A Kinematic Model Compiler for the Estimation of Articulated Motion from Video Sequences

Erik Sudderth

A senior honors project submitted in fulfillment of
the design requirement for the degree of
Bachelor of Science in Electrical Engineering
UNIVERSITY OF CALIFORNIA SAN DIEGO

April 1999

Contents

Chapter 1

Introduction and Motivation

Motion estimation is one of the primary problems in computer vision. The differences between successive images of a real scene provide powerful cues about the structure and behavior of the objects in that environment. Due to the ambiguities introduced by the projection of three-dimensional structure to two-dimensional images, however, accurate and reliable extraction of motion information is fraught with difficulties. Parallel extraction of both the structure of a scene and the motion of the objects present in that scene, without the introduction of many a priori assumptions, is in general infeasible.

One of the most attractive ways to address this problem is to provide a structural model of the objects of interest. Such a model parameterizes both the range of shapes that interesting objects may assume and the ways in which these shapes may change over time. Given a specific application domain, it is usually easy to specify such a model, at least at a heuristic level. For example, in order to track the motion of people through a room, it is natural to begin by describing the structure of human bodies. By providing cues as to the size and shape of the various limbs, and well as the behavior of the joints which hold the limbs together, the motion estimation problem can be made much more reasonable.

Eventually, such high-level domain knowledge about object characteristics must be translated into mathematical constraints to be applied in a motion estimation algorithm. In practice, making this transition can be quite difficult. If a motion estimation algorithm is to be robust and efficient, such modeling clues must be integrated into the heart of the system, not tacked on as a heuristic afterthought. This integration generally involves a great deal of

effort. Unfortunately, in many cases the modifications made tend to be domain specific, and cannot be easily translated to new application areas.

The model compiler described in this document attempts to resolve some of these difficulties. It provides a modeling language which facilitates the succint description of a class of models which, while not universal, can be applied to an extremely broad range of application domains. It then automatically translates these models into the software code necessary to apply a sophisticated motion estimation algorithm which takes full advantage of the model's expressiveness. The resulting software is integrated into a graphical interface which allows interactive experimentation with video data.

1.1 Modeling Assumptions

Although the compiler discussed in this paper has been designed to be as general as possible, by necessity some modeling assumptions are made. We assume that the objects of interest may be adequately described as a collection of bodies joined together by a system of joints. Such structures are called articulated models. Furthermore, we assume that the bodies themselves are rigid, meaning that their fundamental shape may not be distorted. Note, however, that it is often possible to approximate deformable bodies by groups of appropriately connected rigid bodies [6]. No fundamental restrictions on the three-dimensional shape of a given body, or the nature of the joints connecting the different bodies, are made.

The model specification language, which is described in §3.1, could potentially provide input to a wide range of motion estimation, simulation, and synthesis algorithms. The compiler that interprets the modeling language, however, currently generates the code necessary to run an articulated motion estimation algorithm known as expection constrained-maximization, or ECM. An early version of this algorithm was developed and applied by Edward Hunter [2] to the estimation of human upper-body motion. The same algorithm was adapted by Patrick Kelly to lip tracking applications [5, 6]. This early work demonstrated the feasibility of ECM in two diverse application domains. More recently, Hunter applied a

new version of the ECM algorithm, which incorporates significant advances in several areas, to the estimation of human bipedal motion [3]. It is this latest version of the ECM that is discussed in the remainder of this paper and utilized by the model compiler.

1.2 Advantages of a Model Compiler

It is natural to ask whether a model compiler is truly necessary. Research in motion estimation has been pursued for decades, but to my knowledge the model compiler concept is relatively novel. For a variety of reasons, however, recent advances in computing power have made the need for such compilers more and more pressing. In the first place, complex algorithms that were once computationally infeasible are now quite reasonable. However, the implementation of the details of these algorithms has not been matched by a corresponding decrease in complexity. In addition, technological improvements have led to a decreased reliance on synthetic data and more widespread utilization of digitized video. This is a positive step in the direction of building useful vision systems, but it brings with it further implementation burdens for the vision researcher.

The ECM algorithm provides a useful concrete example of these trends. A critical step in the ECM procedure is the application of kinematic constraints in order to insure that the different bodies remain in positions consistent with the relevant joints. The constraints are applied using a principle known as virtual work, as described in §2.3. Virtual work requires that the symbolic Jacobian of the constraint system be available. The derivation of the Jacobian by taking symbolic partial derivatives is straightforward and quite mechanical. However, as realistic models may quite reasonably have thousands of entries in their Jacobian matrices, this is an enormously tedious process to perform by hand. Once the Jacobian is derived, computer code must then be created which calculates the Jacobian using an interface consistent with the rest of the estimation system. Manual entry of such code is a notorious source of bugs.

Even once the purely computational portions of an estimation algorithm have been adapted to a new model, other areas of the system must also be considered. Three-dimensional motion estimation from video sequences is virtually impossible without the availability of a graphical interface which allows interactive experimentation with the estimation process. However, graphical interfaces bring with them large blocks of code which must be modified to interface with new model configurations [3]. For example, the interface must have knowledge of the structure of the model so joint angles may be manipulated and the results visually displayed. Painful experience has shown that adapting an entire software framework to a new model generally involves several weeks of laborious, tedious modifications.

All of this effort might be reasonable if new models were only rarely needed. However, when researching unexplored modeling techniques, the correct modeling choices are almost never made the first time. Multiple revisions are virtually always required. Ideally, the researcher would like to be able to think of a new model, write down a specification, click a button, and be able to apply the estimation algorithm to that new structure. The model compiler described in this paper provides exactly that flexibility. Furthermore, the internal code structure of a model compiler provides a disciplined framework for introducing improvements to a motion estimation algorithm. This helps insure that modifications will be generalizable, sensible extensions rather than domain-specific hacks.

In many ways, the model compiler draws its inspiration from the same concerns that prompted the introduction of object-oriented programming languages [13]. Software objects, if designed properly, may be directly applied in application areas which were never anticipated by the original designer. Similarly, the model compiler allows research in one estimation domain to be easily and flexibly applied to new estimation problems as they arise.

Chapter 2

Estimation of Articulated Motion using the ECM Algorithm

2.1 ECM Algorithm Overview

In this section, I discuss the general structure of the expectation constrained-maximization (ECM) algorithm for the estimation of articulated motion from video sequences. This algorithm was originally designed and implemented by Edward Hunter [3]. Rather than attempting to duplicate the detailed discussion contained in Hunter's thesis, I will instead provide a high-level overview, giving details only as they are relevant to the compiler design.

2.1.1 Dual-Model Representation

Before motion estimation may occur, it is necessary to specify some sort of analytic model of the object of interest. For the purposes of the ECM algorithm, we assume that objects may be represented as a system of rigid bodies whose spatial configurations must satisfy pre-specified constraints. For example, a robot manipulator arm could be modeled as a set of links constrained by the joints which hold them together. However, as the goal of the ECM algorithm is estimation rather than simulation, it is desirable to modify the model so that the inevitable uncertainties in each body's position and orientation may be explicitly accounted for.

The ECM algorithm achieves these goals through the integration of two parallel models.

The first model describes the rigid bodies which make up the body system. Bodies are represented through the superposition of two concentric three-dimensional ellipsoids. The inner ellipsoid represents the true structure of the rigid body. The outer ellipsoid, which must lie entirely outside the inner, represents the degree of uncertainty in the body's estimated position. The relative certainty that the body occupies a given position is taken to be constant within the inner ellipsoid and zero outside the outer ellipsoid, with linear decline in the intermediate area. This profile much more closely matches the silhouettes produced by segmentation algorithms than traditional Gaussian densities. Together, these two ellipsoids completely specify a "generalized density" which is ideally suited to the ECM estimation procedure.

The second model describes the kinematic relationships between the different rigid bodies contained in the body system. In general, these constraints could be almost anything that can be written in the form of an equation involving the inertial coordinates of the various bodies. Currently, however, all constraints are modeled as one of two primitive joint types. Fixed joints have no degrees of freedom, while revolute joints have a single rotational degree of freedom. For a variety of reasons, these two joint types form a very sensible set of basic constraints. They allow the specification of a wide variety of model types, as will be demonstrated in §4.1. In addition, more complex joints such as spherical and twist-and-bend joints may be formed by the concatenation of these basic types. Finally, single degree-of-freedom joints are generally more robust to noise in the estimation procedure than complex constraints which involve many degrees of freedom.

A given joint is described by the position and orientation of the joint axis in the local coordinate frames of the two rigid bodies of interest. The locations of these joint axes provide the link between the ellipsoid structure of the individual bodies and the kinematic relationships between bodies. Because the joint axes and body shapes are specified separately, they may be independently adjusted to best match the character of the input data. This type of modeling flexibility is very important when models must be matched to often

noisy real-world measurements.

2.1.2 ECM Iterative Estimation Procedure

The two parallel models discussed in the previous section are integrated in the ECM algorithm. ECM can be viewed as a generalization of the EM algorithm commonly used in the numerical solution of maximum-likelihood estimation problems [4]. The algorithm is iterative in nature, eventually converging to at least a local maximum. However, as the changes in object position between high-speed video frames are generally quite small, this local maximum will frequently also be the global maximum.

The input to the ECM algorithm is not raw video data, but instead a figure-ground segmentation which labels each pixel as belonging either to the rigid body system or to the background. For most application areas, such silhouettes may be produced using an adaptive motion segmentation algorithm like that described in [14]. Where this is not appropriate, a color segmentation algorithm may be employed [6]. Silhouettes were chosen as the input for the ECM algorithm because they are among the easiest features to reliably extract from noisy video sequences. Specifically, silhouettes do not require the use of special markers or body suits as is common in motion capture systems.

The first step in the estimation procedure is the expectation step, or E-step. Here, the three-dimensional generalized densities for each rigid body are first projected onto the image plane. The resulting two-dimensional distributions are then used to assign the foreground pixels produced by the segmentation algorithm to one or more of the rigid bodies. This effectively produces a "soft labeling" of the foreground pixels, proportionally distributing contested pixels between the different overlapping densities. Occlusion reasoning, based on the three-dimensional kinematic structure, is applied at this point to account for cases where one component of the body system occludes another.

The E-step is followed by a maximization step, or M-step. In this step, the parameters of the various distributions are modified in order to maximize their alignment with the

pixels assigned to them in the E-step. These maximizations are done independently for each component, which is what allows the EM to handle otherwise intractable computational problems. Traditional EM techniques, which use Gaussian densities in the E-step, apply a direct analytic expression to perform the maximization. However, due to their relatively narrow peaks and long tails, Gaussian densities are undesireable for modeling the structure of rigid bodies. Therefore, non-Gaussian densities were used in the rigid body specifications, which in turn requires that the M-step be search based. The search occurs over the six inertial parameters which specify the body's inertial position and orientation.

Traditional EM algorithms would follow the M-step by another E-step, iterating until convergence is achieved. However, this would in general pull the bodies into configurations that are no longer kinematically consistent. Therefore, the ECM algorithm constrains the results of the M-step before proceeding to the next E-step. This constraint step, or static update, attempts to find the valid kinematic configuration which is most consistent with the kinematically invalid M-step results. Consistency is measured using a principle drawn from multibody dynamics known as virtual work. Heuristically, the static update works by connecting virtual springs between the M-step estimates and the prior body configuration. The body is then pulled by these springs into the kinematically valid configuration that minimizes the total spring energy, as shown in figure 2.3. The kinematically correct body locations produced by the static update are then used as the input for the next E-step.

The remainder of this document will focus almost entirely on the static update, as this is the algorithm component for which the model compiler was primarily created. With a few minor exceptions, the E and M steps may be performed on a component by component basis. This means that a single handwritten routine may perform these operations without the need for major modifications to match the interbody relationships in a given model. In contrast, by its nature the static update must be specially modified to match the global structure of the model. Sections §2.2 and §2.3 will discuss the theory behind the static update in much greater depth, while chapter 3 will show how the creation of code to implement this theory

may be automated using a model compiler.

2.2 Kinematic Background

In this section we develop the kinematic principles that allow us to represent the models employed in the ECM algorithm. We assume that the physical mechanism of interest may be adequately modeled by a system of rigid bodies whose relative configurations must satisfy a set of constraints. These constraints will be specified in the form of primitive joints which connect the bodies. Special attention will be paid to the unique kinematic problems that arise from the fact that both inertial and relative parameterizations are required at different points in the estimation procedure.

2.2.1 Rigid Body Representation

A rigid body is a three-dimensional structure in which the relative position of all internal points is fixed; in other words, it is not deformable. It can be shown that a total of six independent parameters are both necessary and sufficient to specify the configuration of a rigid body in space: three to fix the position, and three to determine the orientation. However, in order to avoid singularities in the representation of orientation, four rotational coordinates will be used, only three of which are independent.

We denote the location of the center of mass of the i^{th} rigid body, relative to the inertial, or global, coordinate frame, by the three-element vector $\boldsymbol{\mu}_{0,i}$. Consider a point whose position in the coordinate system of body i is given by $\mathbf{u}_i$. We may then determine the position of this point in the inertial coordinate frame $\mathbf{u}_0$ according to

$$\mathbf{u}_0 = \boldsymbol{\mu}_{0,i} + \mathbf{R}_{0,i}\mathbf{u}_i \tag{2.1}$$

Here, $\mathbf{R}_{0,i}$ is a rotation matrix which maps points from the i^{th} body's coordinate frame to

the inertial coordinate frame. From this, we see that the specification of the orientation of a rigid body amounts to the parameterization of that body's rotation matrix.

This rotation matrix may be parameterized using a wide variety of different variables [11]. However, depending on the application under investigation, some parameterizations may be much more convenient than others. The Rodriquez formula states that any rotation may be represented by an axis of rotation, $\mathbf{v} = [\ v_1 \quad v_2 \quad v_3\ ]^T$, and an angle of rotation θ about that axis. Note that the axis must be specified by a unit vector, so only three of these four parameters are independent. The rotation matrix corresponding to a given angle and axis choice is

$$\mathbf{R}_{0,i} = \mathbf{I} + \tilde{\mathbf{v}} \sin\theta + 2(\tilde{\mathbf{v}})^2 \sin^2 \frac{\theta}{2} \tag{2.2}$$

$$\tilde{\mathbf{v}} = \begin{pmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{pmatrix} \tag{2.3}$$

Here, $\tilde{\mathbf{v}}$ is the matrix representation of a cross product. Applying half-angle identities and simplifying, the rotation matrix is

$$\mathbf{R}_{0,i} = \begin{pmatrix} v_1^2(1-cos\theta)+cos\theta & v_2v_1(1-cos\theta)-v_3sin\theta & v_3v_1(1-cos\theta)+v_2sin\theta \\ v_1v_2(1-cos\theta)+v_3sin\theta & v_2^2(1-cos\theta)+cos\theta & v_3v_2(1-cos\theta)-v_1sin\theta \\ v_1v_3(1-cos\theta)-v_2sin\theta & v_2v_3(1-cos\theta)+v_1sin\theta & v_3^2(1-cos\theta)+cos\theta \end{pmatrix} \tag{2.4}$$

It is often convenient to use an alternate matrix parameterization known as Euler parameters. Euler parameters are given by a simple transformation of the angle-axis rotation representation:

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 & \theta_3 \end{bmatrix}^T = \begin{bmatrix} \cos\frac{\theta}{2} & v_1 \sin\frac{\theta}{2} & v_2 \sin\frac{\theta}{2} & v_3 \sin\frac{\theta}{2} \end{bmatrix}^T \tag{2.5}$$

Note that there are in reality only three independent Euler parameters, as they must satisfy

$$\boldsymbol{\theta}^T\boldsymbol{\theta} = \theta_0^2 + \theta_1^2 + \theta_2^2 + \theta_3^2 = 1 \tag{2.6}$$

We may write the rotation matrix explicitly in terms of the Euler parameters as

$$\mathbf{R}_{0,i} = \begin{pmatrix} 1-2\theta_2^2-2\theta_3^2 & 2(\theta_1\theta_2-\theta_0\theta_3) & 2(\theta_1\theta_3+\theta_0\theta_2) \\ 2(\theta_1\theta_2+\theta_0\theta_3) & 1-2\theta_1^2-2\theta_3^2 & 2(\theta_2\theta_3-\theta_0\theta_1) \\ 2(\theta_1\theta_3-\theta_0\theta_2) & 2(\theta_2\theta_3+\theta_0\theta_1) & 1-2\theta_1^2-2\theta_2^2 \end{pmatrix} \tag{2.7}$$

These Euler parameters, together with the location of the center of mass $\boldsymbol{\mu}_{0,i}$ in inertial space, completely define the configuration of a rigid body. Denoting the Euler parameters of the i^{th} body as $\boldsymbol{\theta}_i$, we collect these parameters in a coordinate vector $\mathbf{q}_i$ for use in later sections:

$$\mathbf{q}_i = \begin{bmatrix} \boldsymbol{\mu}_{0,i}^T & \boldsymbol{\theta}_i^T \end{bmatrix}^T = \begin{bmatrix} \mu_{0,i,1} & \mu_{0,i,2} & \mu_{0,i,3} & \theta_{i,0} & \theta_{i,1} & \theta_{i,2} & \theta_{i,3} \end{bmatrix}^T \tag{2.8}$$

We are now able to generate a rotation matrix given either an angle-axis or Euler parameter specification. However, in the following sections, it will also be necessary to extract the Euler parameters given the nine entries of the rotation matrix. By examining the three eigenvector/eigenvalue pairs of a rotation matrix, explicit formulas for the angle and axis of

rotation may be derived [3]. Assuming r_{ij} represents the $(i,j)^{\text{th}}$ entry of $\mathbf{R}_{0,i}$ we have

$$\begin{aligned}
\mathbf{x} &= \begin{cases} \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T & \text{if } \mathbf{x} \perp \mathbf{e}_2 \text{ and } \mathbf{x} \perp \mathbf{e}_3 \\ \begin{pmatrix} r_{12} & 1-r_{11} & 0 \end{pmatrix}^T & \text{if } \mathbf{x} \perp \mathbf{e}_3 \\ \begin{pmatrix} r_{13} - r_{13}r_{22} + r_{12}r_{23} \\ r_{13}r_{21} + r_{23} - r_{11}r_{23} \\ 1 - r_{11} - r_{12}r_{21} - r_{22} + r_{11}r_{22} \end{pmatrix} & \text{otherwise} \end{cases} \\
\mathbf{v} &= \frac{\mathbf{x}}{||\mathbf{x}||} \\
\theta &= \cos^{-1}\left(\frac{\text{Trace}(\mathbf{R}_{0,i}) - 1}{2}\right)
\end{aligned} \tag{2.9}$$

Using equation 2.9, we may compute the Euler parameters, up to a sign change, as

$$\boldsymbol{\theta} = \begin{pmatrix} \frac{\sqrt{1+\text{Trace}(\mathbf{R}_{0,i})}}{2} \\ \frac{v_1\sqrt{3-\text{Trace}(\mathbf{R}_{0,i})}}{2} \\ \frac{v_2\sqrt{3-\text{Trace}(\mathbf{R}_{0,i})}}{2} \\ \frac{v_3\sqrt{3-\text{Trace}(\mathbf{R}_{0,i})}}{2} \end{pmatrix} \tag{2.10}$$

Choosing the correct sign for the Euler parameters is not simple. Euler parameters are not unique because switching the sign of all four parameters produces an identical rotation matrix. Physically, this is equivalent to flipping the axis of rotation and rotating about the opposite angle. For some applications, any parameter set which produces the correct rotation matrix might be sufficient. However, in order for the numerical estimation procedures discussed later to be stable, we must have a convention which makes the Euler representation for a given rotation matrix unique.

