Programming Language Support for Geometric Computations

By

Mark Gordon Segal

B.A. (Harvard College) 1982
M.S. (University of California) 1986

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA at BERKELEY

Approved: Carlo H. Séquin ..... April 7, 1989
Chair ..... Date
Alan D. Rosenthal ..... 19 APR 89
W. Kahan ..... April 28, 1989

****

Abstract

Implementing an algorithm that computes the values of geometric quantities has required their translation into explicit coordinate arrays comprehensible in an appropriate computer language. The simplicity and intuition inherent in the mathematical expression of geometric objects is lost in a profusion of coordinates rendering programs difficult to implement, understand, and debug.

This problem is overcome by providing a set of programming language constructs based on geometric objects rather than their coordinate representations. These objects and operations on them are adapted from standard mathematical usage. The actual coordinate computations are hidden and carried out automatically, resulting in a concise program looking much like its mathematical counterpart.

These constructs are suitable for programming a range of geometric computations. Examples show their use in the calculations of linear geometry, computer graphics, finite element analysis, relativistic kinematics and robotics.

A compiler for the language constructs can check geometric expression syntax and semantics. It can also recognize special operators and operands (diagonal or sparse matrices, for instance) so that unnecessary and possibly inaccurate floating-point operations are automatically avoided. Calls to a runtime package actually carry out the operations on representations of geometric objects. The issues involved in writing such a compiler and its associated runtime package are discussed. A preliminary implementation is presented and the prospects for its generalization assessed.

Chapter 1

Introduction

Solutions of geometric problems on a computer have been hampered by computer language constructs inadequate to support operations on geometric data types. In particular, current languages oblige us to express a geometric object (a vector or linear transformation, for instance) as an array or matrix of coordinates. Consequently, the code to solve a geometric problem is obscured by the large number of index manipulations required to access these coordinates.

I have developed a set of types and operators for geometric objects that make most coordinate manipulations unnecessary. These constructs can be added to an appropriate computer language. Coordinate computations are carried out automatically, letting the programmer concentrate on the geometrical issues. The formalism also provides a simple but powerful set of geometric operations in a form closely resembling standard mathematical usage.

1.1 Overview

This thesis begins with a couple of short examples to indicate the need for a clear, coordinate-free notation and corresponding programming language constructs. Other approaches to solving geometric problems on a computer are also discussed. Chapter 2 develops the notation that becomes the basis for a set of

geometric types and operators. Chapter 3 presents these types and gives a syntax for declaring and operating on simple geometric objects. Chapter 4 provides a series of diverse examples that show the usefulness of the language constructs in writing simple, clear prorgams to solve a range of problems. Chapter 5 discusses the issues involved in implementing the language constructs both in general and as an extension to an existing compiler. Chapter 6 describes a preliminary compiler implemented as a pre-processor for C. Chapter 7 concludes with some remarks on the usefulness of the work and whether or not it should be extended so as to produce a viable geometric programming system.

1.2 Motivation

As an example, consider the problem of finding the point where a line intersects a plane not parallel to the line. Assume the line is specified by a point $\mathbf{p}$ on it and a unit direction vector $\mathbf{v}$ parallel to it, and the plane by a point $\mathbf{q}$ on it and a unit vector $\mathbf{n}$ normal to it (Figure 1.1). This problem is most naturally solved by substituting the parametric equation of the line ($\mathbf{x} = \mathbf{p} + t\mathbf{v}$) into the implicit equation of the plane ($\mathbf{n} \cdot (\mathbf{x} - \mathbf{q}) = 0$) to get

$$t = \frac{\mathbf{n} \cdot (\mathbf{q} - \mathbf{p})}{\mathbf{n} \cdot \mathbf{v}},$$

so that the intersection point is

$$\mathbf{p} + \frac{\mathbf{n} \cdot (\mathbf{q} - \mathbf{p})}{\mathbf{n} \cdot \mathbf{v}} \mathbf{v}.$$

In a language like C, let us declare data structures to hold these quantities,

```
#define N 3
structure line { real p[N]; real v[N]; } ;
structure plane { real q[N]; real n[N]; } ;
```

a traditional program to compute $\mathbf{x}$ might look as follows

```
function intersect(l,m) returns array[N];
```

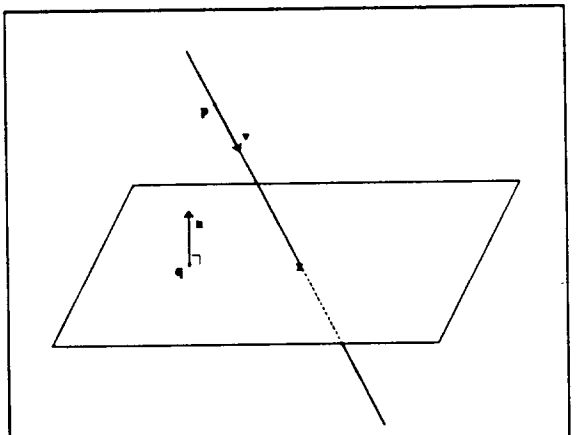


Figure 1.1. Geometric quantities associated with computing the intersection of a line with a plane.

```
l is line[p,v]; m is plane[q,n];
begin
    return add(p, scalarMult(dotp(n, diff(q,p))/dotp(n, v), v));
end intersect
```

I have assumed that the routines `diff`, `dotp`, `add` and `scalarMult` to manipulate arrays have already been written.

This program, although simple, has many problems that become especially cumbersome in more complex applications. One of the worst is that the program looks little like the formula from which it arose. The functional prefix notation obscures the clarity of the mathematical infix notation. Another difficulty is that the program works for a fixed `N` only; a new subroutine must be made for each number of dimensions, yet the solution to the line-hyperplane intersection problem looks the same in any number of dimensions.

