a large database of sample motions for some virtual creature. Thus for some specific creature state and environment state, this database may be queried for motion samples that may possibly suit the setting.

The authors use the terrain-exploring Luxo mentioned above to generate an example database, where the motion primitive of a choice is a single hop. These are classified according to length and height of the jump and stored. Then, as Luxo finds himself in the virtual mountains, he need only search the database for the best motion fit he can find, adapt it somewhat to satisfy e.g. continuity requirements, and then execute it.

7 Conclusions

The idea of using optimal control for realistic-looking motion has some fundamental problems. While it is certainly true that it produces motion that is not only physically valid but also fairly real-looking, the creature does not really move in a way that makes sense to humans. When sequences become longer than a few tenths of a second, the over-optimized nature of the motion starts becoming apparent.

7.1 Over-optimization

Sometimes the best-looking motion is achieved by aborting the process after only a few iterations. In later stages, the algorithm starts finding paths that lower the value of the objective function and do not violate any constraints, yet would never really be displayed by anything living, anything that had to make its decisions in real-time.

Even in an athlete's trained movements, there is built into the body-language an awareness of limitations not only of body but of mind. Human motor control is very reluctant to let joints and muscles get near extreme states, where possible injury always threatens.

The result is that living beings display a conservative, focused use of muscles that optimized spacetime creatures do not display. Unless hemmed in on all sides by constraints, the algorithm will continuously "nudge" the solution until every resource available to the creature is used to its utmost efficiency. The spacetime control approach has all the time in the world to tinker with and make perfect motions that biological creatures have to perform as instant reflexes.

As an example, in several attempts to produce the jump of Figure 5, Luxo insisted on bracing against the inertia of his own head in order to correct his pose in mid-flight, which looks utterly unrealistic. Humans are very protective of their necks and do not lightly allow whip-lash effects to occur. But with Luxo, discouraging such behaviour is difficult. We can penalize heavy use of his neck muscles, and this seems like a good idea. But as it turns out, this makes Luxo reluctant to hold his head steady, and the result is a jump where his neck looks broken.

The way humans hold their heads while jumping simply does not follow from the efficiency rule. We keep our neck muscles tense in a state of anticipation, and this goes for other muscle groups as well. A living creature, ready to pounce into action is not relaxed, but tense.

It is possible that given a sufficiently accurate model of muscles, joints and other purely biological factors, the only fault with the optimized motion would be unrealistic perfection. More complex objective functions can encode more of these factors, probably at a heavy cost in efficiency.

7.2 Motion control

This possibility regardless, high-level control simply is not the strength of this paradigm. There are infinitely complex psychological considerations to any non-reflexive motion. Rather, the most accurate classification of the method omes perhaps through the realisation that, despite all its complexity difficul-

ties, it *is* the simplest method to do what it does.

The problem of "keyframe interpolation", i.e. of generating valid motion over intervals bounded on both sides by complete state descriptions, is fundamental to Computer Graphics animation. Current solutions in industry are impressive but, in the end, compromised. Human tolerance is limited for animation that fundamentally violates our sense of reality. The prettier and more life-like the rendering of creatures becomes, the more disturbing it is when they move wrong. Any progress made towards animation systems that takes better account of the realities of physical law must be greeted with enthusiasm.

References

[1] Witkin, A. and Kass, M., *Spacetime Constraints*, Computer Graphics 22(4) (August 1988), pp. 159-168

[2] Cohen, M. F., *Interactive Spacetime Control for Animation*, Computer Graphics 26(2) (July 1992), pp. 293-302

[3] Liu, Z., Gortler, S. J., Cohen, M. F., *Hierarchical Spacetime Control*, SIGGRAPH Proceedings (1994), pp. 35-42

[4] Zicheng, L. and Cohen, M. F., *An Efficient Symbolic Interface to Constraint Based Animation Systems*, Microsoft Research Technical Report MSR-TR-95-27

[5] Rose, C., Guenter, B., Bodenheimer, B. and Cohen, M. F., *Efficient Generation of Motion Transitions using Spacetime Constraints*, SIGGRAPH Proceedings (1996), pp. 147-154

[6] Zicheng, L. and Cohen, M. F., *Decomposition of Linked Figure Motion: Diving*, 5$^{\text{th}}$ EuroGraphics Workshop on Animation and Simulation (Oslo, Norway, 1994)

[7] Zicheng, L. and Cohen, M.F., *Keyframe Motion Optimization By Relaxing Speed and Timing*, Proceedings of the 6$^{\text{th}}$ Eurographics Workshop on Computer Animation & Simulation, Springer Verlag (1995)

[8] Preston, M., *Parallel Spacetime Animation*, Proceedings of the 6$^{\text{th}}$ Eurographics Workshop on Computer Animation & Simulation, Springer Verlag (1995)

[9] Ngo, J. T. and Marks, J. *Spacetime Constraints Revisited*, SIGGRAPH Proceedings (1993), pp. 343-350

[10] Gritz, L., Hahn, J. K., *Genetic Programming Evolution of Controllers for 3-D Character Animation*, Genetic Programming 1997: Proceedings of the 2nd Annual Conference, pp. 139-146

[11] Huang, P. S., and van de Panne, M., *A Search Algorithm for Planning Dynamic Motions*, Proceedings of the 7th Eurographics Workshop on Computer Animation & Simulation, Springer Verlag (1996), pp. 169-182

[12] Lamouret, A. and van de Panne, M., *Motion Synthesis By Example*, Proceedings of the 7th Eurographics Workshop on Computer Animation & Simulation, Springer Verlag (1996), pp. 199-212

[13] Franke, R., *HQP: a solver for sparse nonlinear optimization*, Technische Universität Ilmenau (1997)

[14] Franke, R., *Omuses: a tool for the Optimization of multistage systems*, Technische Universität Ilmenau (1997)

[15] Griewank, A., Juedes, D. and Utke, J., *ADOL-C: A Package for the Automatic Differentiation of Algorithms written in C/C++* (1996)

[16] Goldstein H., *Classical Mechanics*, Addison-Wesley, Reading, MA (1980)

[17] Hildebrand, F. B., *Methods of Applied Mathematics*, Prentice-Hall, Englewood Cliffs, NJ (1965)

[18] Fletcher, R., *Practical Methods of Optimization*, John Wiley & Sons, (1987)

[19] Burdett, R. G., Skrinar, G. S., and Simon, S.R., *Comparison of mechanical work and metabolic energy consumption during normal gait*, Journal of Orthopaedic Research 1, (1983), pp. 63-72