First, we assume that θ is between 0 and 2π, putting $\theta/2$ between 0 and π. This means that $\sin(\theta/2)$ will always be positive. Also, we may uniquely determine the rotation angle from $\theta = 2\cos^{-1}\theta_0$. Furthermore, we assume that $v_1 > 0$, fixing the orientation of the axis

of rotation. If $v_1 = 0$, we instead assume $v_2 > 0$. With these conventions, equation 2.10 may be used to uniquely determine the relevant Euler parameters given a rotation matrix.

2.2.2 Relative Orientations and Forward Kinematics

As part of the application of kinematic constraints in the ECM algorithm, it will be necessary to compute all of the dependent coordinates in the rigid body given only a minimal subset of independent coordinates. Assuming there are no closed loops in the kinematic configuration, this may be done by first determining the configuration of the lowest body in the chain. Then, the configuration of each upper body may be determined by walking down the different kinematic chains. At each link in the chain, the equations which describe these forward kinematics depend on the joint type connecting it to its lower body.

The following sections derive the constraint equations, as well as the required forward kinematic equations, for the various joint types currently supported by the model compiler. These derivations make use of the concept of relative rotation matrices. Given the inertial orientation matrices $\mathbf{R}_{0,i}$ and $\mathbf{R}_{0,j}$ of the lower and upper bodies, respectively, the relative rotation matrix $\mathbf{R}_{i,j}$ satisfies

$$\begin{aligned} \mathbf{R}_{0,j} &= \mathbf{R}_{0,i}\mathbf{R}_{i,j} \\ \mathbf{R}_{i,j} &= \mathbf{R}_{0,i}^T\mathbf{R}_{0,j} \end{aligned} \tag{2.11}$$

However, we will usually not be able to compute $\mathbf{R}_{i,j}$ according to this formula as $\mathbf{R}_{0,j}$ will be unknown. Instead, we will in general assume we know $\mathbf{q}_i$ as given by equation 2.8 for the lower body, as well as either relative joint information or a portion of the inertial coordinates $\mathbf{q}_j$ of the upper body. We must then derive $\mathbf{R}_{i,j}$ from geometric principles, and use equation 2.11 to compute $\mathbf{R}_{0,j}$.

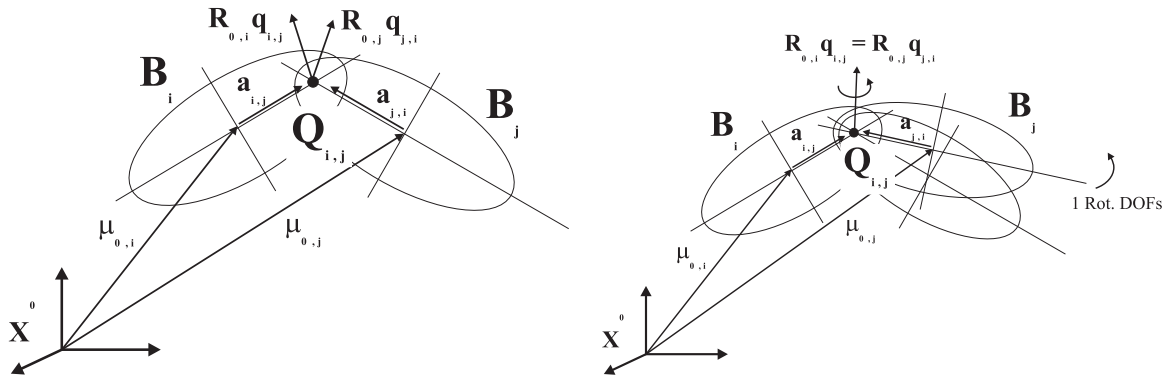


Figure 2.1: Model of the construction of a revolute joint through the alignment of the joint axes in two rigid bodies. Fixed joints are constructed identically, except the rotation angle is always zero [3].

2.2.3 Fixed Joints

In a fixed joint, the two bodies are rigidly connected. There are no degrees of freedom, so knowledge of the lower body's configuration completely determines the configuration of the upper body. A fixed joint is defined by specifying the position and orientation of the joint axis in each coordinate frame, as shown in figure 2.1. We denote the joint position vectors in the local coordinate frames of the lower and upper bodies, respectively, by $\mathbf{a}_{i,j}$ and $\mathbf{a}_{j,i}$. The joint orientations are defined by unit vectors $\mathbf{q}_{i,j}$ and $\mathbf{q}_{j,i}$.

Constraint Equations

Given the joint orientation in both coordinate frames, we may directly compute the relative rotation matrix by employing the angle-axis representation given in equation 2.4. The axis of rotation is perpendicular to the unit vectors which define the joint orientation, and may be found using the cross product. Similarly, the dot product may be used to determine the angle of rotation as follows:

$$\begin{aligned} \mathbf{v} &= \frac{\mathbf{q}_{j,i} \times \mathbf{q}_{i,j}}{||\mathbf{q}_{j,i} \times \mathbf{q}_{i,j}||} \\ \theta &= \cos^{-1}\left(\mathbf{q}_{i,j}^T \mathbf{q}_{j,i}\right) \end{aligned} \tag{2.12}$$

Therefore, we may compute $\mathbf{R}_{i,j}$ for a fixed joint, which constrains the three rotational degrees of freedom. The three translational degrees of freedom may be constrained by noting that the inertial position of the joint axis as computed from each body's coordinate frame must be identical. This provides the following set of constraint equations:

$$\begin{aligned} \mathbf{R}_{0,j} &= \mathbf{R}_{0,i}\mathbf{R}_{i,j} \\ \boldsymbol{\mu}_{0,i} + \mathbf{R}_{0,i}\mathbf{a}_{i,j} &= \boldsymbol{\mu}_{0,j} + \mathbf{R}_{0,j}\mathbf{a}_{j,i} \end{aligned} \tag{2.13}$$

Forward Kinematics

Given the constraints in equation 2.13, the forward kinematics are straightforward. We may directly determine $\mathbf{R}_{0,j}$ using the precomputed relative rotation matrix, and then apply equation 2.10 to compute the Euler parameters $\boldsymbol{\theta}_j$. Given this, we may find the position of the lower body using

$$\boldsymbol{\mu}_{0,j} = \boldsymbol{\mu}_{0,i} + \mathbf{R}_{0,i}\mathbf{a}_{i,j} - \mathbf{R}_{0,j}\mathbf{a}_{j,i} \tag{2.14}$$

2.2.4 Revolute Joints

In a revolute joint, the two bodies have a single rotational degree of freedom. As with a fixed joint, a revolute joint is defined by specifying the position and orientation of the joint axis in each coordinate frame, as shown in figure 2.1. However, unlike fixed joints, revolute joints also allow circular rotation around this axis. We denote the joint position vectors in the local coordinate frames of the lower and upper bodies, respectively, by $\mathbf{a}_{i,j}$ and $\mathbf{a}_{j,i}$. The joint orientations are defined by unit vectors $\mathbf{q}_{i,j}$ and $\mathbf{q}_{j,i}$.

Constraint Equations

A total of five independent constraints are necessary to limit a revolute joint to a single rotational degree of freedom. Position may be constrained in the same manner as it was for fixed joints in equation 2.13. Orientation may be constrained by requiring that the two joint

vectors have the same inertial orientation. This leads to the following constraint system:

$$\begin{aligned} \boldsymbol{\mu}_{0,i} + \mathbf{R}_{0,i}\mathbf{a}_{i,j} &= \boldsymbol{\mu}_{0,j} + \mathbf{R}_{0,j}\mathbf{a}_{j,i} \\ \mathbf{R}_{0,i}\mathbf{q}_{i,j} &= \mathbf{R}_{0,j}\mathbf{q}_{j,i} \end{aligned} \tag{2.15}$$

This is actually six equations, but only five will be linearly independent due to the fact that the orientations are defined by unit vectors. It is tempting to arbitrarily pick two of the three orientation constraints. However, for certain combinations of joint orientations $\mathbf{q}$ and inertial orientations $\boldsymbol{\theta}$, one of the constraints may turn out to be trivial. Therefore, in §2.3.2 a numerical procedure is proposed for dynamically selecting the independent constraint equations at run time.

Forward Kinematics given Relative Joint Angle θ

When manipulating models from a graphical interface, it is far more intuitive to modify the relative joint angle θ than to deal with inertial parameters. In addition, many application domains that might make use of the ECM algorithm are very interested in relative parameterizations. For these reasons, it is important to be able to successfully compute the forward kinematics given a new joint angle.

We may view the relative rotation matrix $\mathbf{R}_{i,j}$ corresponding to a revolute joint in either of two equivalent ways. First, we may see it as a rotation θ about the lower body joint axis $\mathbf{q}_{i,j}$ followed by a fixed rotation between the joint orientations $\mathbf{q}_{i,j}$ and $\mathbf{q}_{j,i}$. Alternatively, the it can be viewed as a fixed rotation between the joint orientations $\mathbf{q}_{i,j}$ and $\mathbf{q}_{j,i}$ followed by a rotation θ about the upper body joint axis $\mathbf{q}_{j,i}$. Symbolically we have

$$\mathbf{R}_{i,j} = \mathbf{R}_{q_{i,j},\theta}\mathbf{R}_{q_{i,j},q_{j,i}} = \mathbf{R}_{q_{i,j},q_{j,i}}\mathbf{R}_{q_{j,i},\theta} \tag{2.16}$$

We already know how to compute both components of this product. $\mathbf{R}_{q_{i,j},q_{j,i}}$ may be computed using equation 2.12, and either $\mathbf{R}_{q_{i,j},\theta}$ or $\mathbf{R}_{q_{j,i},\theta}$ may be computed using equation

2.4. Once the relative rotation matrix is computed, the rest of the forward kinematics are identical to those given in equation 2.14 for the fixed joint.

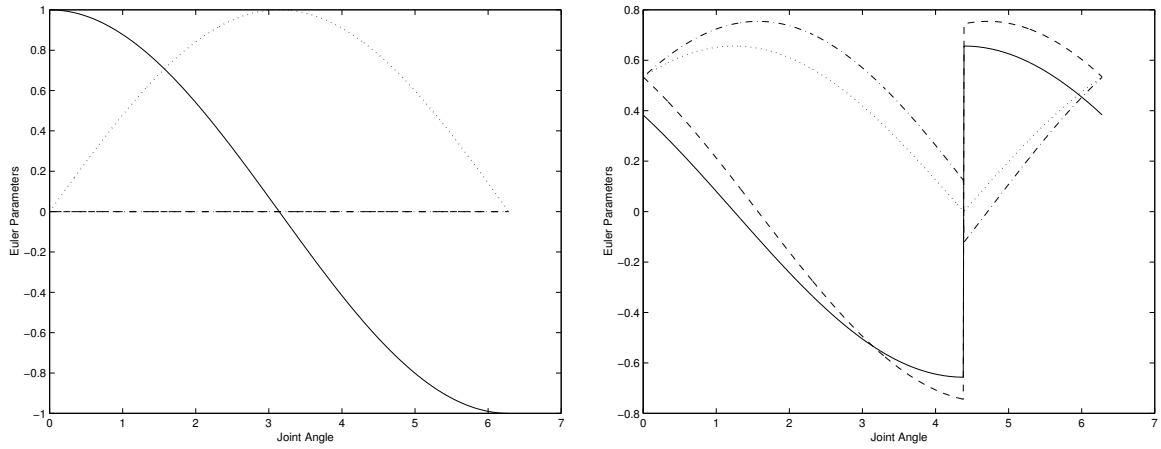
Forward Kinematics given Inertial Euler Parameter $\theta_{j,0}$

As discussed above, relative joint angles are the most natural way to specify the configuration of a revolute joint. However, the constraint equations which define the structure of a revolute joint, as given in equation 2.15, cannot be formulated in terms of the relative joint angle. Conceptually, this occurs because it is not possible to define a joint in terms of a parameter which depends on that joint's validity. Therefore, the independent coordinate used in static updates, as described in §2.3, must be one of the inertial parameters. Since revolute joints have a rotational degree of freedom, it is easier to choose one of the Euler parameters as the independent coordinate. θ_0 is chosen because it has no dependence on the axis of rotation, and it therefore avoids singularity problems which arise when one of the components of the rotation axis is zero.

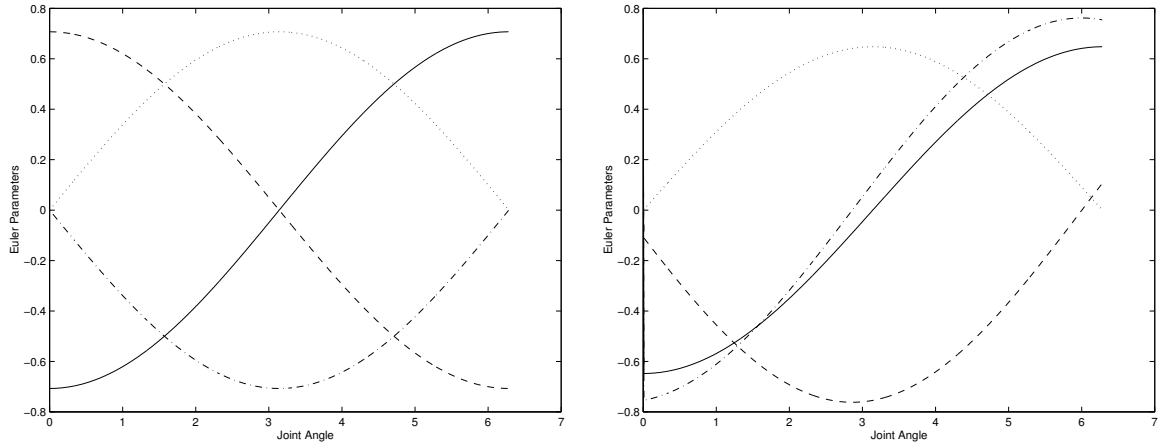
Numerous attempts have been made to find an analytical solution to equation 2.15 assuming that $\theta_{j,0}$ is known. To date, however, no closed-form solution has been found. Attempts to find a symbolic solution using Mathematica's symbolic manipulation capabilities produced formulas with literally millions of terms which were numerically unstable.

Instead, we have implemented a numeric solution in the form of a one-dimensional root-finding problem. For a given joint angle $\hat{\theta}$, we may compute the relative rotation matrix according to equation 2.16. $\mathbf{R}_{i,j}$ may be used to compute $\mathbf{R}_{0,j}$, from which we may determine a set of Euler parameters $\hat{\boldsymbol{\theta}}$ according to equation 2.10. We then compare the computed Euler parameter $\hat{\theta}_{j,0}$ to the independent coordinate $\theta_{j,0}$, and use the discrepancy to improve our estimate of the joint angle $\hat{\theta}$.

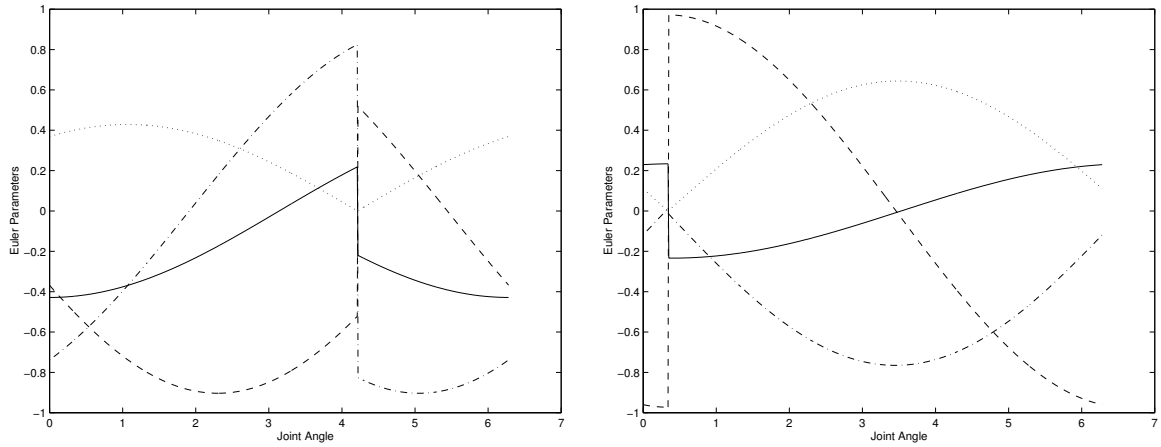
The numeric solution just described has proven to be much more stable than any other solution techniques yet determined. However, for some combinations of joint orientations and inertial coordinates, the functional relationship between the Euler parameters and the



(a) $\mathbf{o}_i = [\ 1 \quad 0 \quad 0\ ]^T$, $\mathbf{o}_j = [\ 1 \quad 0 \quad 0\ ]^T$



(b) $\mathbf{o}_i = [\ 0 \quad 1 \quad 0\ ]^T$, $\mathbf{o}_j = [\ -1 \quad 0 \quad 0\ ]^T$



(c) $\mathbf{o}_i = [\ 1 \quad 0 \quad 1\ ]^T$, $\mathbf{o}_j = [\ -2 \quad -1 \quad 0\ ]^T$

Figure 2.2: Upper body inertial Euler parameters $\theta_0 - \theta_3$ (θ_0 drawn with solid line) as a function of revolute joint angle θ for various joint axis configurations. The left and right columns show two different lower body inertial configurations.

joint angle θ has ambiguities that must be considered by any proposed forward kinematics solution. Figure 2.2 shows plots of the four Euler parameters $\theta_0 - \theta_3$ of the upper body versus joint angle θ. In each plot, θ_0, the Euler parameter used in the forward kinematics solution, is plotted with a solid line.

Graphically, the forward kinematics problem is to find the inverse of these functions so that we may compute θ given θ_0. For most of the plots, this is not a problem as the mapping is one-to-one. There are discontinuities caused by enforcing our convention that the sign of θ_1 always be positive (see §2.2.1), but these may be accounted for. However, in the left-hand plot in figure 2.2(c), the mapping is not one-to-one. For $\theta_0 \approx -0.3$, there are two possible joint angles θ. Such ambiguities do not arise frequently and have as of yet not caused any problems. However, if the estimation procedure is to be stable in all orientations this must be considered. The most sensible means of compensating for this problem is an open research question.

2.3 Static Update

Section §2.2 provides a catalog of useful kinematic transformations. On their own, these formulas could be used to form the basis for a kinematic simulation framework. However, we are interested not in simulation but in the much more challenging problem of motion estimation. This section shows how kinematic constraints may be smoothly integrated into an estimation framework using the principle of virtual work [3, 11].