Even greater problems come about from representing the geometric objects as arrays of numbers. Many applications use a multitude of coordinate systems so

that a simple array of coordinates is really insufficient without an indication of the pertinent coordinate system. Further, the plane normal vector does not transform in the same way as the line's direction vector under a change of coordinates. In any case, coordinate conversions may be required that would obscure the geometry of the problem even further under a barrage of matrix multiplications.

Arrays also obscure the distinction between points and vectors. A point is a location in space, while a vector is a displacement between points. It makes no sense to add two points, although their difference can be interpreted as the displacement from the second to the first. The distinction between points and vectors is essential in some problems, and so should be made clear in a program solving such a problem.

These problems are overcome by providing constructs that account for the definition of linear and affine objects and operations upon them. Calculations are written in coordinate-free notation; the compiler or run-time package chooses the coordinate system or systems for evaluation.

With these constructs, the data structures used to store the parameters for the intersection problem are similar to those for the previous program

```
#define N 3
structure line { point p; vector v; } ;
structure plane { point q; vector n; } ;
```

but the distinction between points and vectors is made explicit. The subroutine itself looks like:

```
function intersect(l,m) returns point;
l is line[p,v] ; m is plane[q,n] ;
begin
  return p + (n†*(q-p)) / ( n†* v) * v ;
end intersect
```

Nothing need be said about different coordinate systems or transformation rules for normal vectors. Any necessary computation on coordinates representing the various geometric objects are hidden from the programmer, making the

resulting program look very much like the formula itself.

A more complex example comes from the study of mechanical systems. If the kinetic energy of a system is quadratic in the system's velocities, Lagrange's equations provide a way to obtain its dynamical equations[1] in the form $f = Ma$, where f is an n-tuple representing forces, M is a symmetric matrix (the inertia tensor), and a is an n-tuple of accelerations. If T is the kinetic energy of any conservative[1] physical system and F the n-tuple of applied forces in compatible units, Lagrange's equations are

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{q}}\right) - \frac{\partial T}{\partial q} = F^{\mathrm{T}},$$

where $q = (q_1, \ldots, q_n)^{\mathrm{T}}$ and $\dot{q} = (\dot{q}_1, \ldots, \dot{q}_n)^{\mathrm{T}}$ describe the system's position and velocity, respectively. For mechanical systems in which T is quadratic in the velocities, the derivatives at a point in *configuration space* can be found symbolically. In this case the first term is of the form $Ma^{\mathrm{T}} + b^{\mathrm{T}}$ (b is a function of position and velocity), and the second term is an n-tuple c, a function of position and velocity. Rearrangement of Lagrange's equations yields $Ma = F - b - c$, or $f = Ma$ with $f = F - b - c$.

Finding the path of a mechanical system through configuration space can be viewed as a geometric problem. As a consequence of Lagrange's equations, the path that the configuration parameters (degrees of freedom) follow through configuration space is a shortest path on a surface of constant energy. The spaces of interest in this problem are the configuration space, the tangent space to the configuration space (velocity space), and the three dimensional space in which energies, velocities, and accelerations are measured and computed. Various maps relate configuration positions and velocities to their three dimensional counterparts. These maps along with their derivatives appear in Lagrange's equations. Once the appropriate maps have been found, the value of the inertia tensor M can be computed.

The formalism enables the programmer to refer to the spaces and maps di-

rectly, without reference to a profusion of indistinguishable coordinate arrays. A program that uses the formalism in solving Lagrange's equations displays geometric calculations in succinct notation similar to standard mathematical usage (such a program appears in a later chapter).

1.3 Other Approaches

Most of the previous work in this area can be divided into two categories. The first category is languages or systems for handling arrays. APL (Array Processing Language) is a programming language, one of the major objectives of which is to simplify constructing and operating on multi-dimensional arrays[2]. While APL provides terse specification of array operations, including such common operations as matrix addition and multiplication, it does not make any provision for geometric objects such as points and vectors. APL is inadequate for supporting these objects because there is no means to associate a coordinate frame with an array; expressions are thus coordinate dependent. The same deficiencies plague other systems designed primarily for handling arrays and matrices: examples include TORRIX[3] and MATLAB[4].

The second category consists of systems to handle operations on specific types of high-level geometric objects. Examples of such systems include: graphics languages such as PostScript[5] and graphics standards[6], robot programming languages that allow a user to position three-dimensional rigid objects and specify robot arm manipulations of those objects[7][8], and even geometric calculators based on homogeneous coordinates that allow specification of and calculations on points, lines, and planes[9]. The problem with these systems is that they are too specific. There is no way to generalize their constructs to higher dimensions (or even lower dimensions) or to other related geometric problems. The methods to be presented here could be used to build such systems.

A different approach is taken in symbolic systems designed to handle geometric

objects[10][11]. Such a system uses the language of differential geometry to build and manipulate such objects as tensors, forms, and connections. Manipulations include a number of symbolic computations of interest to differential geometers. This is in contrast to programming language constructs designed to aid in the determination of numerical values for geometric objects. A symbolic system might be used to produce code using the language constructs that, once the desired symbolic manipulations have been performed, finds numerical values for the resulting objects given differing input parameters.

There are several possible options for implementing the language constructs. One is to simply provide a set of routines that handle geometric objects in coordinate-free form. Another is to use a language providing object-oriented constructs to incorporate geometric objects and operations (candidate languages for this approach include Ada[12], C++[13], or LISP-based LOOPS[14]). The most comprehensive possibility modifies an existing compiler to recognize geometric constructs so that optimization and error checking can be done at compile time. These options will be taken up in Chapter 5, but before discussing their merits, we must decide what geometric objects and operations are pertinent, how they are to be represented, and how they are likely to be used.