2.3.1 The Principle of Virtual Work

Consider a rigid body located in inertial space. The body is parameterized by a vector of seven inertial coordinates $\mathbf{q}_i$ as described in equation 2.8. The body is acted on by a system

of linear and torsional forces, which can be resolved into a single resultant force vector

$$\mathbf{F}_i = \begin{bmatrix} F_{i,\mu_1} & F_{i,\mu_2} & F_{i,\mu_3} & F_{i,\theta_0} & F_{i,\theta_1} & F_{i,\theta_2} & F_{i,\theta_3} \end{bmatrix}^T \tag{2.17}$$

This vector gives the net force acting on each body coordinate. If the body is in static equilibrium, we may write $\mathbf{F}_i = 0$. Therefore, for any arbitrarily small displacement of the body's coordinates from static equilibrium $\delta\mathbf{q}_i$, we have

$$\mathbf{F}_i^T \delta\mathbf{q}_i = 0 \tag{2.18}$$

Now consider a system of n rigid bodies in static equilibrium. From equation 2.18, we may directly write

$$\delta W \triangleq \sum_{i=1}^{n} \mathbf{F}_i^T \delta\mathbf{q}_i = 0 \tag{2.19}$$

Here, δW is defined as the virtual work of all the forces acting on the system. Decomposing $\mathbf{F}_i$ into $\mathbf{F}_{c,i}$ and $\mathbf{F}_{e,i}$, the forces due to internal constraints and external stimuli, we may write

$$\delta W = \delta W_c + \delta W_e = \sum_{i=1}^{n} \mathbf{F}_{c,i}^T \delta\mathbf{q}_i + \sum_{i=1}^{n} \mathbf{F}_{e,i}^T \delta\mathbf{q}_i = 0 \tag{2.20}$$

Assuming workless constraints, we let $\mathbf{F}_{c,i} = 0$. This is equivalent to assuming that the joints holding the bodies together are frictionless. The virtual work equations then become

$$\begin{aligned} \delta W &= \delta W_e = \sum_{i=1}^{n} \mathbf{F}_{e,i}^T \delta\mathbf{q}_i = \mathbf{Q}^T \delta\mathbf{q} = 0 \\ \mathbf{Q} &\triangleq \begin{bmatrix} \mathbf{F}_{e,1}^T & \mathbf{F}_{e,2}^T & \cdots & \mathbf{F}_{e,n}^T \end{bmatrix}^T \\ \delta\mathbf{q} &\triangleq \begin{bmatrix} \delta\mathbf{q}_1^T & \delta\mathbf{q}_2^T & \cdots & \delta\mathbf{q}_n^T \end{bmatrix}^T \end{aligned} \tag{2.21}$$

Here, $\delta\mathbf{q}_i$ is a vector of virtual displacements in the system coordinates, and $\mathbf{Q}$ is a vector of generalized forces acting on these system coordinates. Equation 2.21 provides a condition

for the determination of the position of static equilibrium given a set of externally applied forces. Note, however, that this equation does *not* imply that $\mathbf{Q} = 0$, because the elements of the virtual displacement vector will in general not be linearly independent due to the presence of constraints.

We may resolve this ambiguity using a technique known as generalized coordinate partitioning. Suppose that the body system is constrained by $n_c \leq 7n$ constraint equations. Note that n of these constraint equations will arise simply by applying equation 2.6 to the Euler parameters of each body. Assuming that the constraint equations are holonomic and scleronomic, or time-independent, we may write the equations as

$$\mathbf{C}(\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n) = \mathbf{C}(\mathbf{q}) = \begin{bmatrix} C_1(\mathbf{q}) & C_2(\mathbf{q}) & \cdots & C_{n_c}(\mathbf{q}) \end{bmatrix}^T = 0 \tag{2.22}$$

Taking a Taylor series expansion of this constraint system about a virtual displacement of the system coordinates $\delta\mathbf{q}$, we have

$$\begin{aligned} \mathbf{C}_q \delta\mathbf{q} &= 0 \\ \mathbf{C}_q &\triangleq \begin{pmatrix} \frac{\delta C_1}{\delta q_1} & \cdots & \frac{\delta C_1}{\delta q_{7n}} \\ \vdots & \ddots & \vdots \\ \frac{\delta C_{n_c}}{\delta q_1} & \cdots & \frac{\delta C_{n_c}}{\delta q_{7n}} \end{pmatrix} \end{aligned} \tag{2.23}$$

Now suppose we partition the coordinate vector $\mathbf{q}$ into a vector $\mathbf{q}_I$ of $7n - n_c$ independent coordinates and a vector $\mathbf{q}_D$ of n_c dependent coordinates:

$$\mathbf{q} = \begin{bmatrix} \mathbf{q}_I^T & \mathbf{q}_D^T \end{bmatrix}^T \tag{2.24}$$

Then, after appropriately permuting the columns of $\mathbf{C}_q$, we may partition the constraint

Jacobian to match the coordinate partitioning and obtain

$$\begin{aligned} \underset{n_c \times 7n}{\mathbf{C}_q} &= \begin{bmatrix} \underset{n_c \times (7n-n_c)}{\mathbf{C}_{q_i}} & \underset{n_c \times n_c}{\mathbf{C}_{q_d}} \end{bmatrix} \\ \mathbf{C}_q \delta \mathbf{q} &= \mathbf{C}_{q_i} \delta \mathbf{q}_I + \mathbf{C}_{q_d} \delta \mathbf{q}_D = 0 \end{aligned} \tag{2.25}$$

If the constraints are linearly independent and non-degenerate, $\mathbf{C}_{q_d}$ may be chosen to be full rank and therefore invertible. We may modify equation 2.25 to obtain

$$\begin{aligned} \mathbf{C}_{q_d} \delta \mathbf{q}_D &= -\mathbf{C}_{q_i} \delta \mathbf{q}_I \\ \delta \mathbf{q}_D &= -\mathbf{C}_{q_d}^{-1} \mathbf{C}_{q_i} \delta \mathbf{q}_I \end{aligned} \tag{2.26}$$

Finally, combining equations 2.24 and 2.26, we have

$$\delta \mathbf{q} = \begin{bmatrix} \delta \mathbf{q}_I \\ \delta \mathbf{q}_D \end{bmatrix} = \begin{bmatrix} \mathbf{I} \\ -\mathbf{C}_{q_d}^{-1} \mathbf{C}_{q_i} \end{bmatrix} \delta \mathbf{q}_I \triangleq \mathbf{B}_{di} \delta \mathbf{q}_I \tag{2.27}$$

The $\mathbf{B}_{di}$ matrix derived above allows us to write a virtual change in the system coordinates $\delta \mathbf{q}$ as a linear function of a virtual change in the independent system coordinates $\delta \mathbf{q}_I$. Combining equations 2.21 and 2.27, we may write

$$\begin{aligned} \delta W_e = \mathbf{Q}^T \mathbf{B}_{di} \delta \mathbf{q}_I &= 0 \\ \mathbf{B}_{di}^T \mathbf{Q} &= 0 \end{aligned} \tag{2.28}$$

The second line follows from the linear independence of the independent system coordinates.

2.3.2 Dynamic Constraint Selection using the SVD

In the derivation of equation 2.28, the assumption was made that the $\mathbf{C}_{q_d}$ matrix, as obtained from equation 2.25, could be chosen to be invertible. If all of the constraint equations are

independent and non-trivial, this condition will hold. However, in practice it can sometimes be difficult to determine constraint sets that are well behaved in all inertial orientations.

Consider the revolute joint discussed in §2.2.4, whose constraint system is given by equation 2.15. The first three constraints, which require that the joint positions be aligned in both coordinate frames, are independent in all orientations. However, the orientation constraints are more problematic. As the joint orientations are described by unit vectors, only two of the orientation constraints will be independent. Also, it is easy to show that for some combinations of inertial orientations and joint directions, one of the three equations may become degenerate (ie, it will reduce to $0 = 0$). The specific equation that becomes degenerate may change as the bodies are spun through space.

This situation poses problems for the construction of the $\mathbf{C}_q$ matrix. An arbitrary selection of two of the three orientation constraints may lead to a degenerate constraint system at some orientations, which will in turn make the $\mathbf{C}_{q_d}$ matrix non-invertible. One possible solution would be to monitor the values of the constraint equations, and swap in a different equation when one of the existing equations becomes degenerate. However, this would require complex logic that might become infeasible as the number of system joints becomes large. Also, it is not clear how switching the constraints at run-time would affect the numeric stability of the optimization algorithms which are used to solve the virtual work equations.

Instead, a more elegant and stable solution may be found using the singular value decomposition, or SVD [12]. We begin by augmenting the system of n_c constraint equations given in equation 2.22 with n_r additional redundant constraint equations. For example, we would add one additional constraint equation for each revolute joint in the system so that all three orientation equations are included. We may compute the Jacobian of this augmented constraint system and rewrite equation 2.25 as

$$\begin{aligned} \underset{(n_c+n_r)\times 7n}{\mathbf{C}_q} &= \begin{bmatrix} \underset{(n_c+n_r)\times(7n-n_c)}{\mathbf{C}_{q_i}} & \underset{(n_c+n_r)\times n_c}{\mathbf{C}_{q_d}} \end{bmatrix} \\ \mathbf{C}_q \delta\mathbf{q} &= \mathbf{C}_{q_i}\delta\mathbf{q}_I + \mathbf{C}_{q_d}\delta\mathbf{q}_D = 0 \end{aligned} \tag{2.29}$$

Assuming the redundant constraints are properly chosen to account for any physical singularities, we now have $r(\mathbf{C}_q) = r(\mathbf{C}_{q_d}) = n_c$ for *all* inertial orientations. We may decompose the augmented $\mathbf{C}_{q_d}$ matrix using the SVD as

$$\mathbf{C}_{q_d} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \begin{bmatrix} \mathbf{U}_r & \mathbf{U}_n \end{bmatrix} \begin{bmatrix} \overline{\mathbf{\Sigma}} \\ \mathbf{0} \end{bmatrix} \mathbf{V}^T = \mathbf{U}_r \overline{\mathbf{\Sigma}} \mathbf{V}^T \tag{2.30}$$

$$\mathbf{U} \in \mathbb{R}^{(n_c+n_r)\times(n_c+n_r)} \quad \mathbf{U}_r, \mathbf{\Sigma} \in \mathbb{R}^{(n_c+n_r)\times n_c} \quad \mathbf{V}, \overline{\mathbf{\Sigma}} \in \mathbb{R}^{n_c \times n_c} \quad \mathbf{U}_n \in \mathbb{R}^{n_r \times n_c}$$

From the properties of the singular value decomposition, we know that $\mathcal{R}(\mathbf{U}_r) = \mathcal{R}(\mathbf{C}_{q_d})$. The SVD provides a stable numerical procedure for determining an n_c–column orthonormal basis for $\mathbf{C}_{q_d}$. Using this result, we may compute the Moore-Penrose pseudoinverse of $\mathbf{C}_{q_d}$ as

$$\mathbf{C}_{q_d}^+ = \mathbf{V}\overline{\mathbf{\Sigma}}^{-1}\mathbf{U}_r^T \tag{2.31}$$

Combining equations 2.27 and 2.31, we have

$$\delta\mathbf{q} = \begin{bmatrix} \delta\mathbf{q}_I \\ \delta\mathbf{q}_D \end{bmatrix} = \begin{bmatrix} \mathbf{I} \\ -\mathbf{C}_{q_d}^+ \mathbf{C}_{q_i} \end{bmatrix} \delta\mathbf{q}_I \triangleq \mathbf{B}_{di}\delta\mathbf{q}_I \tag{2.32}$$

The pseudoinverse in equation 2.31 is formed from linear combinations of constraints which are all internally consistent. Therefore, heuristically equation 2.32 should produce a virtual displacement in the dependent coordinates which is equivalent to that produced by equation 2.27. The pseudoinverse simply allows us to automatically discard constraints which have become degenerate, or numerically close to degenerate. Although this equivalence has not been rigorously demonstrated, the positive results of the experiments in §4.1 provide experimental evidence which supports this approach.

2.3.3 Integration of Virtual Work with the ECM Algorithm

Equation 2.28 provides a readily computable condition for determining whether a system of kinematically constrained rigid bodies are in equilibrium with the externally applied forces. We may directly apply these results to transform the kinematically invalid body configurations generated by the M-step into the most consistent kinematically valid system configuration. The exact meaning of consistency is determined by the way in which the external forces composing the $\mathbf{Q}$ vector are defined.

After a given M-step, two sets of rigid bodies are present: one corresponding to the prior kinematically valid configuration, and the other corresponding to the kinematically invalid M-step results. The kinematically valid bodies are called primary bodies, and the M-step results are called intermediate bodies. We denote the coordinate vectors of the i^{th} primary and intermediate bodies as $\mathbf{q}_i$ and $\mathbf{q}'_i$, respectively. We may then compute the force on the i^{th} body, as specified in equation 2.17, as

$$\mathbf{F}_i[j] = k_{i,j}(\mathbf{q}'_i[j] - \mathbf{q}_i[j]) \quad j = 1, \ldots, 7 \tag{2.33}$$

Physically, equation 2.33 corresponds to attaching a set of linear and torsional springs between the primary and intermediate bodies. The virtual work procedure acts to pull the primary bodies into the kinematically valid configuration at which the spring forces exactly balance, as shown in figure 2.3. This is taken to be the configuration most consistent with the data. Note that in general the spring constant k may be different for every coordinate of every body. This is an important parameter which may be used by the modeler to compensate for noise or ambiguities in the input data.

Given the spring force definition in equation 2.33, we have all the necessary information to perform the static update. Using the joint constraint formulas derived in §2.2, we may write a system of constraint equations $\mathbf{C}(\mathbf{q})$ corresponding to equation 2.22. We augment these equations with redundant constraints as described in §2.3.2 as necessary to avoid

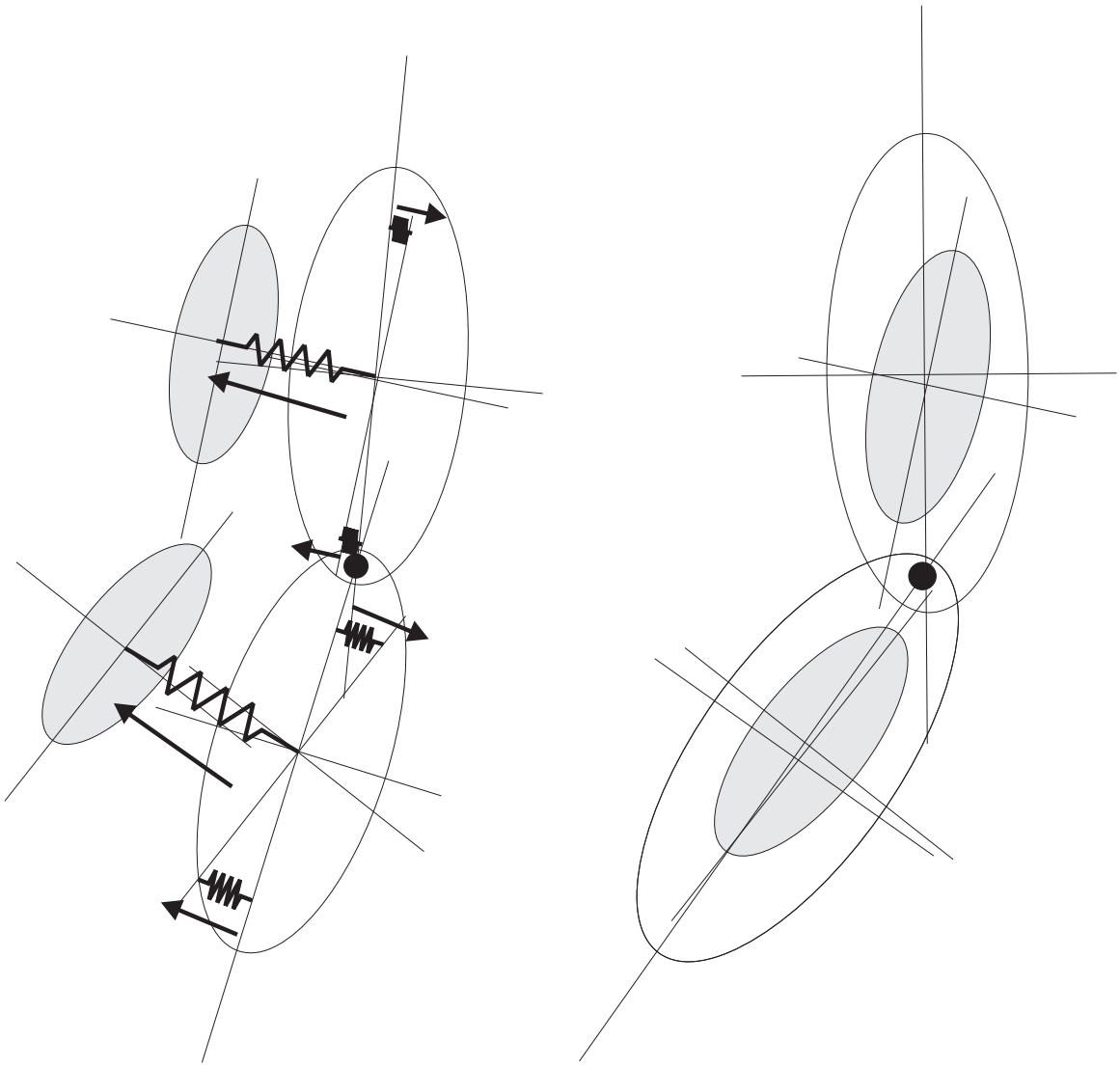


Figure 2.3: Physical interpretation of the principle of virtual work as using springs to pull the kinematically consistent primary bodies into the closest possible alignment with the kinematically inconsistent intermediate bodies. The intermediate bodies are drawn shaded [3].

singularities. After identifying a set of sensible independent coordinates, we symbolically determine the constraint Jacobian $\mathbf{C}_q$ and partition the results as in equation 2.29. Given symbolic formulas for $\mathbf{C}_{q_i}$ and $\mathbf{C}_{q_d}$, we may numerically compute the value of equation 2.28 for an arbitrarily specified independent coordinate vector $\mathbf{q}_I$ as follows:

1. Using the forward kinematic relationships derived in §2.2, compute the dependent coordinates $\mathbf{q}_D$. Combine these with the independent coordinates $\mathbf{q}_I$ according to equation 2.24 to form the system coordinate vector $\mathbf{q}$.

2. Using the system configuration determined from forward kinematics, determine the spring forces according to equation 2.33 for each body. Concatenate the individual body forces according to equation 2.21 in order to form the generalized force vector $\mathbf{Q}$.

3. Numerically compute $\mathbf{C}_{q_i}$ and $\mathbf{C}_{q_d}$ using the previously derived symbolic formulas and the system coordinates $\mathbf{q}$.

4. Compute the pseudoinverse of $\mathbf{C}_{q_d}$ according to equations 2.30 and 2.31.

5. Use equation 2.32 to calculate the $\mathbf{B}_{di}$ matrix.

6. As in equation 2.28, compute $\mathbf{B}_{di}^T\mathbf{Q}$ in order to determine whether the system is in equilibrium with the spring forces.

The procedure just listed allows the virtual work equations to be computed given a set of independent coordinates. We may determine the optimal solution to this nonlinear function using a nonlinear optimization technique such as Newton-Raphson [10].

Chapter 3

Design and Implementation of a Kinematic Model Compiler

Chapter 2 provided the kinematic and algorithmic basis for the computation of the static update step in the ECM algorithm. In this chapter, we show how a compiler may be designed which will translate a model specification into the code necessary to perform interactive motion estimation research using that model. §3.1 discusses the model specification language which provides the compiler input, while §3.2 explains how the specification may be processed to generate estimation code. See chapter 4 for examples of the types of models which may be created using this system.

3.1 Model Specification Language

There are a variety of potential ways in which the structure of kinematic models could be specified. For example, one could design a graphical user interface (GUI) which allowed bodies to be interactively resized, manipulated, and connected into an appropriate model. However, this compiler instead uses a formal specification language, much like structured programming languagues. This is thought to be a superior approach for a variety of reasons. First, the syntactic restrictions that formal languages enforce can be designed to nicely parallel the physical restrictions that models must obey. This helps prevent the accidental introduction of assumptions in the compilation process that do not generalize across all possible valid models. Also, a specification language will in general allow more precise control

over the model design than a graphical interface. Finally, by keeping the model specification format to plain text files, we retain the potential for cross-platform model exchange. Of course, if graphical model creation is desired at some point in the future, a GUI could always be designed which produces the model specification language as its output.