Chapter 2

Geometrical Foundations

The purpose of this chapter is to familiarize the reader with the mathematical concepts and notation used in later chapters and to provide some simple examples with which to gauge the efficacy of the programming constructs to be introduced. It is not meant to be a comprehensive introduction to linear geometry.

2.1 Geometry

Mathematicians have devoted much effort to develop succinct notations for the precise expression of geometric concepts. The inability of programmers to translate this notation to programming language constructs has accounted for considerable difficulty when writing a program to solve a geometric problem.

To alleviate this difficulty, I review several elements of linear geometry in a consistent notation suitable for adaptation to an appropriate programming language. The resulting programming constructs also aid the computer solution of problems in differential geometry because differential geometry is built on linear approximations.

2.2 Linear and affine spaces, mappings

One result of the usual way of programming a geometric problem is a confusion between points and vectors. Both are typically implemented as coordinate arrays in which the number of entries is equal to the dimension of the modeled space. This superficial similarity may mask the differing effects of coordinate transformations on points and vectors.

In fact, a point represents a location in some space, while a vector represents a displacement between two points[15]. An immediate consequence is that a translation of a space affects its points, but not the vectors between points. Further, the sum of two vectors makes sense as one displacement followed by another; the resulting total displacement is another vector. However, the sum of two points, two locations in space, is not a geometric quantity. While it is possible to sum the *coordinates* of a pair of points, the result depends on the choice of origin and thus on more than the points themselves.

Similarly, it is nonsense to negate a point, multiply a point by a scalar or subtract a point from a vector. However, we may think of the addition of a point to a vector as a displacement of the point by the vector, resulting in a displaced point. We can also interpret the difference of two points as the displacement between them.

These ideas are formalized by recalling the definition of a *linear space*[15] and defining an *affine space*, A, as:

(1) For every pair of elements $\mathbf{P}, \mathbf{Q} \in A$, there is assigned a vector of a linear space L called the *displacement* between $\mathbf{P}$ and $\mathbf{Q}$ and denoted $\mathbf{PQ}$ (or $\mathbf{Q} - \mathbf{P}$).

(2) For each element $\mathbf{P} \in A$ and each displacement $\mathbf{v} \in \mathrm{L}$ there is exactly one element $\mathbf{Q}$ in A such that $\mathbf{v} = \mathbf{PQ}$.

(3) If $\mathbf{P}$, $\mathbf{Q}$ and $\mathbf{R}$ are arbitrary elements in A, then $\mathbf{PQ} + \mathbf{QR} = \mathbf{PR}$.

Functions can be defined that operate on the elements of linear spaces or affine

spaces. The *linear map* operates on a linear space element and produces another linear space element (possibly in a different space); recall that any linear map M_L satisfies $M_L(\alpha\mathbf{u}+\beta\mathbf{v}) = \alpha M_L(\mathbf{u}) + \beta M_L(\mathbf{v})$ where α and β are scalars. An *affine map* takes linear or affine space elements to linear or affine space elements. An affine map M_A with a linear space as range satisfies

$$M_A(\mathbf{v}) = M_L(\mathbf{v}) + \mathbf{O}'.$$

If O' is a vector, then M_A takes vectors to vectors; if O' is a point, then M_A takes vectors to points.

An affine map with an affine space as range is of the form

$$M_A(\mathbf{P}) = M_L(\mathbf{OP}) + \mathbf{O}',$$

where M_L is a linear mapping, $\mathbf{O}$ is an element in the domain of M_A, and $\mathbf{O}'$ is an element in its range. The range can either be a linear space (in which case M_A takes points to vectors) or an affine space (in which case M_A takes points to points). If $\mathbf{O} = \mathbf{O}'$, M_A is often (inappropriately: an origin must be specified) called a *pure linear operator*. If M_L is the identity, then M_A is called a *pure translation* by (or through) $\mathbf{OO}'$.

2.3 Bases

A (finite-dimensional) linear space is equivalent *but not identical* to the space of coordinate n-tuples where n is equal to the dimension of the linear space. These two spaces are said to be *isomorphic*[16]. A bijective (one-to-one and onto) linear mapping relates any two isomorphic linear spaces. Once an origin has been picked, the affine space of points can also be described by some n-tuple.

The relationship of vectors or points to n-tuples is specified with a *coordinate basis* or *affine coordinate basis*, respectively. A coordinate basis (or *linear basis* or simply *basis*) consists of a set of linearly independent linear space elements. It can

be used to express any vector in terms of n coordinates. The vector is recovered from the coordinates by multiplying each of the basis vectors by the corresponding coordinate, and summing the results.

An affine basis consists of a basis and an origin. Once an origin has been determined, a point can also be represented as an n-tuple by adding the origin to the vector defined by the n-tuple in the linear basis.

Using the rules of matrix multiplication, if $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ is a basis for a vector space and $\mathbf{v}$ is a vector in that space, we write

$$\mathbf{v} = \begin{pmatrix} \mathbf{x}_1 \cdots \mathbf{x}_n \end{pmatrix} \underbrace{\begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix}}_{n-\text{tuple}} = \alpha_1 \mathbf{x}_1 + \ldots + \alpha_n \mathbf{x}_n,$$

where each of the α_i is a scalar. The same formula specifies a point if the vector represents a displacement from a fixed origin.

The n-tuples form a convenient way to represent vectors and points in a computer. However, the basis belongs to the modeled space, and only the modeler knows the relationship between the basis vectors and their representation in the computer. For an affine space, only the modeler knows the location of the origin relative to which the affine basis specifies a point's coordinates. The basis and affine basis that take n-tuples to the modeled vectors and points are called *intangible*. Other bases may be defined on the modeled space, but they can all be related to the intangible basis (and thus to one another, if none is singular).