3.1.1 Language Structure and Style

Spiritually, the model specification language owes much of its origin to the C and C++ programming languages [13]. The basic concept is to allow the creation of rigid bodies, as well as the instantiation of constraints between those bodies, to occur using function-style declarations. Some of these functions are predefined language elements, while others may be defined by the modeler. The key difference is that the "functions" are analyzed during the model compilation process in order to create the model structure, and then never executed again. In contrast, functions in standard languages define blocks of code that will be evaluated when the compiled program is executed.

Variables and comments are borrowed directly from C++. Single-line comments (using the "//" notation) and multi-line comments (using the "/* */" notation) are both supported. Currently, only two simple variable types, *double* and *bool*, are supported, because this is all that is really necessary for model specification. The *double* type allows the specification of any necessary length, size, or position information, while the *bool* type allows conditional expressions to be used. As the model specification language specifies a conceptual structure rather than a run-time algorithm, the other low-level types available in C are extraneous.

The scope of variable declarations follows the standard C++ block strucure rules. In other words, variables names are valid within the innermost enclosing set of block delimeters. The block delimeter characters, as in C++, are curly braces. All of the standard arithmetic operations are supported, as well as assignment using the '=' operator. In addition, *if–then–else* constructs may be used to evaluate boolean expressions. Basic mathematical functions such as *sin* and *cos* are supported for modeling convenience. Display 3.1 shows an example

```
// This is a single-line comment followed by variable declarations
    double hypotenuse = 10;
    double theta = 37;
    double opposite = hypotenuse*sin(theta);  // Sample mathematical ops
    double adjacent = hypotenuse*cos(theta);

/* This is a multi-line comment
    double swap = opposite;
    opposite = adjacent;
    adjacent = swap;
*/
    if (adjacent < opposite) {
        // This block never reached because the swap code is commented
    }
    else {
        // This block of code will be evaluated instead
    }
```

Display 3.1: Sample of basic C++ language constructs supported by the model compiler.

of the types of basic constructs supported by the model specification language.

3.1.2 Literal Instantiation of Rigid Bodies

By themselves, the basic C++ language features discussed in the previous section would be of little use. They must be integrated with additional keywords and language constructs specifically designed for kinematic model specification. The largest functional block in the model specification language is the *rigidBodySystem*. This block specifies that all of the declarations contained within it pertain to a single system of kinematically constrained rigid bodies. Although there could theoretically be many *rigidBodySystem* blocks, perhaps allowing the motion of two different objects to be simultaneously estimated, all currently explored examples use only a single body system.

The interior of a *rigidBodySystem* block may contain any of the standard language declarations. Its primary role, however, is to allow the creation of rigid bodies. These may be formed using an *ellipsoidBody* block, as shown in display 3.2. These rigid body instantiations

```
rigidBodySystem <systemName> {
    ellipsoidBody <bodyName1> {
        innerLam(<double>, <double>, <double>);
        outerLam(<double>, <double>, <double>);
        // Other declarations...
    }
    ellipsoidBody <bodyName2> {
        innerLam(<double>, <double>, <double>);
        outerLam(<double>, <double>, <double>);
        // Other declarations...
    }
    // Other ellipsoidBody declarations...
}
```

Display 3.2: Syntax for instantiation of literal rigid bodies.

are termed literal because the code that defines each *ellipsoidBody* appears directly inside the *rigidBodySystem*. Rigid bodies may contain a variety of information, but the two most basic declarations required for every rigid body are the *innerLam* and *outerLam* functions. These are predefined functions which use their arguments to set the x, y, and z dimensions of the ellipsoids which define each body's shape as described in §2.1.

In order to better illustrate the various language constructs, a series of sample model specifications will be given. Over the course of these examples, we will develop a kinematic model for the PUMA 600 industrial manipulator arm [7]. This is chosen as a good example of the kinematic chains commonly studied in robotics. Our initial PUMA specification is shown in display 3.3. It has three links plus a fixed base. Note that this model would not be very useful as is, because there are currently no constraints specified between the different rigid bodies. Also, there is a large amount of repetition in the literal declaration of similar links. These concerns will be addressed in the following sections.

```
// Kinematic model of PUMA 600 robot manipulator arm
// Version 1: Literal body declarations, no joint constraints

double fac = 1.25;  // Size factor between inner and outer pdfs

rigidBodySystem puma600 {
// Literally instantiate the different body links
    ellipsoidBody base {
        double height = 1.0;
        double radius = 0.5;

        innerLam(radius, height, radius);
        outerLam(fac*radius, fac*height, fac*radius);
    }

    ellipsoidBody link1 {
        double length = 0.8;
        double radius = 0.4;

        innerLam(radius, radius, length);
        outerLam(fac*radius, fac*radius, fac*length);
    }

    ellipsoidBody link2 {
        double length = 1.0;
        double radius = 0.7;

        innerLam(radius, radius, length);
        outerLam(fac*radius, fac*radius, fac*length);
    }

    ellipsoidBody link3 {
        double length = 1.2;
        double radius = 0.4;

        innerLam(radius, radius, length);
        outerLam(fac*radius, fac*radius, fac*length);
    }
}
```

Display 3.3: First version of PUMA manipulator arm specification.

```
ellipsoidBody <funcName>(<type> <arg1>, <type> <arg2>, ...) {
    innerLam(<double>, <double>, <double>);
    outerLam(<double>, <double>, <double>);
    // Other declarations...
}

rigidBodySystem <systemName> {
    <funcName> <bodyName1>(<param1>, <param2>, ...);
    <funcName> <bodyName2>(<param1>, <param2>, ...);
    // Other functional or literal ellipsoidBody instantiations...
}
```

Display 3.4: Syntax for instantiation of functional rigid bodies.

3.1.3 Functional Instantiation of Rigid Bodies

For many kinematic models, there is a large degree of symmetry between different parts of the model. In order to take advantage of this, and to make model modifications faster and less error prone, functional rigid body declarations are introduced. The basic concept is to have blocks of code that syntactically look very much like C functions. When they are called from inside a *rigidBodySystem* block, they automatically create a new rigid body according to a list of arguments passed as parameters. In the sense that these functional body declarations are evaluated at compile time, they are actually closer to what are known as macros in traditional computer languages. However, parameter type checking and local namespaces are both used in the analysis of body functions, making them much more stable and flexible than normal macros.

Display 3.4 shows the syntax for the use of functional rigid bodies. A function definition is placed before the *rigidBodySystem* block which defines the number and type of parameters for each body function, as well as how those parameters are used to make rigid bodies. The function names may then be used to create body declarations inside the *rigidBodySystem* block. Display 3.5 shows the PUMA specification after modifications to use functional body declarations. We determine an appropriate functional structure by noting that the second

```
// Kinematic model of PUMA 600 robot manipulator arm
// Version 2: Functional body declarations, no joint constraints

double fac = 1.25;  // Size factor between inner and outer pdfs

// Function definition for planar links 2 & 3
ellipsoidBody PlanarLinkage(double length, double width, double depth,
                            bool defaultHorizontal)
{
    if (defaultHorizontal) {
        innerLam(length, width, depth);
        outerLam(fac*length, fac*width, fac*depth);
    }
    else {
        innerLam(width, length, depth);
        outerLam(fac*width, fac*length, fac*depth);
    }
}

rigidBodySystem puma600 {
// Literally instantiate the base and link 1
    ellipsoidBody base {
        double height = 1.0;
        double radius = 0.5;

        innerLam(radius, height, radius);
        outerLam(fac*radius, fac*height, fac*radius);
    }
    ellipsoidBody link1 {
        double length = 0.8;
        double radius = 0.4;

        innerLam(radius, radius, length);
        outerLam(fac*radius, fac*radius, fac*length);
    }

// Use function to instantiate links 2 & 3
    PlanarLinkage link2(1.0, 0.7, 0.2, false);
    PlanarLinkage link3(1.2, 0.4, 0.2, true);
}
```

Display 3.5: Second version of PUMA manipulator arm specification.

```
bvec <bvecName> {
    // Any standard language statements may also be used
    pos(<double>, <double>, <double>);
    orient(<double>, <double>, <double>);
}
```

Display 3.6: Syntax for instantiation of bound vectors.

and third links move in a plane defined by the first link. Note that literal and functional declarations may be freely mixed in the same model. As will be seen in §4.1, for systems with many rigid bodies functions become a critical tool for managing complexity.

3.1.4 Bound Vectors and Joint Constraints

In order to make the specifications in the previous sections useful for estimation tasks, we must provide a capability for specifying constraints between rigid bodies. The fundamental building block of constraint specifications in the model specification language is a structure known as a bound vector, specified with the keyword *bvec*. Conceptually, a bound vector is a unit vector whose base is bound to a location in space. This location may be defined either within the local coordinate frame of a body, or in inertial space. Display 3.6 shows the syntax for the declaration of bound vector blocks. The *pos* and *orient* declarations specify the x, y, and z coordinates of the bound vector position and orientation. If the *bvec* block appears inside an *ellipsoidBody* block, the bound vector is interpreted to be bound to that body's local coordinate frame. Otherwise, the *bvec* is assumed to be bound to the inertial frame.

Bound vectors provide a natural means for specifying constraints. For example, in §2.2.3 and §2.2.4, fixed and revolute joints were defined in terms of the alignment of vectors in two coordinate frames. The joint position vector **a** is specified by the *bvec pos* declaration, and the joint orientation vector **o** is specified by the *bvec orient* declaration. Given two bound vectors in different coordinate frames, we may then constrain those frames using the

```
// Create a fixed joint between two rigid bodies
constraint fixed(<body1Name>.<bvecName> , <body2Name>.<bvecName> );
// Create a revolute joint between two rigid bodies
constraint revolute(<body1Name>.<bvecName> , <body2Name>.<bvecName> );
// Create a revolute joint with a nonzero default joint angle
constraint revolute(<body1Name>.<bvecName> , <body2Name>.<bvecName> ,
                    defaultAngle );
// Create a fixed joint between a rigid body and the inertial frame
constraint fixed(<systemName>.<bodyName>.<bvecName> ,
                 <inertialBvecName> );
```

Display 3.7: Syntax for creation of joint constraints using bound vectors.

constraint keyword as shown in display 3.7.

In order to specify constraints, it is necessary to simultaneously reference the bound vectors contained within different rigid bodies by name. However, in general one or both of the *bvec* variable names will have gone out of scope by the time the *constraint* specification appears. This problem is resolved using the '.' operator, just as it is used to access object members in C++. By prepending the *bvec* name with the *ellipsoidBody* name, we allow bound vectors to be referenced outside of the body in which they are declared. Similarly, we may prepend the *ellipsoidBody* name with the *rigidBodySystem* name in order to access bound vectors outside of the body system in which they originally appeared. Display 3.7 illustrates these operations. Note that globally bound vectors, as they are always in the global namespace, may be used in constraints without any additional qualifiers.

Displays 3.8, 3.9, and 3.10 show the final version of the PUMA specification, incorporating joint constraints to appropriately control the relationships between the different bodies in the system. A fifth body has also been added representing the manipulator at the end of the arm. The model makes use of both fixed and revolute joints, which are currently the only joint types supported by the compiler. However, other joint types, such as prismatic and spherical joints, could be directly incorporated into the existing syntax, utilizing the same bound vector concept. Figure 3.1 shows the compiled model as visualized from within the

```
// Kinematic model of PUMA 600 robot manipulator arm
// Version 3: Final version including joint constraints

double fac = 1.25;  // Size factor between inner and outer pdfs

ellipsoidBody PlanarLinkage(double length, double width, double depth,
                            bool defaultHorizontal)
{
    if (defaultHorizontal) {
        innerLam(depth, width, length);
        outerLam(fac*depth, fac*width, fac*length);
    }
    else {
        innerLam(depth, length, width);
        outerLam(fac*depth, fac*length, fac*width);
    }

    bvec lowerLink {
        orient(1, 0, 0);
        if (defaultHorizontal) {
            pos(0, 0, 0.8*length);
        }
        else {
            pos(0, -0.8*length, 0);
        }
    }
    bvec upperLink {
        orient(1, 0, 0);
        if (defaultHorizontal) {
            pos(0, 0, -0.8*length);
        }
        else {
            pos(0, 0.8*length, 0);
        }
    }
}

bvec origin {
    pos(1,-2.5,10);
    orient(-1,0,0);
}
```

Display 3.8: Final version of PUMA manipulator arm specification (part 1/3).

```
rigidBodySystem puma600 {
// Literally instantiate the unique links
    ellipsoidBody base {
        double height = 1.0;
        double radius = 0.5;
        innerLam(radius, height, radius);
        outerLam(fac*radius, fac*height, fac*radius);

        bvec inertialAnchor {
            pos(0, -height, 0);
            orient(-1, 0, 0);
        }
        bvec upperLink {
            pos(0, 0.9*height, 0);
            orient(0, -1, 0);
        }
    }
    ellipsoidBody link1 {
        double length = 0.8;
        double radius = 0.4;
        innerLam(length, radius, radius);
        outerLam(fac*length, fac*radius, fac*radius);

        bvec lowerLink {
            pos(0.3*length, -0.5*radius, 0);
            orient(0, -1, 0);
        }
        bvec upperLink {
            pos(-0.7*length, 0, 0);
            orient(1, 0, 0);
        }
    }
    ellipsoidBody manipulator {
        double radius = 0.3;
        innerLam(radius, radius, radius);
        outerLam(fac*radius, fac*radius, fac*radius);
        bvec lowerLink {
            pos(0, 0, 1.5*radius);
            orient(1, 0, 0);
        }
    }
```

Display 3.9: Final version of PUMA manipulator arm specification (part 2/3).

```
// Use functions to instantiate the planar links
    PlanarLinkage link2(1.0, 0.7, 0.2, false);
    PlanarLinkage link3(1.2, 0.4, 0.2, true);

// Constrain the four arm links together
    constraint revolute(base.upperLink, link1.lowerLink, 180);
    constraint revolute(link1.upperLink, link2.lowerLink, 90);
    constraint revolute(link2.upperLink, link3.lowerLink);
    constraint fixed(link3.upperLink, manipulator.lowerLink);
}

// Constrain the base to the inertial frame
constraint revolute(puma600.base.inertialAnchor, origin);
```

Display 3.10: Final version of PUMA manipulator arm specification (part 3/3).

graphical interface. Compare the structure of the completed model with the specification file to get a sense of how bound vector relationships translate into rigid body constraints. The means by which this compilation occurs will be examined in the following section.

3.2 Code Generation

The model specification language discussed in the previous section provides a succint means of specifying the structure of a system of kinematically constrained rigid bodies. All of the information needed to apply the ECM algorithm, or any of a host of other kinematic estimation or simulation techniques, is fully defined by the model specification. The task before the compiler, then, is to parse this information, transforming it into code which implements the algorithm of choice. The kinematic model compiler described here uses a variety of software tools and techniques, drawn from a diverse range of application areas, to accomplish this goal. The primary steps in the compilation process are summarized by the following list.

1. The model specification is parsed using a grammar corresponding to the specification language, producing a data structure which stores all relevant body and constraint

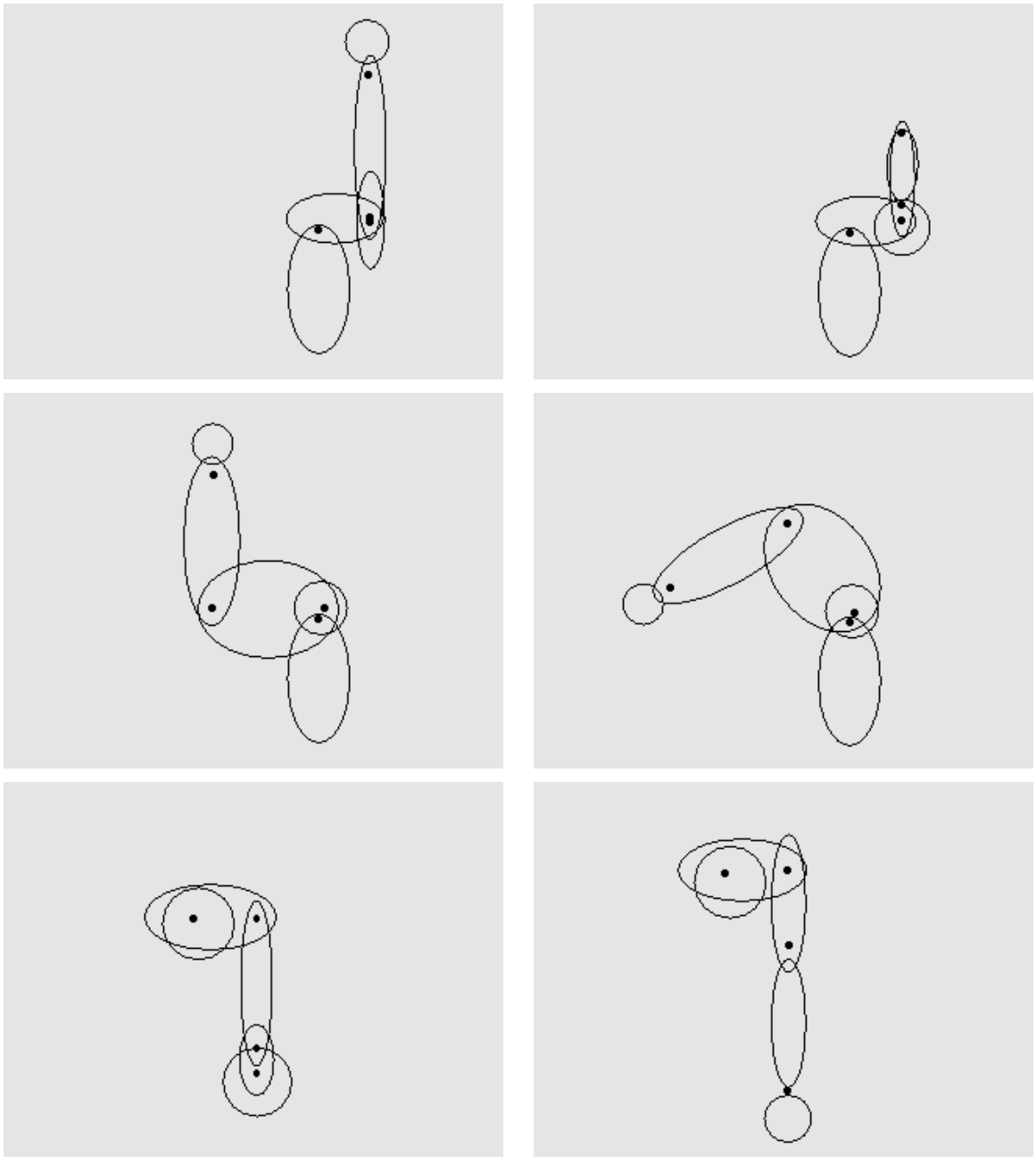


Figure 3.1: Compiled model of PUMA 600 robotic manipulator arm. The left and right columns show two different arm positions from front, side, and overhead viewing angles.

definitions.

2. The model data is analyzed to check that the model structure is kinematically valid. Relevant information needed by later compilation phases is extracted.

3. The coordinates for the bodies in the system are partitioned into a sensible set of dependent and independent coordinates as described in §2.3.

4. The constraint equations are formed from the model joint information, and their symbolic jacobian is computed using Mathematica.

5. The constraint jacobian is partitioned according to equation 2.29, and C++ methods are generated to compute the $\mathbf{C}_{q_i}$ and $\mathbf{C}_{q_d}$ matrices.

6. The resulting jacobian computation methods, as well as other code necessary to define the model structure, are adapted to interface with the software framework discussed in [3].