The basis is a linear or affine operator taking n-tuples into the space of vectors or points. Since there are many distinct bases for a given linear space, there are many ways of expressing a modeled vector as an n-tuple. But the modeled vector (or point) is the same regardless of the basis in which it is expressed.

In robotics, a different basis is introduced for each joint in a robot arm (Figure 2.1). Figure 2.2 shows the situation in graphics where a modeled object defined in terms of a world coordinate basis is viewed on a computer screen defined in

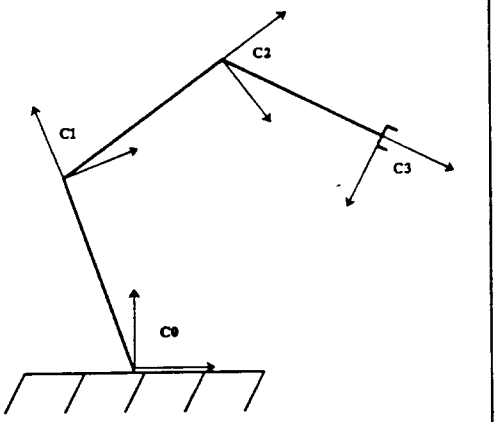


Figure 2.1. A jointed robot arm, with a coordinate frame at each joint.

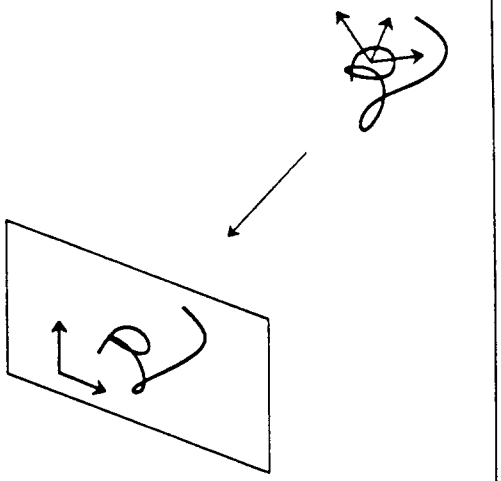


Figure 2.2. Projecting a 3-D graphical object onto a 2-D screen defined in terms of an eye coordinate frame.

terms of an eye coordinate basis. In general, problems in geometry are often efficiently solved by choosing advantageous bases in which to calculate. We would like a convenient means for specifying an object in any one of these bases without having to remember later in which basis the object was last specified.

2.4 Change of Coordinates

To distinguish between n-tuples and vectors, $\bar{\mathbf{v}}$, $\bar{\mathbf{u}}$ denote n-tuples, while $\mathbf{v}$, $\mathbf{u}$ denote the corresponding vectors. Similarly, $\bar{\mathbf{P}}$ and $\bar{\mathbf{Q}}$ denote n-tuples whose corresponding points are $\mathbf{P}$ and $\mathbf{Q}$.

Suppose we are given two n-tuples

$$\vec{\mathbf{v}} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \quad \text{and} \quad \vec{\mathbf{v}}' = \begin{pmatrix} v_1' \\ \vdots \\ v_n' \end{pmatrix}$$

describing the same vector $\mathbf{v}$ in coordinate bases C and C', respectively. The relation between these two n-tuples is given by a matrix of scalars

$$\vec{\mathbf{v}}' = M\vec{\mathbf{v}}, \quad M = \begin{pmatrix} m_{11} & \dots & m_{1n} \\ \vdots & \ddots & \vdots \\ m_{n1} & \dots & m_{nn} \end{pmatrix}.$$

This means that if $C = (\mathbf{c}_1, \dots, \mathbf{c}_n)$ and $C' = (\mathbf{c}'_1, \dots, \mathbf{c}'_n)$, then $\mathbf{v}$ can be written

$$\mathbf{v} \quad = \quad \begin{pmatrix} & & \\ \mathbf{c}'_1 & \dots & \mathbf{c}'_n \\ & & \end{pmatrix} M\vec{\mathbf{v}} \quad = \quad \begin{pmatrix} & & \\ \mathbf{c}_1 & \dots & \mathbf{c}_n \\ & & \end{pmatrix} \vec{\mathbf{v}}.$$

Since the same M works for any $\mathbf{v}$, M also relates C and C', that is,

$$C'M = C.$$

The rows of M determine the relationship among corresponding pairs of n-tuples, while its columns determine the relationship of the two sets of basis vectors. M is often called the "matrix of C with respect to C'."

C and C' map an n-tuple onto a vector, so C^{-1} and C'^{-1} map a vector onto an n-tuple. That is, C^{-1} computes the coordinates of a vector in coordinate basis C. Thus we may write

$$M = C'^{-1}C.$$

This formula gives us a useful interpretation of the matrix M. M first takes coordinates in C and produces the corresponding vector. It then extracts the coordinates of this vector in coordinate basis C'.

If C_f is an *affine coordinate basis* with C as basis and $\mathbf{O}_C$ as origin, then $\mathbf{P} = C_f(\bar{\mathbf{P}}) = C\bar{\mathbf{P}} + \mathbf{O}_C$ so that $C_f^{-1}(\mathbf{P}) = C^{-1}(\mathbf{P} - \mathbf{O}_C)$. An affine coordinate basis will also be referred to as a *coordinate frame*. If C'_f is another affine basis with C' as basis and $\mathbf{O}_{C'}$ as origin, then the map taking a point's coordinates in C_f to coordinates in C'_f is

$$M_f(\bar{\mathbf{P}}) = C'^{-1}_f C_f(\bar{\mathbf{P}})$$

or

$$M_f(\bar{\mathbf{P}}) = C'^{-1}C\bar{\mathbf{P}} + C'^{-1}(\mathbf{O}_C - \mathbf{O}_{C'}) = C'^{-1}C(\bar{\mathbf{P}} + C^{-1}(\mathbf{O}_C - \mathbf{O}_{C'})).$$

Thus M_f is an affine operator taking n-tuples to n-tuples with an offset in its domain or its range (or possibly both; the offset may be distributed between domain and range).