The following sections will discuss these steps in more detail. Although many of these steps require hundreds of lines of C++ code to implement, the discussion will be kept at a high level. The primary focus will be on explaining the software tools involved, and the means by which they may be integrated into a coherent compiler structure. Except where otherwise noted, the compilation process is implemented using C++. All of the design and testing of the compiler was conducted using Microsoft Visual C++ 6.0 on a Windows NT workstation. However, with the exception of the graphical interface, the code uses only standard ANSI-compliant C++ constructs [13]. Therefore, it could easily be ported to another operating system at some point in the future.

3.2.1 Model Specification Parsing and Analysis

The model specification language described in §3.1 is based on C and C++, and keeps their well-defined grammatical structure. Therefore, specification files may be analyzed by defining

a formal grammar for the language and then parsing that grammar. Two standard compiler design tools, called Lex and Yacc [8], greatly facilitate the grammar specification process. Lex, which is a lexer, breaks the input text into appropriate symbolic units. Yacc, which is a parser, recognizes relationships between these symbolic units and then takes actions based on these relationships. For this project, the GNU freeware versions of Lex and Yacc, called Flex and Bison, were actually used. However, the differences are for the most part inconsequential. The grammar itself is basically a subset of the C++ grammar [1], with modifications to allow for special blocks like *rigidBodySystem*, *ellipsoidBody*, and *bvec*.

The actions taken by the Yacc grammar cause rigid bodies, bound vectors, joint constraints, and other model information to be stored in a data structure, which is implemented as a hierarchy of C++ classes. Once the end of the specification file is reached, this data structure is analyzed for consistency. For example, we verify that there is only a single joint between any two bodies, and check that one of the bodies is bound to the inertial frame. Then, an appropriate traversal order of the bodies for kinematic updates is determined. Currently, as loops in the kinematic chain are not supported, the kinematic update may be performed using a simple preorder traversal of the tree structure formed by the joint constraints [1]. After the traversal order has been determined, the body structure is transformed into an alternate data structure representation which is more suitable for the subsequent compilation steps.

3.2.2 Generation of Virtual Work Equations

Given a set of bodies connected by joints, it is straightforward to generate the appropriate constraint equations using the kinematic results in §2.2. Where appropriate, the constraint equations are augmented for numerical stability as described in §2.3.2. Also, for kinematic chains without loops, coordinate partitioning as discussed in §2.3.1 is easily achieved. For each body, we choose the independent coordinates to be those that make the forward kinematics simplest. The number of independent coordinates in each body is equal to the number

of degrees of freedom in the joint connecting that body to the previous body in the traversal order. When the compiler is enhanced to support loops in the kinematic structure, a more complex coordinate partitioning scheme will have to be determined. For very complex structures, numerical partitioning schemes are generally recommended [11].

Once the constraint equations have been determined, it is necessary to determine the symbolic form of the constraint jacobian. This task is accomplished using Mathematica 3.0's extensive symbolic manipulation capabilities. The compiler generates mathematica commands which define the constraint equations, form these equations into a system, and then derive the jacobian of that system. Given the coordinate partitioning already determined, the resulting $\mathbf{C}_q$ matrix is easily divided into $\mathbf{C}_{q_i}$ and $\mathbf{C}_{q_d}$. The entries of these matrices are output by Mathematica and read back in by the compiler. The compiler then creates C++ methods which, given the current model inertial coordinates, automatically compute the numeric values of $\mathbf{C}_{q_i}$ and $\mathbf{C}_{q_d}$. These routines are designed to fit into a handcoded set of functions which implement the virtual work procedure described in §2.3.3.

3.2.3 Integration with ECM Graphical Interface

As part of his Ph.D. thesis research on the ECM algorithm, Edward Hunter implemented a software framework [3] comprising over 15,000 lines of C++ code. It provides a graphical interface which may be used to manipulate models, adjust algorithm parameters, and interactively apply the ECM algorithm to digitized video data. Rather than duplicating such an extensive programming effort, the model compiler was designed to interface with and extend the existing software structure.

The original software structure was specialized for bipedal motion, and therefore used planar kinematics. For the compiler, the internal data structures were augmented to store the necessary three-dimensional body and joint information. In addition, the kinematic relationships described in §2.2.1 were implemented so that three-dimensional body configurations could be appropriately computed. Routines were written that could perform the

static update, as described in §2.3.3, on any rigid body system using the constraint jacobian functions generated by the model compiler. A variety of other minor changes throughout the code were also made in order to fully accomodate the introduction of three-dimensional kinematics.

Given these changes to the software framework, completing the model compilation process is straightforward. The compiled methods which generate the constraint jacobian are placed in a header file which is copied to the directory containing the software framework. In addition, another header which defines the model structure using the existing software framework methods is generated by the model compiler and copied. The ECM graphical interface is then recompiled using these headers, producing an executable specialized to the compiled model. Using this executable, the ECM may be interactively applied to the model structure specified in the original specification file, as demonstrated in chapter 4.

Chapter 4

Experimental Results

4.1 Sample Compiled Models

In this section, we present the results of the application of the model compiler to four different sample model specifications. First, the model specification file is given. Then, several images are shown of sample model configurations as viewed from different perspectives. These model configurations were generated by interactively manipulating joint angles with the graphical interface. Finally, images of sample static updates, as computed using the compiled virtual work equations, are shown. The initial conditions for the static update are manually defined by manipulating intermediate bodies representing the M-step results. This allows us to determine a potential M-step result, with as much noise or bias as we like, and then verify that the static update does indeed find the kinematic configuration which most closely matches that M-step. As the ECM algorithm itself has already been subjected to a rigorous quantitative analysis [3], the results in this section are restricted to a qualitative proof of the model compiler concept.

In order to understand why the results in this chapter are presented using synthetic M-step results rather than processed video data, some knowledge of the history of the ECM algorithm is helpful. Prior to the model compiler design, the software was specialized to the estimation of human bipedal motion, for which the kinematics may be reduced to a much simpler planar structure. The model compiler completely redesigns the static update algorithm component to utilize three-dimensional kinematics. As the E and M steps are

separately processed for each component, they may be handcoded once without the need for compilation. Therefore, the compiler component of the framework is complete. However, the existing M-step makes assumptions for efficiency purposes which depend on planar kinematics. Before the complete three-dimensional algorithm may be applied to video data, the M-step must be redesigned to allow the possibility of non-planar motion. This is not a difficult task, but it will require a few weeks of design and testing that time constraints did not allow.

The synthetic M-steps used in this chapter are viewed as a sensible compromise, in that earlier work on the ECM algorithm indicates that the eventual creation of a functional M-step should not pose any serious difficulties. Also, synthetic M-steps are not necessarily a disadvantage. As the motion between subsequent video frames is very small, the M-step results will in general be very close to the previous body configuration. The much larger synthetic M-steps used in this section place a larger demand on the virtual work equations, which allows a more rigorous test of the static update's stability and robustness. The following sections will demonstrate that the principle of virtual work does an extremely effective job of finding a global compromise between contradictory M-step results.

4.1.1 PUMA 600 Robotic Manipulator Arm

In §3.1, the PUMA 600 manipulator arm model specification was developed, and screen shots of the resulting model were shown in figure 3.1. In this section, we show two sample static updates for the PUMA arm. Figure 4.1 shows the first static update, which involves the planar motion of only the second and third links. The intermediate bodies are drawn in a slightly lighter shade of gray with red arrows connecting the means of the primary and intermediate bodies. The arrows represent the linear spring forces acting on the bodies. There are torsional spring forces as well, but these are currently not graphically represented. A careful examination of the intermediate bodies prior to the static update, as shown in the left column, reveals that their joint axes do not align, and that a perfect kinematic update

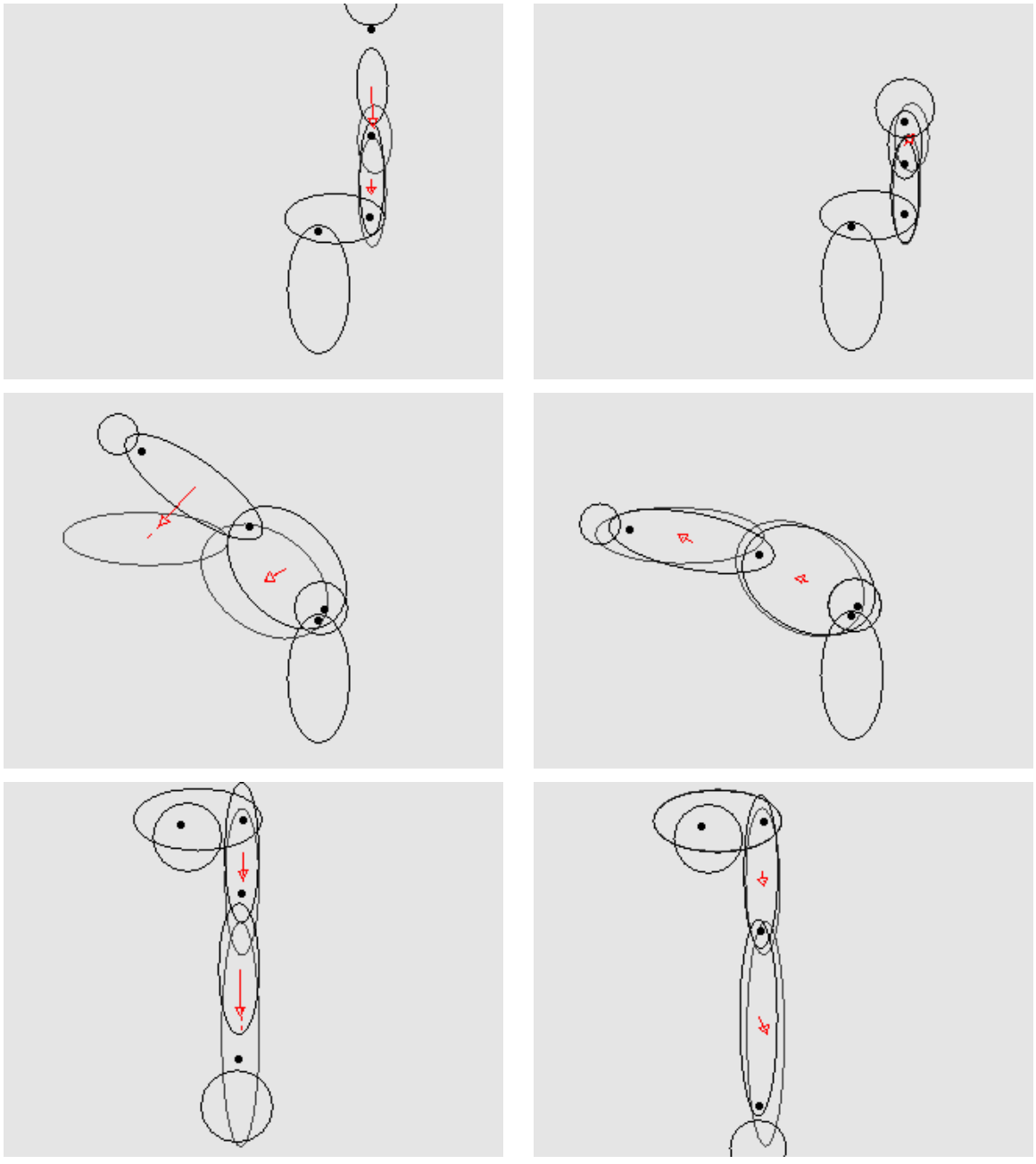


Figure 4.1: PUMA arm sample static update #1. The left and right columns show the primary and intermediate body configurations before and after the static update.

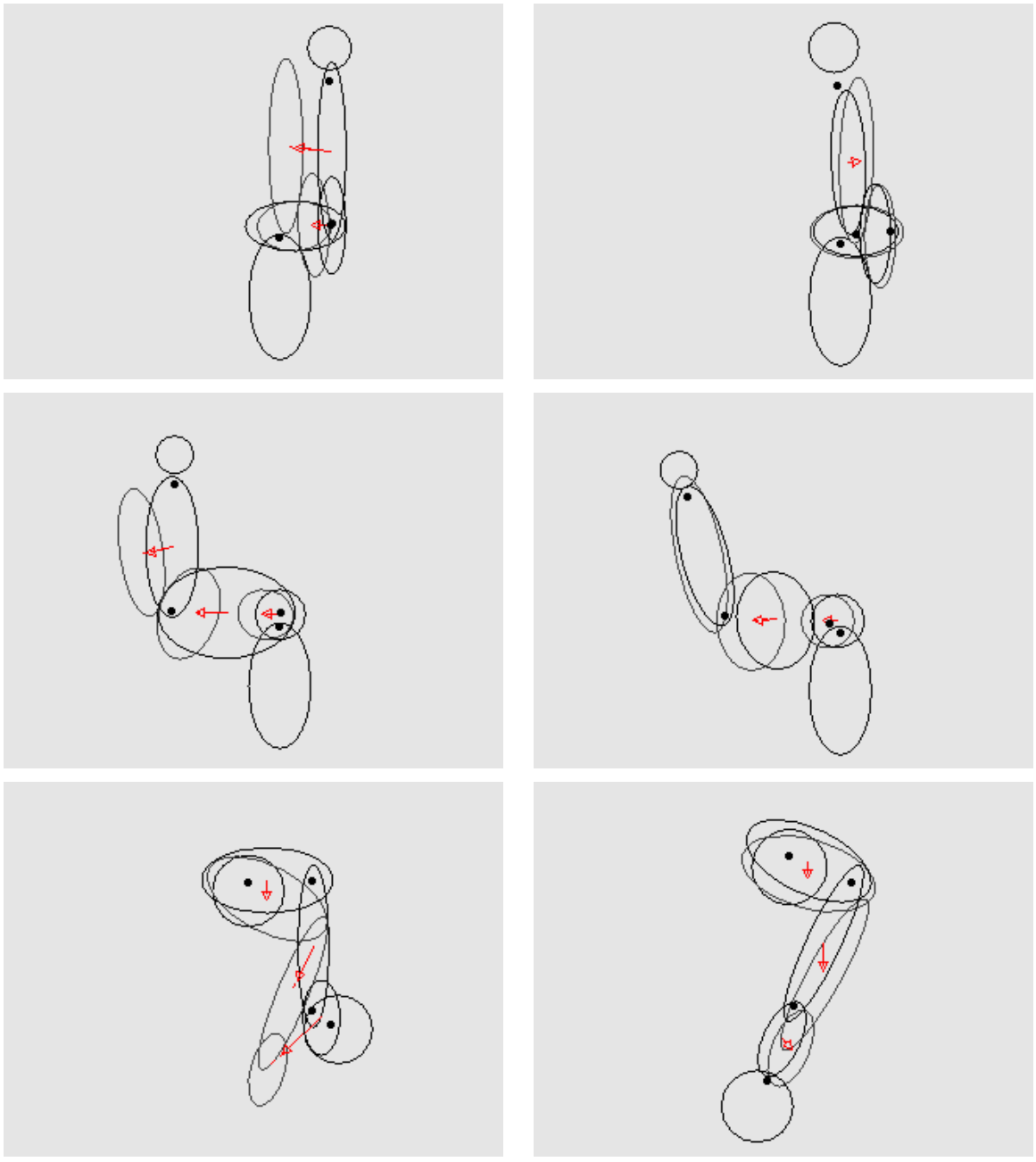


Figure 4.2: PUMA arm sample static update #2. The left and right columns show the primary and intermediate body configurations before and after the static update.

is therefore not possible. As a consequence, the post static update bodies, as shown in the right column, still have forces acting upon them. However, these forces are balanced so that the body system is in equilibrium. We can see that the static update effectively finds a global balance between the conflicting demands of the different intermediate bodies.

Figure 4.2 shows the second static update. This time there is rotation of the first link as well as movement of the planar joints, which produces a net three-dimensional motion for the second and third links. However, we see that the static update still finds a reasonable equilibrium position between the conflicting M-step bodies.

4.1.2 Human Lower Body Bipedal Motion Model

The human lower body bipedal model examined in this section is identical in structure to the model utilized in [3]. However, earlier implementations of the ECM used planar kinematics, while the model compiler static update uses three-dimensional kinematics. Displays 4.1, 4.2, and 4.3 show the specification file for the lower-body model. Figure 4.3 shows the compiled model in both the default resting pose and taking a step. The joint axes are drawn on the plots to help make the kinematic structure more clear. Note the use of functions in the model specfication to capture the parallelism between the right and left halves of the body. Also, note how joint initial conditions were used to give the legs a more lifelike starting posture.

Figures 4.4 and 4.5 show the lower body bipedal model following two completed static updates. The first static update takes the model from rest to an intermediate step postion, and the second update takes the model from the intermediate posture to a full step. Note the noise and inaccuracies in the M-step positioning, including extraneous offsets perpendicular to the direction of motion. The kinematic update still finds the configuration which most effectively balances the applied forces.

```
// Human lower-body model for bipedal motion estimation
double fac = 1.25;  // Size factor between inner and outer pdfs

ellipsoidBody Thigh(double length, double radius, bool leftSide) {
    innerLam(radius,radius,length);
    outerLam(fac*radius, fac*radius, fac*length);
    bvec hip {
        if (leftSide) {
            pos(-0.1*radius, 0, 0.8*length);
        }
        else {
            pos(0.1*radius, 0, 0.8*length);
        }
        orient(-1, 0, 0);
    }
    bvec knee {
        pos(0, 0, -0.8*length);
        orient(-1, 0, 0);
    }
}

ellipsoidBody Shin(double length, double radius) {
    innerLam(radius,radius,length);
    outerLam(fac*radius, fac*radius, fac*length);
    bvec knee {
        pos(0, 0, 0.8*length);
        orient(-1, 0, 0);
    }
    bvec ankle {
        pos(0, 0, -0.8*length);
        orient(-1, 0, 0);
    }
}

ellipsoidBody Foot(double length, double width, double height) {
    innerLam(width, height, length);
    outerLam(fac*width, fac*height, fac*length);
    bvec ankle {
        pos(0, 0, -0.8*length);
        orient(-1, 0, 0);
    }
}
```

Display 4.1: Human lower body bipedal motion model specification (part 1/3).

```
bvec origin {
    pos(0,0,8);
    orient(0,1,0);
}

rigidBodySystem lowerBodyBipedal {
    ellipsoidBody trunk {
        double width = 0.45;
        double height = 0.64;
        double depth = 0.27;

        innerLam(width, height, depth);
        outerLam(fac*width, fac*height, fac*depth);

        bvec neck {
            pos(0, 0.95*height, 0);
            orient(-1, 0, 0);
        }
        bvec waist {
            pos(0, -0.85*height, 0);
            orient(-1, 0, 0);
        }
    }
    ellipsoidBody pelvis {
        double width = 0.45;
        double height = 0.32;
        double depth = 0.27;

        innerLam(width, height, depth);
        outerLam(fac*width, fac*height, fac*depth);

        bvec inertialAnchor {
            pos(0, 0, 0);
            orient(0, 1, 0);
        }
        bvec waist {
            pos(0, 0.8*height, 0);
            orient(-1, 0, 0);
        }
```

Display 4.2: Human lower body bipedal motion model specification (part 2/3).

```
        bvec leftHip {
            pos(0.6*width, -0.7*height, 0);
            orient(-1, 0, 0);
        }
        bvec rightHip {
            pos(-0.6*width, -0.7*height, 0);
            orient(-1, 0, 0);
        }
    }
    ellipsoidBody head {
        double radius = 0.23;
        double height = 0.3;

        innerLam(radius, height, radius);
        outerLam(fac*radius, fac*height, fac*radius);

        bvec neck {
            pos(0, -0.95*height, 0);
            orient(-1, 0, 0);
        }
    }

    Foot rightFoot(0.27, 0.18, 0.14);
    Foot leftFoot(0.27, 0.18, 0.14);
    Thigh rightThigh(0.41, 0.23, false);
    Thigh leftThigh(0.41, 0.23, true);
    Shin leftShin(0.55, 0.23);
    Shin rightShin(0.55, 0.23);

    constraint fixed(pelvis.waist, trunk.waist);
    constraint fixed(trunk.neck, head.neck);

    double postureDelta = 10;   // To keep knees from being locked
    constraint revolute(pelvis.leftHip, leftThigh.hip, 90-postureDelta);
    constraint revolute(pelvis.rightHip, rightThigh.hip, 90-postureDelta);
    constraint revolute(leftShin.knee, leftThigh.knee, 2*postureDelta);
    constraint revolute(rightThigh.knee, rightShin.knee, 2*postureDelta);
    constraint revolute(rightFoot.ankle, rightShin.ankle, 90-postureDelta);
    constraint revolute(leftFoot.ankle, leftShin.ankle, 90-postureDelta);
}

constraint revolute(lowerBodyBipedal.pelvis.inertialAnchor, origin);
```

Display 4.3: Human lower body bipedal motion model specification (part 3/3).