In computer graphics, objects are typically arranged in a hierarchy with each object made up of a set of pre-existing objects. Each pre-existing object is placed by giving the relationship of its coordinate basis to that of the object of which it is a part. If C'_f is the embedded object's basis and C_f is the embedding object's basis, the relationship is given by M_f.

If X is a linear transformation taking vectors to vectors, then we may write $\mathbf{u} = X\mathbf{v}$ for appropriate vectors $\mathbf{u}$ and $\mathbf{v}$. Extracting coordinates in C on both sides of the equation and inserting the identity transformation as $C'C'^{-1}$ between X and $\mathbf{v}$ gives the coordinate representation of vector transformation X:

$$C^{-1}\mathbf{u} = (C^{-1}XC')C'^{-1}\mathbf{v}.$$

That is, if $\vec{\mathbf{u}}$ represents $\mathbf{u}$ in C and $\vec{\mathbf{v}}$ represents $\mathbf{v}$ in C', then $M = C^{-1}XC'$ is the matrix representing X in these coordinate bases. Rewriting X in terms of M gives

$$X = CMC'^{-1}.$$

This formula can be used to interpret X as taking each vector of C' to a corresponding vector of CM.

Just as C is a basis for a vector space, C^{-1} is also a basis. The operation of C^{-1} may be expressed as

$$\vec{\mathbf{v}} = C^{-1}\mathbf{v} = \begin{pmatrix} \mathbf{c}_1^* \\ \vdots \\ \mathbf{c}_n^* \end{pmatrix} \mathbf{v}$$

so that each *row* of C^{-1} operates on $\mathbf{v}$ to produce a single coordinate. Each row in C^{-1} is thus a *linear functional* or *covector*; it maps a vector onto a real number.

The rows of C^{-1} constitute a basis on the space of covectors L^*, *dual* to the linear space L. Such a basis of coordinate extractors is called a *coordinate system.*

Thus there are three kinds of linear space of the same dimension with which we are concerned: the space of n-dimensional vectors, the space of linear functionals on n-dimensional vectors, and the space of n-tuples (which can be used to represent either of these). So that the notation of matrix multiplication can be employed when working with coordinates, the coordinates of an n-tuple representing a vector are written in a column, while those of an n-tuple representing a covector are arranged in a row. This distinction adds a fourth kind of linear space: the row n-tuples.

The affine analog for a coordinate system is an affine coordinate system:

$$\bar{\mathbf{P}} = C_f^{-1}(\mathbf{P}) = C^{-1}(\mathbf{P} - \mathbf{O}) = \begin{pmatrix} \mathbf{c}_1^* \\ \vdots \\ \mathbf{c}_n^* \end{pmatrix} (\mathbf{P} - \mathbf{O}).$$

In this case $\mathbf{c}_{f_i}^*(\mathbf{P}) \equiv \mathbf{c}_i^*(\mathbf{P} - \mathbf{O})$ is called an *affine functional* with offset in its domain.

2.5 Inner Products; Duality

Since L (spanned by C) and L^* (spanned by C^{-1}) are both linear spaces of the same dimension, there are many isomorphisms that relate them. One such isomorphism is determined by the *inner product* defined on the linear space L. The inner product is a bilinear functional B with these properties: for $\mathbf{v}, \mathbf{u} \in L$,

(1) Symmetry: $B(\mathbf{v}, \mathbf{u}) = B(\mathbf{u}, \mathbf{v})$.

(2) Positive definiteness: $B(\mathbf{v}, \mathbf{v}) \geq 0$ with equality if and only if $\mathbf{v}$ is the zero vector.

Usually $B(\mathbf{u}, \mathbf{v})$ is written $\langle \mathbf{u}, \mathbf{v} \rangle$.

If C is a basis and $\mathbf{u} = C\vec{\mathbf{u}}$ and $\mathbf{v} = C\vec{\mathbf{v}}$ are expressed, then the *standard* inner product in C is given, in coordinates, by $\vec{\mathbf{v}}^{\mathrm{T}}\vec{\mathbf{u}}$. This formula is not valid in

all coordinate systems; if the change of coordinate matrix $M = C^*C'$ relates the primed n-tuples to the unprimed ones, then

$$(\mathbf{v},\mathbf{u}) \quad = \quad \vec{\mathbf{v}}^{\mathrm{T}}\vec{\mathbf{u}} \quad = \quad \bar{\mathbf{v}}'(MM^{\mathrm{T}})^{-1}\bar{\mathbf{u}}'.$$

If M is an orthogonal matrix ($M^{\mathrm{T}}M = MM^{\mathrm{T}} = I$), then the inner product is the same if computed in either C or C'. Any basis in which the inner product is standard is called *orthonormal.*

The interpretation of $\mathbf{v}^*$, the dual of $\mathbf{v}$, depends on a chosen coordinate system. This preferred system is usually made to coincide with an orthogonal system, and we define $\mathbf{v}^*$ by

$$\mathbf{v}^*\mathbf{u} = \langle \mathbf{v},\mathbf{u}\rangle \quad \text{for all } \mathbf{u}.$$

Another way of specifying an inner product is to give its matrix in some basis so that

$$\langle \mathbf{v},\mathbf{u}\rangle = (C^{-1}\mathbf{v})^T N(C^{-1}\mathbf{u})$$

where N is a positive definite symmetric matrix. From this formula,

$$\mathbf{v}^* = (C^{-1}\mathbf{v})^T NC^{-1}.$$

Consequently, $\vec{\mathbf{v}}^* = \vec{\mathbf{v}}^{\mathrm{T}}$ in an orthonormal basis.