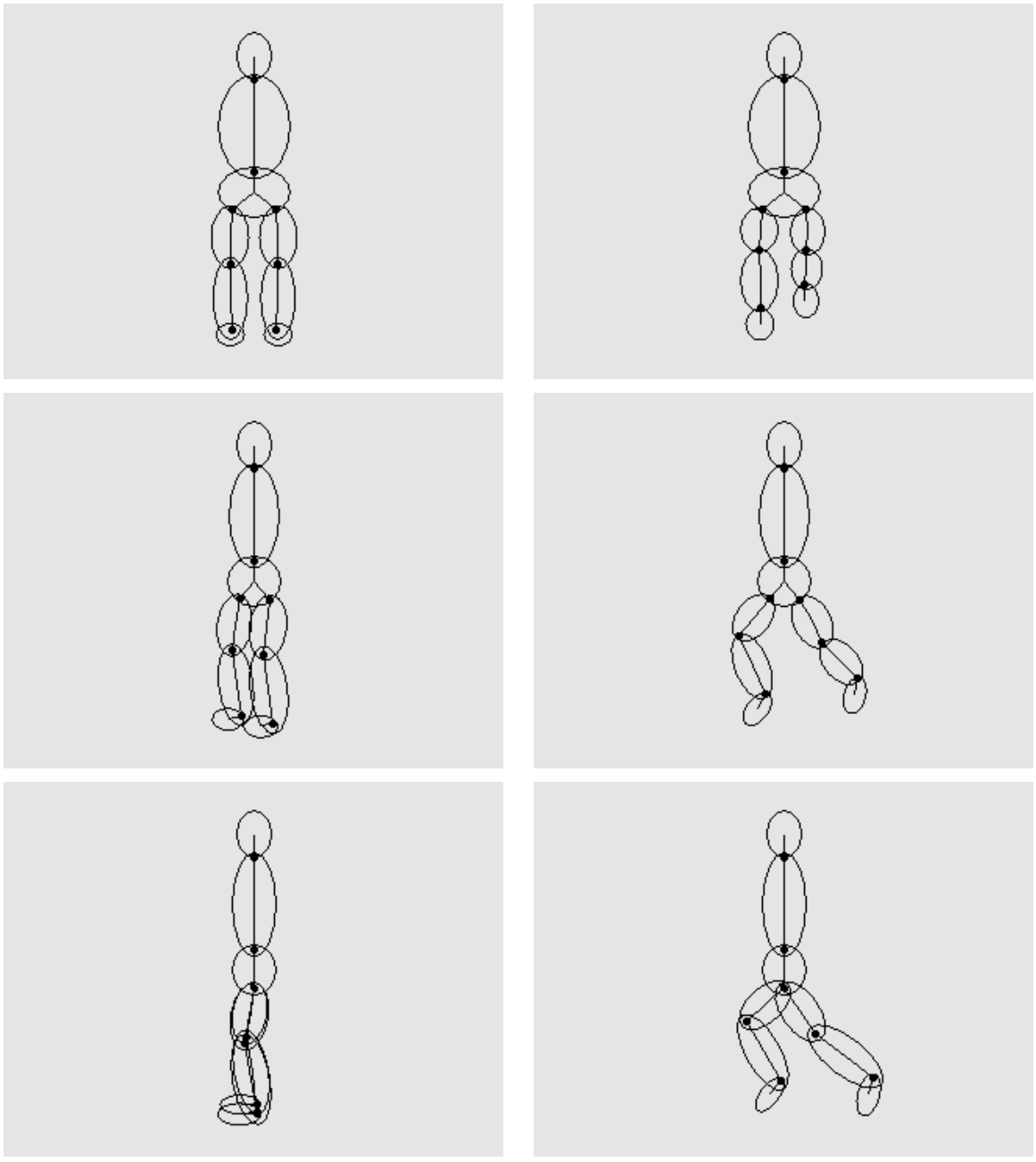


Figure 4.3: Compiled model of human lower body bipedal motion. The left and right columns show the figure at the default rest position and taking a step from different angles.

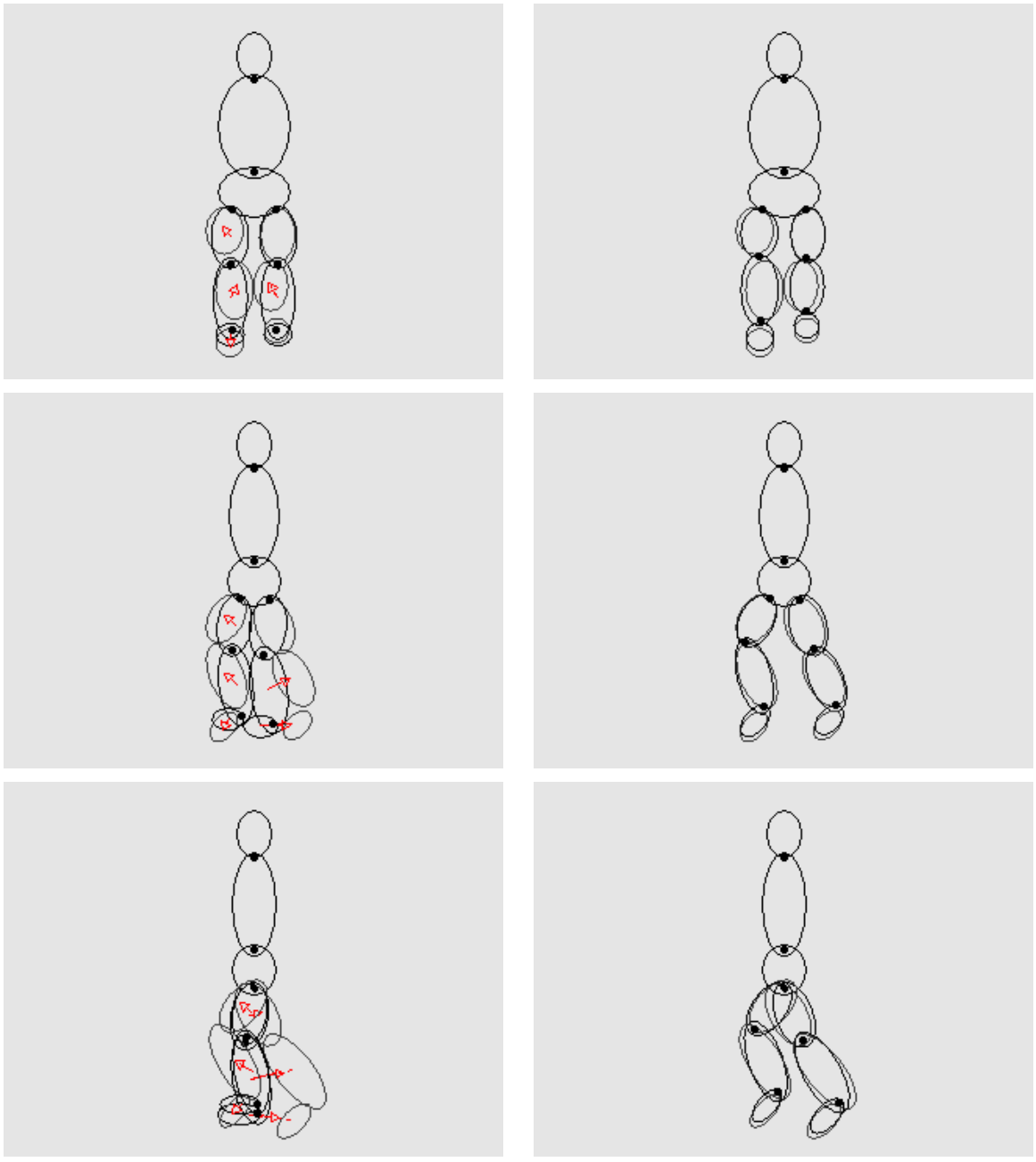


Figure 4.4: Lower body bipedal model static update #1. The left and right columns show the primary and intermediate body configurations before and after the static update.

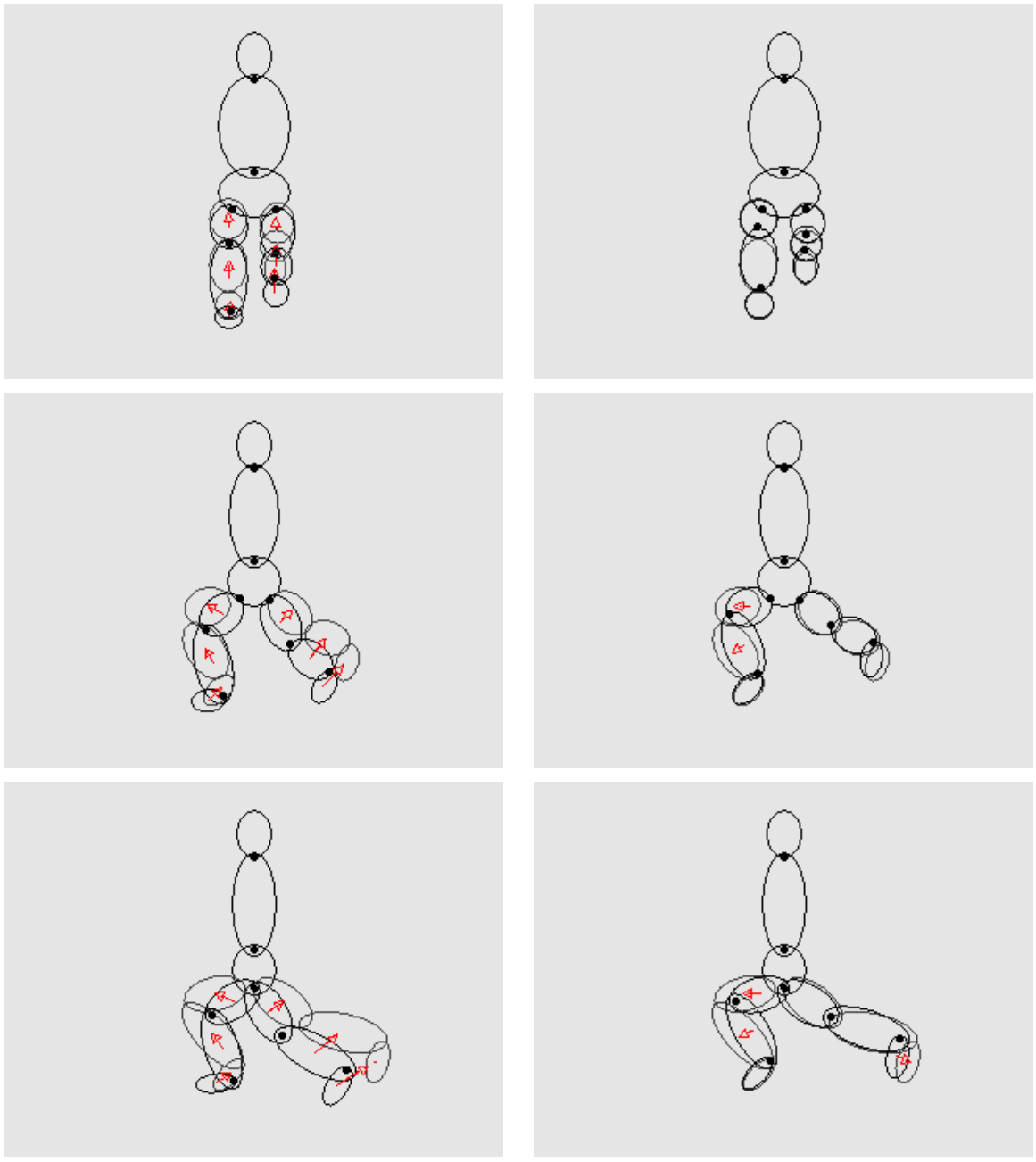


Figure 4.5: Lower body bipedal model static update #2. The left and right columns show the primary and intermediate body configurations before and after the static update.

4.1.3 Human Full Body Model

The human full body model examined in this section is a constrained version of the full body model which is specified in [3], but which has never been implemented because of the kinematic complexity. It is constrained because the spherical joints at the shoulders and twist-and-bend joints at the elbows have been replaced with revolute joints. This was necessary because only revolute and fixed joints are currently supported by the model compiler. However, this restriction is actually desireable for some applications. Consider the estimation of the articulated motion of a jogger from a single camera view. Due to projection ambiguities, there will be insufficient information to accurately estimate a fully spherical shoulder joint. However, as the modeler knows that joggers tend to swing their arms back and forth rather than waving them wildly, he may use an appropriately oriented revolute joint to constrain the shoulder estimates to the most probable set of orientations.

The full body model in this section uses shoulder and elbow joints designed to mimic a jogging motion. Displays 4.4–4.8 give the model specification file for the full body model. Figure 4.6 shows the default jogger configuration, as well as views of the model in midstride. Note that the shoulder is set to swing parallel to the body, as in a running motion. However, the elbow angle is slightly offset towards the center of the body, mimicing the way runners swing their forearms at an angle. Figure 4.7 shows some more exotic poses of the full body model, demonstrating the wide range of postures that can be attained even without high degree-of-freedom joints.

Figures 4.8 and 4.9 show two sample static updates for the full body model. They show an update from rest to an intermediate jogging position, and then from that intermediate position to full stride. Once again, the virtual work procedure is seen to find positions which effectively compromise between the mutually unattainable M-step results.

```
// Human full-body model with arm movements matched to runner
double fac = 1.25;  // Size factor between inner and outer pdfs

ellipsoidBody UpperArm(double length, double radius, bool leftSide) {
    innerLam(radius, length, radius);
    outerLam(fac*radius, fac*length, fac*radius);
    bvec shoulder {
        if (leftSide) {
            pos(-radius, -0.9*length, 0);
        }
        else {
            pos(radius, -0.9*length, 0);
        }
        orient(-1, 0, 0);
    }
    bvec elbow {
        pos(0, 0.8*length, 0);
        orient(1, 0, 0);
    }
}

ellipsoidBody ForeArm(double length, double radius, double offsetAngle) {
    innerLam(radius, length, radius);
    outerLam(fac*radius, fac*length, fac*radius);
    bvec elbow {
        pos(0, -0.8*length, 0);
        orient(cos(offsetAngle), -sin(offsetAngle), 0);
    }
    bvec wrist {
        pos(0, 0.8*length, 0);
        orient(1, 0, 0);
    }
}

ellipsoidBody Hand(double length, double radius) {
    innerLam(radius, length, radius);
    outerLam(fac*radius, fac*length, fac*radius);
    bvec wrist {
        pos(0, -0.7*length, 0);
        orient(1, 0, 0);
    }
}
```

Display 4.4: Human full body model specification (part 1/5).

```
ellipsoidBody Thigh(double length, double radius, bool leftSide) {
    innerLam(radius,radius,length);
    outerLam(fac*radius, fac*radius, fac*length);
    bvec hip {
        if (leftSide) {
            pos(-0.1*radius, 0, 0.8*length);
        }
        else {
            pos(0.1*radius, 0, 0.8*length);
        }
        orient(-1, 0, 0);
    }
    bvec knee {
        pos(0, 0, -0.8*length);
        orient(-1, 0, 0);
    }
}
ellipsoidBody Shin(double length, double radius) {
    innerLam(radius,radius,length);
    outerLam(fac*radius, fac*radius, fac*length);
    bvec knee {
        pos(0, 0, 0.8*length);
        orient(-1, 0, 0);
    }
    bvec ankle {
        pos(0, 0, -0.8*length);
        orient(-1, 0, 0);
    }
}
ellipsoidBody Foot(double length, double width, double height) {
    innerLam(width, height, length);
    outerLam(fac*width, fac*height, fac*length);
    bvec ankle {
        pos(0, 0, -0.8*length);
        orient(-1, 0, 0);
    }
}

bvec origin {
    pos(0,0,8);
    orient(0,1,0);
}
```

Display 4.5: Human full body model specification (part 2/5).

```
rigidBodySystem lowerBodyBipedal {
    ellipsoidBody trunk {
        double width = 0.45;
        double height = 0.64;
        double depth = 0.27;
        innerLam(width, depth, height);
        outerLam(fac*width, fac*depth, fac*height);

        bvec neck {
            pos(0, 0, 0.95*height);
            orient(-1, 0, 0);
        }
        bvec waist {
            pos(0, 0, -0.85*height);
            orient(-1, 0, 0);
        }
        bvec leftShoulder {
            pos(0.95*width, 0, 0.8*height);
            orient(-1, 0, 0);
        }
        bvec rightShoulder {
            pos(-0.95*width, 0, 0.8*height);
            orient(-1, 0, 0);
        }
    }

    ellipsoidBody pelvis {
        double width = 0.45;
        double height = 0.32;
        double depth = 0.27;
        innerLam(width, height, depth);
        outerLam(fac*width, fac*height, fac*depth);

        bvec inertialAnchor {
            pos(0, 0, 0);
            orient(0, 1, 0);
        }
        bvec waist {
            pos(0, 0.8*height, 0);
            orient(-1, 0, 0);
        }
```

Display 4.6: Human full body model specification (part 3/5).

```
        bvec leftHip {
            pos(0.6*width, -0.7*height, 0);
            orient(-1, 0, 0);
        }
        bvec rightHip {
            pos(-0.6*width, -0.7*height, 0);
            orient(-1, 0, 0);
        }
    }

    ellipsoidBody head {
        double radius = 0.23;
        double height = 0.3;

        innerLam(radius, radius, height);
        outerLam(fac*radius, fac*radius, fac*height);

        bvec neck {
            pos(0, 0, -0.95*height);
            orient(-1, 0, 0);
        }
    }

// Build and constrain lower body
    Foot rightFoot(0.27, 0.18, 0.14);
    Foot leftFoot(0.27, 0.18, 0.14);
    Thigh rightThigh(0.41, 0.23, false);
    Thigh leftThigh(0.41, 0.23, true);
    Shin leftShin(0.55, 0.23);
    Shin rightShin(0.55, 0.23);

    double postureDelta = 10;   // To keep knees from being locked
    constraint revolute(pelvis.leftHip, leftThigh.hip, 90-postureDelta);
    constraint revolute(pelvis.rightHip, rightThigh.hip, 90-postureDelta);
    constraint revolute(leftShin.knee, leftThigh.knee, 2*postureDelta);
    constraint revolute(rightThigh.knee, rightShin.knee, 2*postureDelta);
    constraint revolute(rightFoot.ankle, rightShin.ankle, 90-postureDelta);
    constraint revolute(leftFoot.ankle, leftShin.ankle, 90-postureDelta);
```

Display 4.7: Human full body model specification (part 4/5).

```
// Build and constrain upper body
    double elbowOffset = 10;
    UpperArm leftUpperArm(0.45, 0.18, true);
    UpperArm rightUpperArm(0.45, 0.18, false);
    ForeArm leftForeArm(0.4, 0.18, elbowOffset);
    ForeArm rightForeArm(0.4, 0.18, -elbowOffset);
    Hand leftHand(0.2, 0.15);
    Hand rightHand(0.2, 0.15);

    double elbowAngle = 15;
    constraint revolute(pelvis.waist, trunk.waist, 90);
    constraint fixed(trunk.neck, head.neck);
    constraint revolute(trunk.leftShoulder, leftUpperArm.shoulder, 90);
    constraint revolute(trunk.rightShoulder, rightUpperArm.shoulder, 90);
    constraint revolute(leftUpperArm.elbow, leftForeArm.elbow, elbowAngle);
    constraint revolute(rightUpperArm.elbow,rightForeArm.elbow,elbowAngle);
    constraint fixed(leftForeArm.wrist, leftHand.wrist);
    constraint fixed(rightForeArm.wrist, rightHand.wrist);
}

constraint revolute(lowerBodyBipedal.pelvis.inertialAnchor, origin);
```

Display 4.8: Human full body model specification (part 5/5).

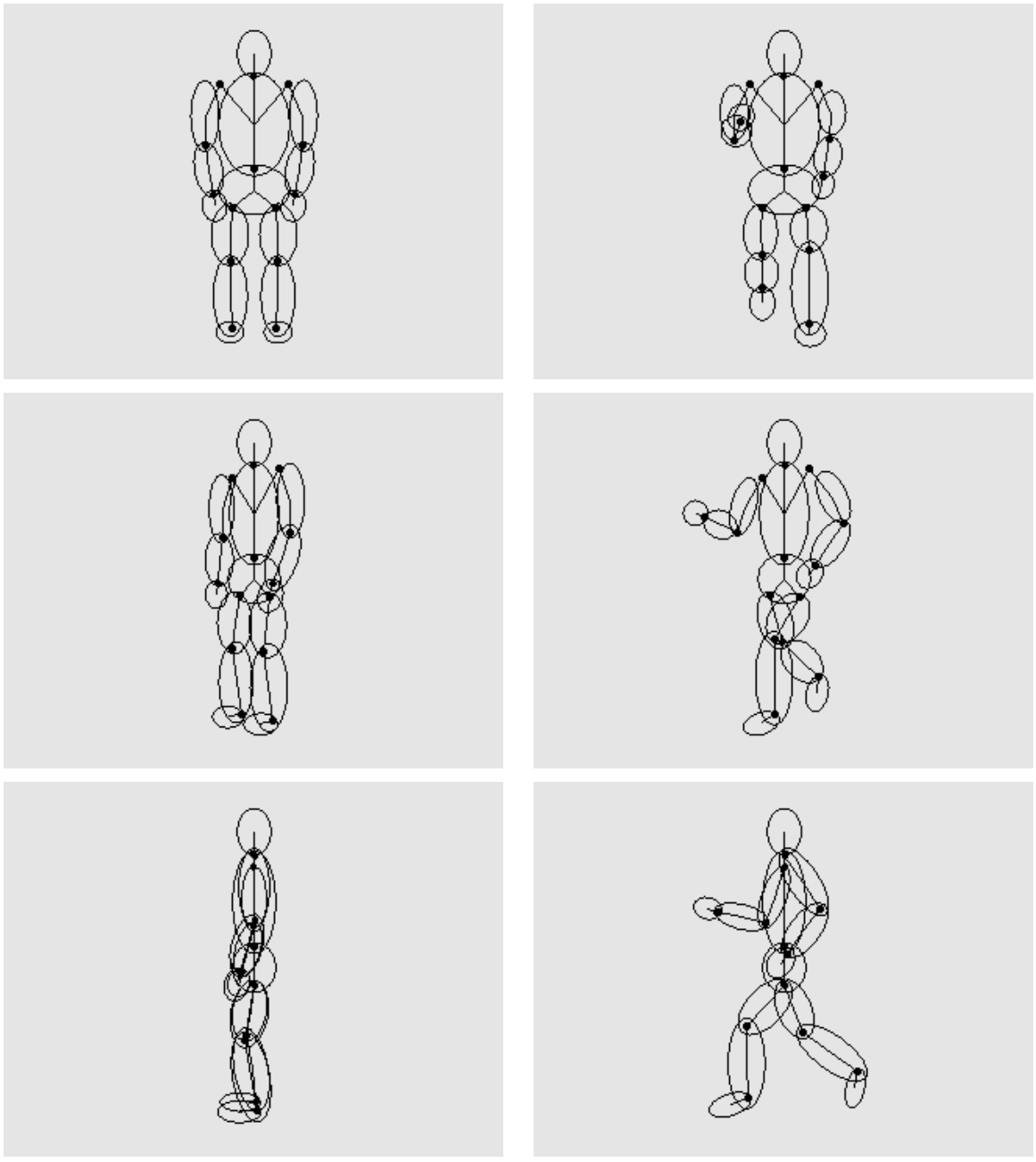


Figure 4.6: Compiled full body model specialized for jogging motion. The left and right columns show the figure at the default rest position and in midstride.

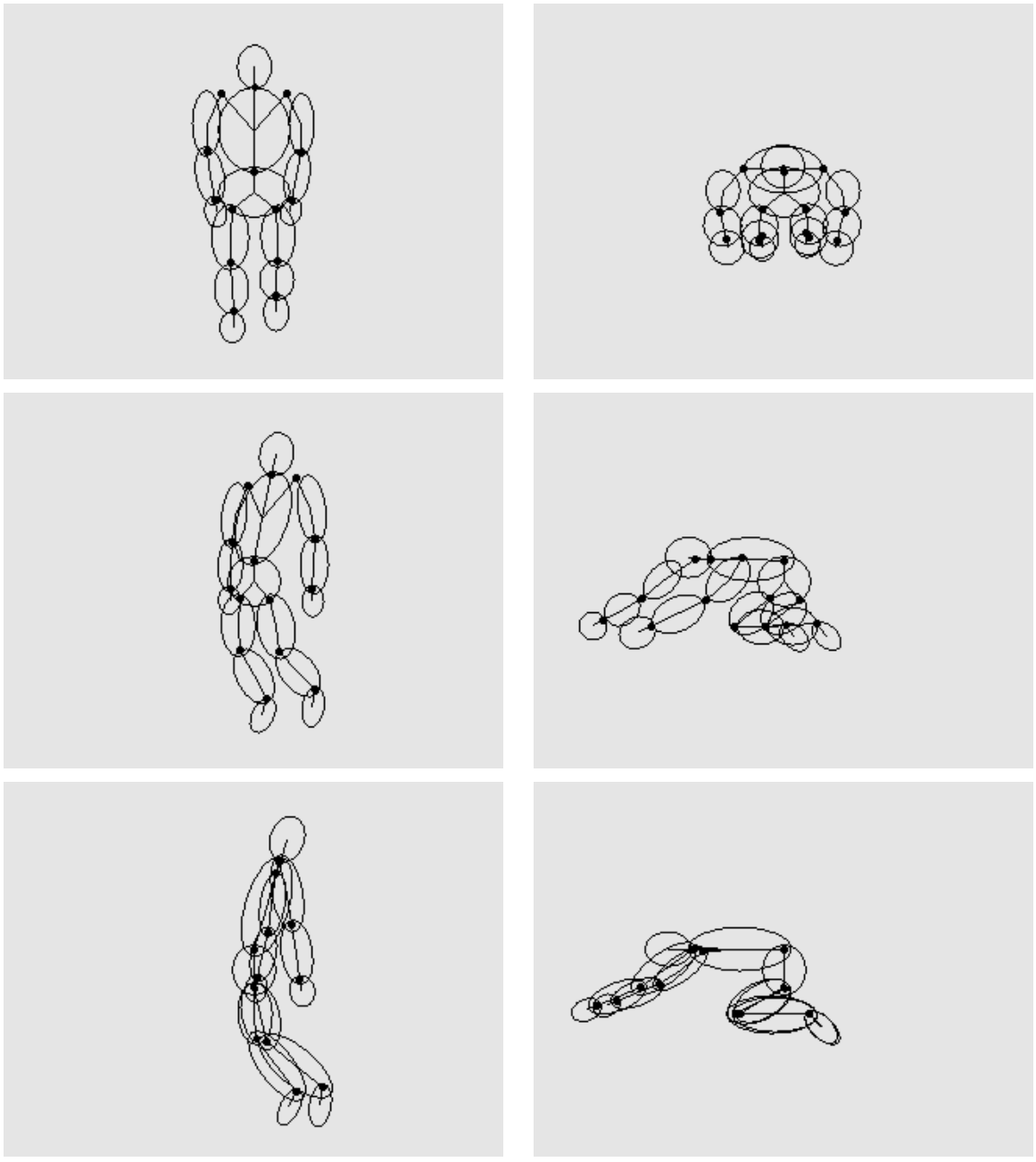


Figure 4.7: More unusual poses of the full body model: leaping and prayer.

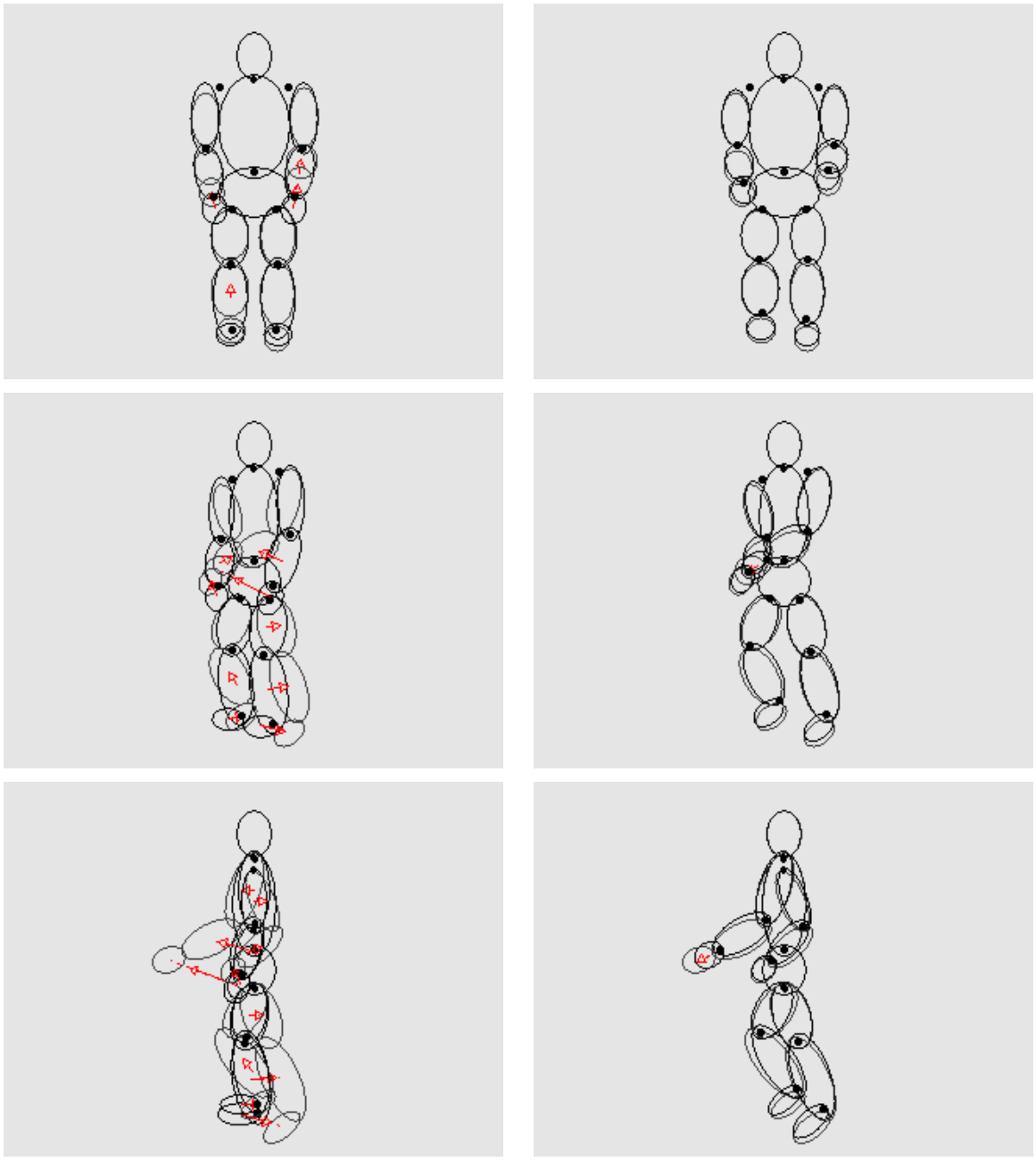


Figure 4.8: Full body human model static update #1. The left and right columns show the primary and intermediate body configurations before and after the static update.

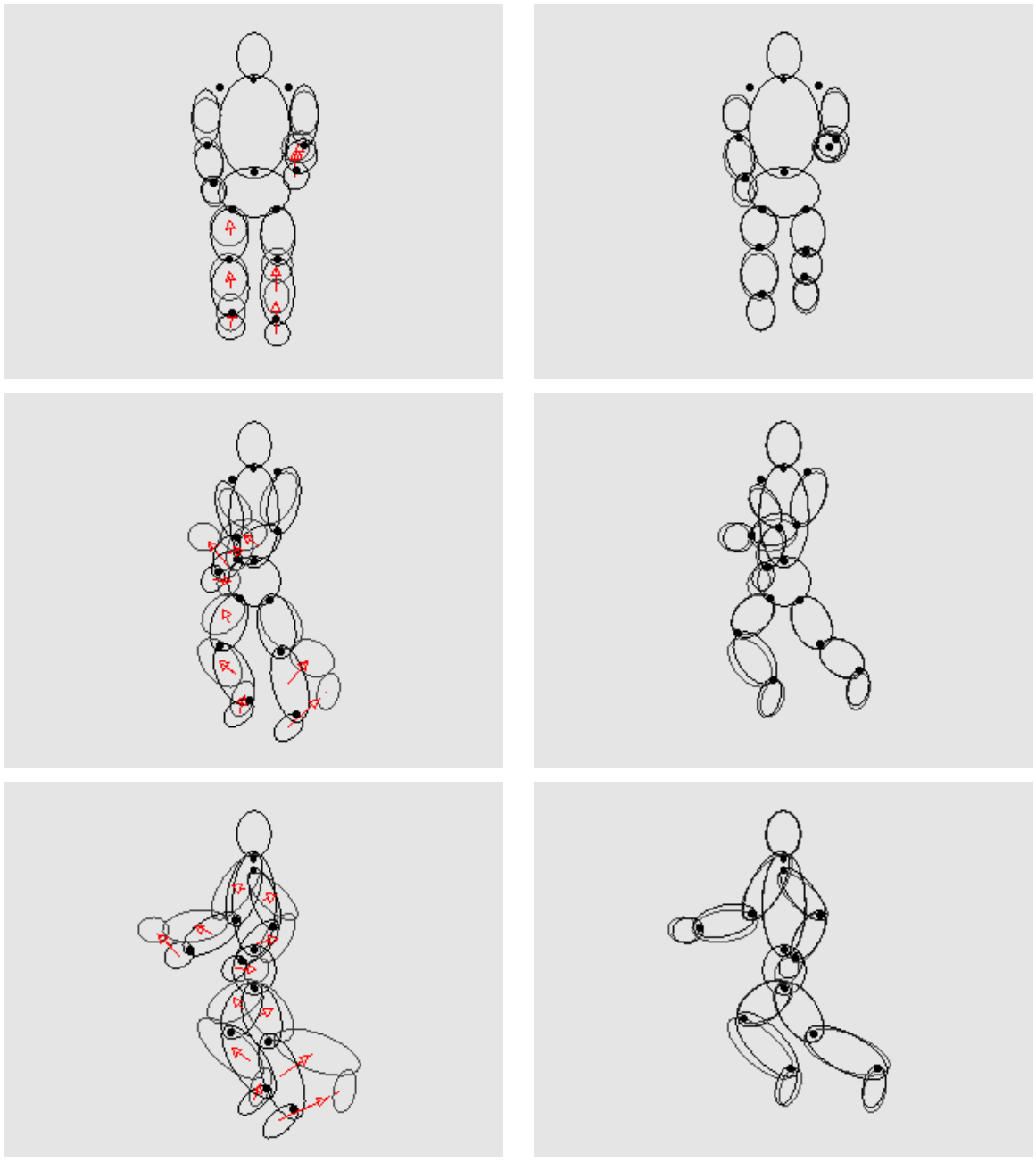


Figure 4.9: Full body human model static update #2. The left and right columns show the primary and intermediate body configurations before and after the static update.

4.1.4 Stanford/JPL Robotic Hand

In order to show that the modeling opportunities for rigid body systems are not limited to human bodies and traditional linkage chains, in this section we present a kinematic hand model. The modeled hand is actually robotic rather than human, and is known as the Stanford/JPL hand. It has been designed and used by researchers in robotic manipulation and control [9]. A robot hand is modeled because the mechanical joints have a more direct correspondence to the constraints supported by the model compiler, and because having only three fingers reduces the computational burden on the kinematic update. However, a five-finger hand patterned after a human hand could be designed in a very similar fashion. Such a model might have applications in remote control of robotic manipulators.

Displays 4.9–4.11 show the model specification for the hand. Figures 4.10 and 4.11 show sample hand configurations. The three finger design is quite versatile, demonstrating the ability to grasp objects of a wide variety of shapes and sizes. Note that the first joint of each finger rotates in a direction perpendicular to the second and third joints to allow a wider range of grasping positions. This behavior could not be modeled with planar kinematics, so the hand model is seen to fundamentally depend on the three-dimensional static update provided by the model compiler.