The inner product defines a norm by $\|\mathbf{v}\| \equiv \sqrt{\langle \mathbf{v},\mathbf{v}\rangle}$. It is also used to define the angle between two vectors: $\frac{\langle \mathbf{v},\mathbf{u}\rangle}{\|\mathbf{u}\|\|\mathbf{v}\|} = \cos(\alpha)$ where α is the angle between the unit vectors $\mathbf{v}$ and $\mathbf{u}$.

Any positive definite inner product is standard in some basis. However, sometimes it is useful to relax the positive definiteness criterion. As long as it is *non-degenerate*, then there is some coordinate basis in which an inner product is given by

$$\begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & -1 & & \\ & & & & \ddots & \\ & & & & & -1 \end{pmatrix}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{r}$$

(the off-diagonal elements are zero). r is called the *index* of the inner product. Inner products with index zero are positive definite; a linear space with a positive definite inner product is called *euclidean*. The *Minkowski* space, of interest in relativity, is a 4-dimensional space (three space dimensions plus one time dimension) with an inner product of index 1. The inner product of a vector with itself in this space is called its *invariant interval*, the sign of which determines if the vector is timelike, lightlike, or spacelike.

2.6 Subspaces

A subspace R of a linear space S is a linear space whose elements form a subset of S. An affine subspace R_a of an affine space S_a is a linear subspace passing through some point. In three-space, for example, a plane is a two-dimensional affine subspace, and a line is a one-dimensional subspace. A zero-dimensional subspace is a single point.

There are many ways to specify a particular subspace within a space. For instance, in euclidean three-space, a plane may be specified as a point and two spanning vectors, three points, a point and a normal vector, or a normal vector and a distance from an origin.

The most obvious subspace specification is a set of spanning vectors defining the linear subspace (with an origin if the subspace is affine). Thus, we may write

$$\mathbf{u} = \begin{pmatrix} & & \\ \mathbf{v}_1 & \dots & \mathbf{v}_m \\ & & \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{pmatrix}$$

for $\mathbf{u} \in R$. The m linearly independent vectors $\mathbf{v}_i$ span R in the total space S; the α_i are scalars. The dimension of R is m. This type of subspace specification is sometimes called *parametric*, because as the α_i vary, the resulting $\mathbf{u}$ vary over R.

A second method specifies a subspace by supplying a set of covectors *orthogonal*

to R. In this case we require each vector of R to satisfy

$$\mathbf{v}_i^* \mathbf{u} = 0, \quad i = 0, ..., n - m,$$

where $n \geq m$ is the dimension of the total space S. The vectors $\mathbf{v}_i$ corresponding to each $\mathbf{v}_i^*$ are each orthogonal to any vector in R; the $\mathbf{v}_i$ thus determine the *orthogonal complement* to R. This form is often called *implicit* specification, since the subspace is defined by a set of equations that its vectors must satisfy. In general, a specification may be *mixed* consisting of both parametric and implicit parts.

If $m > n/2$, the pure implicit form is more compact than the pure parametric form since fewer vectors (or linear functionals) need be given. If $m < n/2$, the parametric form provides the terser specification. In three-space, a line is typically specified with a vector giving its direction (parametric form), although two independent linear functionals (or vectors orthogonal to the line's direction) are sometimes used (*Plucker coordinates*). A plane is usually specified with a single vector orthogonal to it (a normal vector).

Converting from parametric to implicit form or vice versa may require arbitrary choices. If a plane R_a in three dimensions passes through $\mathbf{P}$ and is spanned by $\mathbf{v}_1$ and $\mathbf{v}_2$, its parametric equation is

$$\mathbf{P} + \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2.$$

Its implicit form is immediately found as

$$(\mathbf{v}_1 \times \mathbf{v}_2) \cdot (\mathbf{Q} - \mathbf{P}) = \mathbf{0}.$$

for all $\mathbf{Q} \in R$. However, if we are given only the point $\mathbf{P}$ and a normal vector $\mathbf{n}$, finding two spanning vectors requires arbitrarily choosing a vector $\mathbf{w}_1$ perpendicular to $\mathbf{n}$ after which a second one can be found as $\mathbf{w}_2 = \mathbf{n} \times \mathbf{w}_1$.

2.7 Projections

A *projection* is a linear operator that maps a vector to a fixed subspace. Consider a linear subspace R of a linear space S. Pick a basis C_R on R and C on S. If $C_R = \{\mathbf{c}_1, \ldots, \mathbf{c}_m\}$ and $C = \{\mathbf{c}_1, \ldots, \mathbf{c}_m, \ldots, \mathbf{c}_n\}$, then a projection $P : L \mapsto S$ can be written

$$P = C_R \Sigma C^{-1}$$

where Σ is a rectangular matrix with ones along its diagonal and zeros elsewhere. The effect of P is to first compute the coordinates of a vector in a coordinate system C in which a subset of the basis vectors span the subspace, toss away the unneeded coordinates by applying Σ, then get the projected vector in the subspace with the coordinate system C_R.