Figure 4.12 shows a sample static update computed with the hand model. It demonstrates a grasping motion such as might typically be used for small objects. Note that the range of motion in the grasping update is smaller than that seen in the other compiled model examples. The reason for this is that the static updates for the hand have in general demonstrated less stability than the static updates for the other compiled models. The specific reasons for this are still under investigation.

```
// Kinematic model of Stanford/JPL three-finger robotic hand
double fac = 1.25;  // Size factor between inner and outer pdfs

ellipsoidBody LowerFinger(double length, double radius, bool thumb) {
    innerLam(radius,radius,length);
    outerLam(fac*radius, fac*radius, fac*length);
    bvec lowerJoint {
        pos(0, 0, 0.6*length);
        orient(0, 1, 0);
    }
    bvec upperJoint {
        pos(0, 0, -0.8*length);
        if (thumb) {
            orient(-1, 0, 0);
        }
        else {
            orient(1, 0, 0);
        }
    }
}
ellipsoidBody MidFinger(double length, double radius, bool thumb) {
    innerLam(radius,radius,length);
    outerLam(fac*radius, fac*radius, fac*length);
    bvec lowerJoint {
        pos(0, 0, 0.8*length);
        if (thumb) {
            orient(-1, 0, 0);
        }
        else {
            orient(1, 0, 0);
        }
    }
    bvec upperJoint {
        pos(0, 0, -0.8*length);
        if (thumb) {
            orient(-1, 0, 0);
        }
        else {
            orient(1, 0, 0);
        }
    }
}
```

Display 4.9: Stanford/JPL robotic hand specification (part 1/3).

```
ellipsoidBody FingerTip(double length, double radius, bool thumb) {
    innerLam(radius,radius,length);
    outerLam(fac*radius, fac*radius, fac*length);
    bvec joint {
        pos(0, 0, 0.8*length);
        if (thumb) {
            orient(-1, 0, 0);
        }
        else {
            orient(1, 0, 0);
        }
    }
}
bvec origin {
    pos(0,-2,10);
    orient(0,1,0);
}

rigidBodySystem JPLHand {
    ellipsoidBody palm {
        double width = 0.7;
        double length = 1.0;
        double depth = 0.27;
        innerLam(length, depth, width);
        outerLam(fac*length, fac*depth, fac*width);
        bvec firstFinger {
            pos(-0.7*length, 0, -0.6*width);
            orient(0, 1, 0);
        }
        bvec secondFinger {
            pos(-0.7*length, 0, 0.6*width);
            orient(0, 1, 0);
        }
        bvec thumb {
            pos(0.2*length, 0.6*depth, 0);
            orient(1, 0, 0);
        }
        bvec wrist {
            pos(0.7*length, 0, 0);
            orient(0, 0, -1);
        }
    }
```

Display 4.10: Stanford/JPL robotic hand specification (part 2/3).

```
    ellipsoidBody arm {
        double radius = 0.45;
        double length = 0.8;
        innerLam(radius, length, radius);
        outerLam(fac*radius, fac*length, fac*radius);

        bvec inertialAnchor {
            pos(0, 0, 0);
            orient(0, 1, 0);
        }
        bvec wrist {
            pos(0, 0.7*length, 0);
            orient(0, 0, -1);
        }
    }

    LowerFinger lowFinger1(0.3, 0.25, false);
    LowerFinger lowFinger2(0.3, 0.25, false);
    LowerFinger lowThumb(0.3, 0.25, true);
    MidFinger midFinger1(0.5, 0.25, false);
    MidFinger midFinger2(0.5, 0.25, false);
    MidFinger midThumb(0.5, 0.25, true);
    FingerTip tip1(0.35, 0.2, false);
    FingerTip tip2(0.35, 0.2, false);
    FingerTip thumbTip(0.35, 0.2, true);

    constraint revolute(arm.wrist, palm.wrist, 45);
    constraint revolute(palm.firstFinger, lowFinger1.lowerJoint, 90);
    constraint revolute(palm.secondFinger, lowFinger2.lowerJoint, 90);
    constraint revolute(palm.thumb, lowThumb.lowerJoint, 90);
    constraint revolute(lowFinger1.upperJoint, midFinger1.lowerJoint, 10);
    constraint revolute(lowFinger2.upperJoint, midFinger2.lowerJoint, 10);
    constraint revolute(lowThumb.upperJoint, midThumb.lowerJoint, 20);
    constraint revolute(midFinger1.upperJoint, tip1.joint, 10);
    constraint revolute(midFinger2.upperJoint, tip2.joint, 10);
    constraint revolute(midThumb.upperJoint, thumbTip.joint, 30);
}

constraint revolute(JPLHand.arm.inertialAnchor, origin);
```

Display 4.11: Stanford/JPL robotic hand specification (part 3/3).

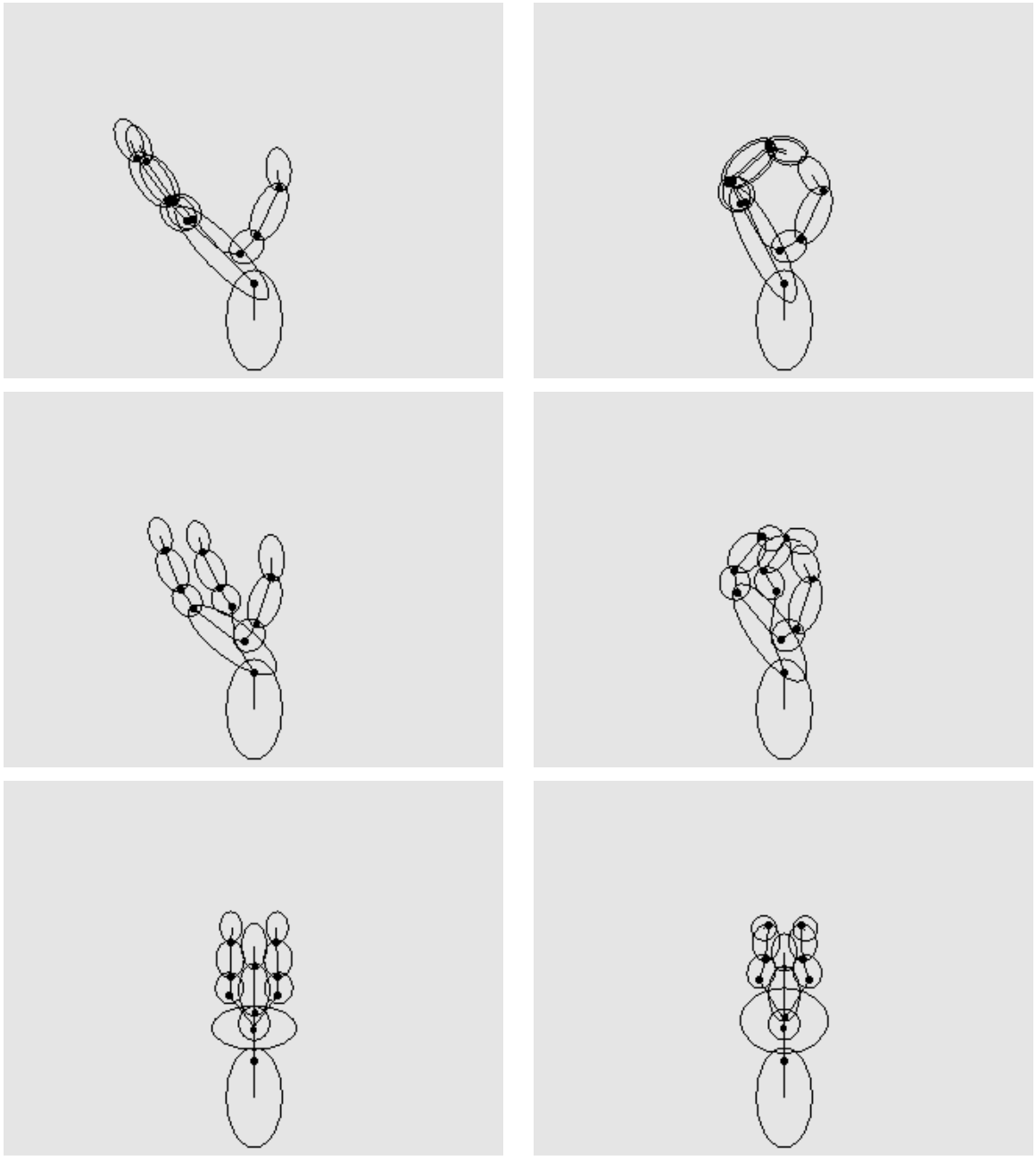


Figure 4.10: Compiled model of Stanford/JPL robotic hand. The left and right columns show the hand at the default rest position and in a grasping posture.

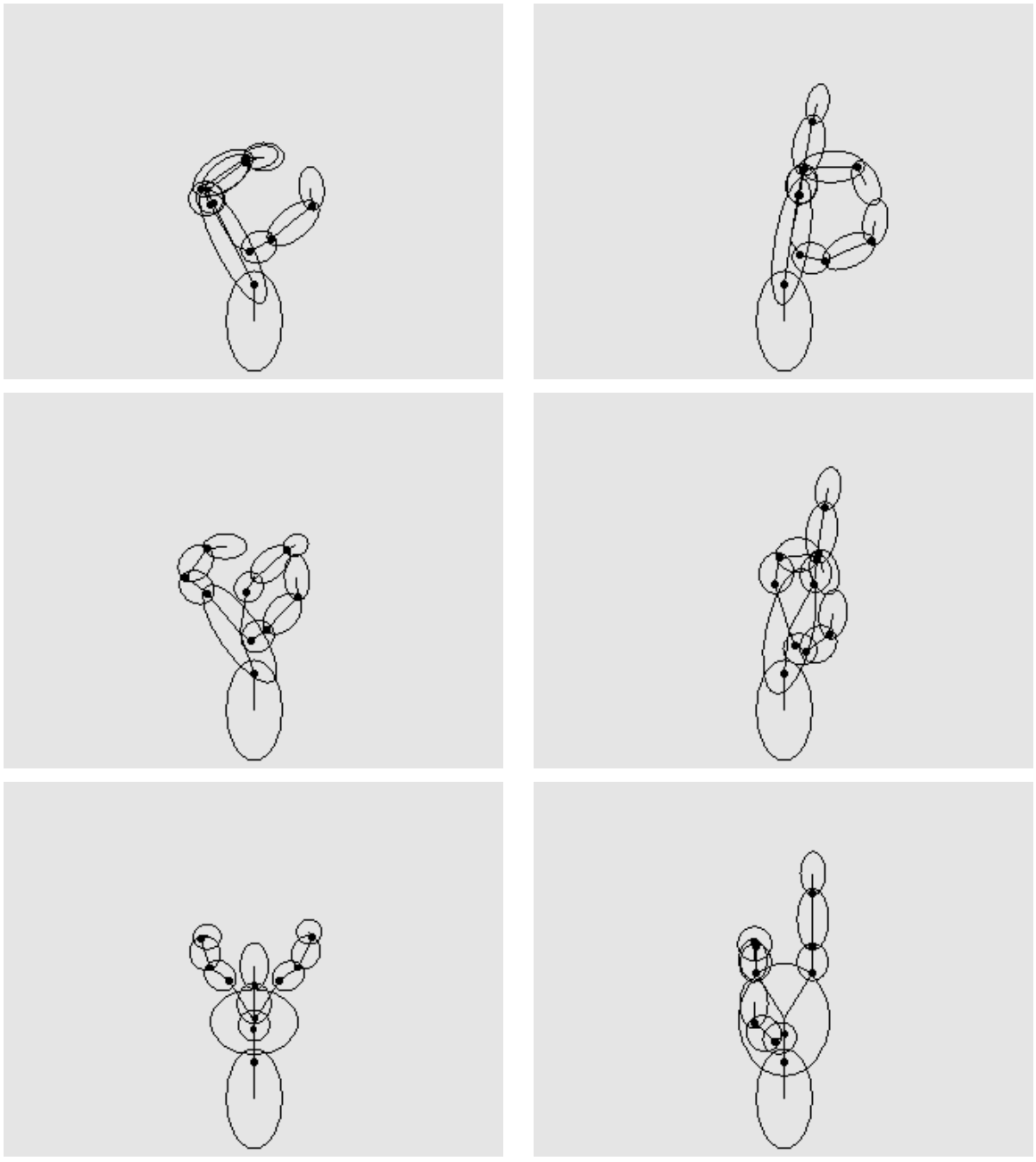


Figure 4.11: More complex poses of the hand model: a widely spaced grasping position and an OK sign.

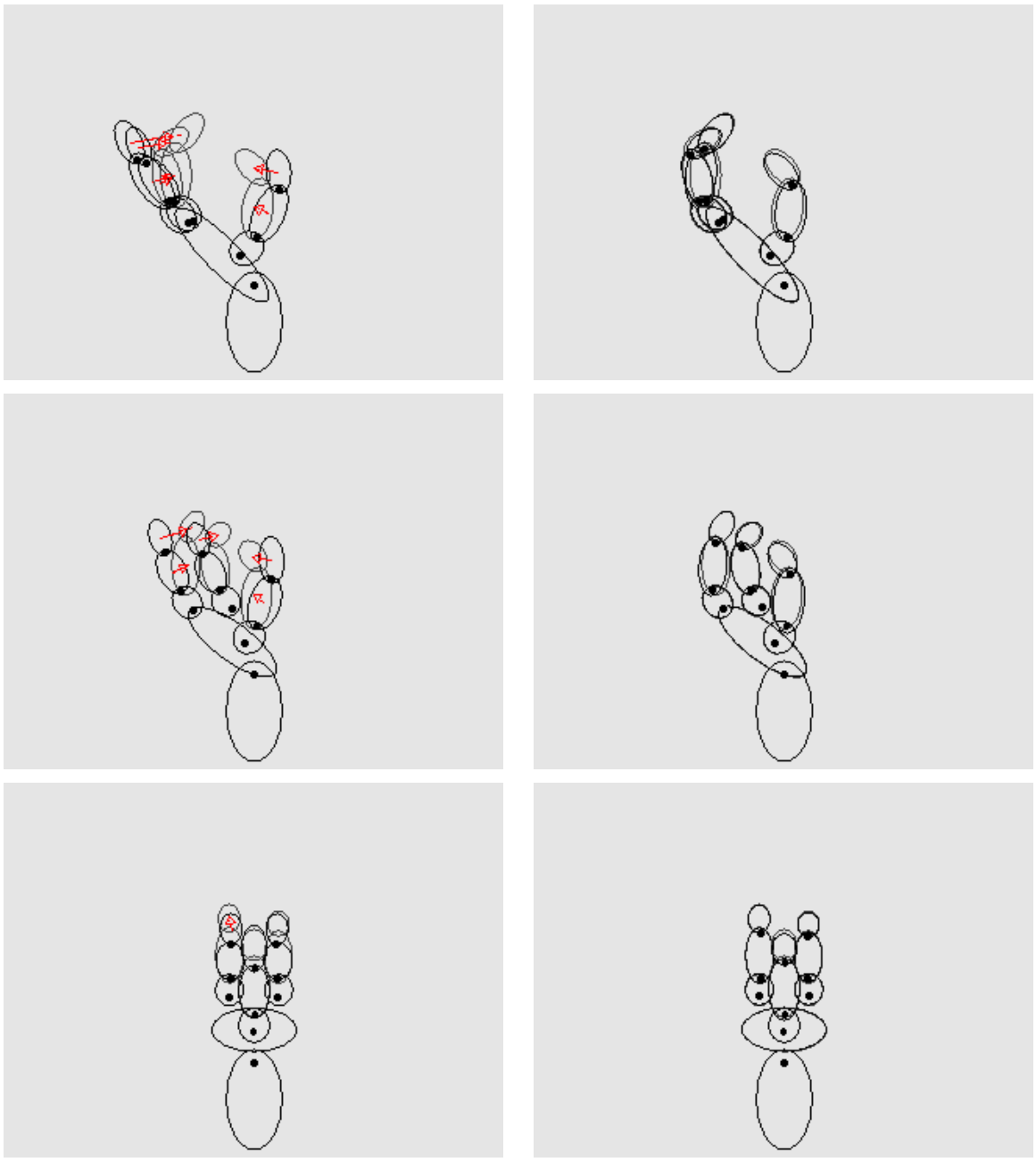


Figure 4.12: Stanford/JPL hand model static update. The left and right columns show the primary and intermediate body configurations before and after the static update.

4.2 Recommendations

This document has described the design, implementation, and preliminary analysis of a kinematic model compiler for the estimation of articulated motion from video. The model compiler concept, in the sophistication presented in this document, is a virtually unexplored research area. In addition, the expection constrained maximization, or ECM, algorithm that the compiler implements has only recently been introduced, and its properties are therefore far from fully understood. As a consequence of this novelty, a wide range of interesting research questions have become apparent over the course of this project.

Currently, the most pressing concern is to flesh out the details of the ECM software framework [3] to allow full three-dimensional motion estimation from captured video data. This involves restructuring the M-step, as discussed in §4.1. Once this is done, experimental trials should be conducted to validate the applicability of three-dimensional kinematics to the ECM. Experience with earlier versions of the ECM has shown that these experiments will undoubtedly expose a variety of new research questions. Ideally, video captured from multiple camera angles should be used to examine kinematic structures that take full advantage of a non-planar kinematic representation. At the same time, a more quantitative analysis of the ECM estimates than is presented in this document should also be undertaken.

The next obvious research direction for the compiler is to expand the supported set of kinematic constraints. Prismatic joints, which have a single translational degree of freedom, and spherical joints, which have a full three rotational degrees of freedom, are the most likely candidates. These new joint types may raise novel kinematics problems like that discussed for revolute joints in §2.2.4. In addition, the global arrangement of kinematic constraints should be diversified to allow loops in the kinematic chains. This will make the coordinate partitioning problem more difficult. The introduction of both prismatic joints and loops would permit the investigation of a much wider variety of kinematic models with the latest ECM algorithms, such as the lip models designed by Kelly [6].

Another interesting modification would be to reintroduce the planar kinematics used in

the prior ECM implementation [3]. For models where planar kinematics are sufficient, the complexity of the virtual work equations may be reduced by several orders of magnitude. The primary task here is to identify a consistent criterion by which three-dimensional models may be reduced to the planar case under some circumstances. An especially intriguing possibility is the integration of three-dimensional and planar kinematics in different regions of the same model.

The model specification language is also open to investigation. Obviously, a variety of other useful C++ language features could be translated to the specification grammar. In addition, it would be useful to add other blocks to the language which affect other aspects of the estimation process besides the model structure. For example, parameters could be passed to the different algorithm components based on a priori knowledge of the video characteristics. Another possibility is the introduction of interface blocks which allow the user to specify a custom set of controls for modifying the model's parameters in the graphical interface.

Finally, and perhaps most importantly, the model compiler provides a framework for investigating new and diverse variants of the ECM, or even entirely different estimation techniques. Once these new concepts are implemented in the compiler framework, they may then be applied and tested on a wide variety of models. Ultimately, the model compiler's must valuable contribution is to facilitate the translation of existing research to new problem domains as they arise.

Bibliography

[1] Headingtion, M.R. and D.D. Riley, *Data Abstraction and Structures Using C++*. Jones and Bartlett, 1997.

[2] Hunter, E.A., P.H. Kelly and R. Jain, "Estimation of Articulated Motion using Kinematically Constrained Mixture Densities," *Proceedings IEEE Nonrigid and Articulated Motion Workshop*, San Juan, Puerto Rico, 16 June 1997 (pp. 10–17).

[3] Hunter, E.A., "Visual Estimation of Articulated Motion using the Expectation Constrained Maximization Algorithm," Ph.D. Dissertation, Electrical and Computer Engineering (Intelligent Systems, Robotics and Control), University of California, San Diego, April 1999.

[4] Kay, S.M., *Fundamentals of Statistical Signal Processing: Estimation Theory*. Prentice-Hall, 1993.

[5] Kelly, P.H., E.A. Hunter, K. Kreutz-Delgado and R. Jain, "Lip Posture Estimation using Kinematically Constrained Mixture Models," *Proceedings British Machine Vision Conference*, Southampton, UK, September 1998 (pp. 74–83).

[6] Kelly, P.H., "Mouth Posture Estimation using a Geometric Model of the Lips," Ph.D. Dissertation, Electrical and Computer Engineering (Electronic Circuits and Systems), University of California, San Diego, December 1998.

[7] Koivo, A.J., *Fundamentals for Control of Robotic Manipulators*. John Wiley and Sons, 1989.

[8] Levine, J.R., T. Mason, and D. Brown, *lex & yacc*. O'Reilly & Associates, 1992.

[9] Mason, M.T. and J.K. Salisbury, Jr., *Robot Hands and the Mechanics of Manipulation*. Massachusetts Institute of Technology, 1985.

[10] Press, W.H., S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C*. Cambridge, 1992.

[11] Shabana, A.A., *Dynamics of Multibody Systems*. Cambridge, 1998.

[12] Strang, Gilbert, *Linear Algebra and its Applications*. Harcourt Brace Jovanovich, 1988.

[13] Stroustrup, B., *The C++ Programming Language*. Addison-Wesley, 1997.

[14] Sudderth, E., E. Hunter, K. Kreutz-Delgado, P. Kelly, and R. Jain, "Adaptive Video Segmentation: Theory and Real-Time Implementation," *Proceedings of the 1998 Image Understanding Workshop*, Monterey, CA, November 1998 (pp. 177–181).