We can use an inner product to define an *orthogonal projection*; if P is an orthogonal projection, then

$$P(\mathbf{w})^*(\mathbf{w} - P\mathbf{w}) = 0$$

for any vector $\mathbf{w}$. To obtain this relationship, we define

$$C_R{}^* = \begin{pmatrix} \mathbf{c}_1^* \\ \vdots \\ \mathbf{c}_m^* \end{pmatrix}.$$

It is easy to verify that

$$P = C_R(C_R{}^* C_R)^{-1} C_R{}^*.$$

For instance, consider the situation depicted in Figure 2.2 in which the position of the screen basis E is specified in terms of of a unit vector $\mathbf{e}$ giving the eye direction (the direction into the screen) and a second unit vector $\mathbf{f}$ that projects onto the screen's y-axis. Then $E = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$ with

$$\begin{aligned} \mathbf{v}_3 &= \mathbf{e}, \\ \mathbf{v}_2 &= \frac{(I - \mathbf{e}\mathbf{e}^*)\mathbf{f}}{\|(I - \mathbf{e}\mathbf{e}^*)\mathbf{f}\|}, \\ \mathbf{v}_1 &= \mathbf{v}_2 \times \mathbf{v}_3. \end{aligned}$$

$\mathbf{e}\mathbf{e}^*$ is the orthogonal projection onto $\mathbf{e}$, so $I - \mathbf{e}\mathbf{e}^*$ is the orthogonal projection onto the plane perpendicular to $\mathbf{e}$. $\mathbf{v}_1$ requires no normalization because $\mathbf{v}_2$ and $\mathbf{v}_3$ are perpendicular and each have unit length.

2.8 Multilinear Operators

So far we have considered vectors, covectors, linear operators (linear maps), and bilinear functionals (inner products). These objects belong to a class of general objects called *multilinear operators*[17]. An n-linear operator takes n linear space elements as argument and produces a linear space element as result. The term multilinear means that the operator is linear in each individual argument element for any choice of the remaining argument elements.

Multilinear operators of practical interest take as argument a collection of p vectors from a particular linear space and q vectors from the corresponding dual space (in general, the order in which the vectors are supplied to make up the argument is significant). Such a $p+q$–linear operator is said to be *contravariant* of order p and *covariant* of order q. The terminology comes from the way in which coordinates of the operator transform when expressed in a particular coordinate system. If the range of a multilinear operator is the real line, then it is called a *tensor*; a tensor T is a mapping

$$T : \underbrace{V \times V \times \ldots \times V}_{p} \times \underbrace{V^* \times V^* \times \ldots \times V^*}_{q} \rightarrow R.$$

If V is the space of n-tuples, then T is sometimes called a *classical* tensor[18].

2.9 Determinants, Cross Products and Rigid Motions

The determinant is central to the algebra of the cross product and related operators, as well as their generalizations in higher dimensions. These operators, in

turn, form a means for concise representation of rigid motions.

The determinant in n dimensions is an n-linear functional taking n vectors to a real number. It is also skew-symmetric, meaning that its value changes sign if any two argument vectors are interchanged. It turns out that any skew-symmetric tensor of covariant order n in n dimensions is unique up to a constant factor[19]. The determinant is the particular tensor of this form whose value is equal to one when evaluated on an orthonormal set of vectors.

The determinant partitions the set of bases on a space into two equivalence classes: those with positive determinant and those with negative determinant. Those bases assigned a positive value are *positively oriented*, while those assigned a negative value are *negatively oriented*. Because of skew-symmetry of the determinant, interchanging two vectors of a basis reverses its orientation. In three dimensions, for instance,

$$\det(\mathbf{u}, \mathbf{v}, \mathbf{w}) = -\det(\mathbf{w}, \mathbf{v}, \mathbf{u}).$$

The determinant is useful in its own right because it computes the oriented volume of the parallelepiped spanned by the argument vectors. In particular, if any subset of its argument vectors is linearly dependent, det = 0.

These properties allow the determinant to be used as the algebraic starting point for deriving several useful functionals and operators in coordinate-free form. Let $f(\mathbf{v_1}, \ldots, \mathbf{v_n})$ to be the n-linear functional f applied to the tuple of vectors $(\mathbf{v_1}, \ldots, \mathbf{v_n})$. Consider the bilinear functional $B^{\mathbf{u}}$ derived from the three-dimensional determinant by

$$B^{\mathbf{u}}(\mathbf{v}, \mathbf{w}) = \det(\mathbf{u}, \mathbf{v}, \mathbf{w}).$$

Another way of writing $B^{\mathbf{u}}$ makes use of the *exterior derivative* operator[20] $\lrcorner$:

$$(\mathbf{u} \lrcorner \det)(\mathbf{v}, \mathbf{w}) = \det(\mathbf{u}, \mathbf{v}, \mathbf{w}).$$

Computing a second exterior derivation $\mathbf{u}\lrcorner\mathbf{v}\lrcorner det$ (exterior derivation is associative), we obtain

$$(\mathbf{u}\lrcorner\mathbf{v}\lrcorner det)(\mathbf{w}) = \det(\mathbf{u}, \mathbf{v}, \mathbf{w}).$$

It is easy to see that the dual of the cross-product, $(\mathbf{u} \times \mathbf{v})^*$, satisfies the same relationship as $\mathbf{u}\lrcorner\mathbf{v}\lrcorner det$.

Returning to the skew-symmetric bilinear functional $\mathbf{u}\lrcorner det$, we can define a linear mapping $\mathbf{u}^\times$ by

$$\mathbf{w}^*(\mathbf{u}^\times\mathbf{v}) = (\mathbf{u}\lrcorner det)(\mathbf{v}, \mathbf{w}) = \det(\mathbf{u}, \mathbf{v}, \mathbf{w}).$$

$\mathbf{u}^\times$ is the linear map corresponding to the bilinear functional $\mathbf{u}\lrcorner det$. That is, $\mathbf{u}^\times\mathbf{v} = \mathbf{u} \times \mathbf{v}$.

The operator $\mathbf{u}^\times$ can be used to *differentially* specify any rigid motion in three dimensions. To do so, consider the equation $NM(\theta) = \frac{d}{d\theta}M(\theta)$, where N and M are both linear mappings of a space onto itself (automorphisms). The solution to this equation is

$$M(\theta) = exp(\theta N).$$

N specifies a *vector field* over its domain. $M(\theta)$, applied to a vector $\mathbf{v}$, computes the "solution curve" of this vector field at time θ starting from $\mathbf{v}$. (That is, if a particle is placed at an offset $\mathbf{v}$ from the origin at time $\theta = 0$ and its velocity at each point is given by the vector field N, then the particle will be offset from the origin by $exp(\theta N)\mathbf{v}$ at time θ. If the vector field is given by $\mathbf{u}^\times$ then $exp(\theta\mathbf{u}^\times)$ is a proper orthogonal transformation (a rotation) parametrized by θ. If $\mathbf{u}$ is a unit vector, θ is the rotation angle in radians about the axis $\mathbf{u}$. This method of specifying a rotation (or, when origins are added, any rigid motion[21]) is compact and displays the relevant parameters clearly.

These techniques can be applied in n-dimensions to yield similar operators. As one example, the *external product* of $n-1$ vectors $(\mathbf{v}_1, \ldots, \mathbf{v}_{n-1})$ in n dimensions is

$$\mathbf{v}_1\lrcorner \ldots \lrcorner\mathbf{v}_{n-1}\lrcorner det.$$

The external product is the n-dimensional generalization of the dual of the cross-product because the resulting covector is orthogonal to each of the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$.

The method of differential specification of a rigid motion extends to non-positive definite inner products. Relativity provides an example. A Lorentz transformation T is a rigid motion of Minkowski space. It is therefore expressible as $T = exp(\theta S)$, where S is a skew-symmetric operator[22]. In this case T describes the motion in Minkowski space of an object starting from rest undergoing uniform acceleration (possibly coupled with uniform rotation) after proper time θ.

2.10 Derivatives

Differential geometry attempts to describe the structure of surfaces of arbitrary dimension through relationships among linear approximations to these surfaces. Therefore, I introduce notation for derivatives of geometric quantities so that programming language constructs can be developed that can aid in the computer solution of problems in differential geometry.

Suppose a vector $\mathbf{v}$ is given by its coordinates in some coordinate system C so that $\mathbf{v} = C\vec{\mathbf{v}}$. Then $\mathbf{v}' = C\vec{\mathbf{v}}' + C'\vec{\mathbf{v}}$. This formula can be interpreted as stating that the velocity of a vector in a moving coordinate system is a sum of two components: the velocity of the vector relative to the moving system and the velocity due to the movement of the system itself[23]. The velocity of the system (C') is always relative to some other (fixed) system.

If $\vec{\mathbf{v}}$ is constant in C (that is, $\mathbf{v}$ remains fixed relative to C), then $\mathbf{v}' = C'\vec{\mathbf{v}}$. Carrying this reasoning from the vector $\mathbf{v}$ to a whole coordinate system C, suppose $C = XD$ where X is a linear transformation and D is another coordinate system. Then $C' = XD' + X'D$. Since D is fixed relative to itself, $C' = XD'$ relative to D. In this way a transformation rule for velocities of vectors and coordinate systems is obtained that appears formally the same as the change of coordinate formulae for fixed vectors. The component of the velocity due to the movement of

the coordinate system relative to a fixed system is hidden by expressing a vector velocity relative to the coordinate system in which the vector itself is expressed. If the velocity relative to some other coordinate system is desired, an implicit addition of relative coordinate system velocities is carried out.

This technique can be used for partial derivatives as well, as long as partial derivatives relative to differing variables are kept separate. It can also be used for higher order derivatives.

Chapter 3

Programming Language Constructs for Geometry

The notation and methods of the last chapter form the basis for programming language objects and operators that facilitate the programming of geometric problems. These constructs can, in principle, be embedded in most any programming language, as they do not affect control structures. However, they are most naturally added to a declarative language admitting the use of infix operators (PASCAL or C, for example).

This section may be treated as a manual for the language constructs. The reader may find it simplest to read sections 3.1 through 3.4 and the beginning of section 3.5, and then skip to Chapter 4, referring to this chapter as required.

3.1 Spaces

All geometry is based on the structure of one or more spaces. A space consists of a collection of vectors or points of the same dimension. No computation is carried out on a space, but rather on the objects that refer to them. Therefore spaces have function at declaration time only.

A space declaration associates a space *name* with a number of *dimensions*. The

number of dimensions may vary at run time, although compile time optimizations may be possible if it is fixed. Spaces with different names, even if they have the same number of dimensions, are distinct.

Additional structure may be created by grouping two or more spaces to form a cartesian product space. The order of the spaces in such a grouping is significant. For example, we might set

$$S = S_1 \times \ldots \times S_n.$$

Once again, n may vary at run time. As an example, one may be building an interactive robot arm modeler that accepts as input a number of joints. Each joint is assigned a configuration space S_i, so that the total configuration space of the arm is S. If all of the S_i are the same S_1, we may write

$$S = S_1 * *n.$$

The syntactic definition of a space declaration is:

---

spaceDeclaration :=
  space *spaceName* **dimension** *n* ;
  space *spaceName* (spaceList) ;
  space *spaceName* spaceName ** *n* ;
spaceList :=
  spaceName
  spaceName , spaceList

In this as in all syntactic definitions, the bold typeface indicates keywords or literals, while italics indicate variable names. Syntactic categories are indicated by roman type. Different forms of a syntactic category are placed on separate lines. An optional argument is denoted by placing angle brackets around it. The character *n* signifies a positive integer.

Examples of space declarations:

```
space M3 dimension 3;   /* Three dimensional space */
space joint dimension 2;   /* joint with 2 d.o.f. */
space config6 joint ** n ;   /* 6d configuration space */
```

A space declaration actually induces the definition of three spaces. The first is the linear space of vectors, denoted by the unadorned name given the space. The second is the space of covectors on that linear space, denoted by affixing the adjoint sign (†) to the end of the space name (the adjoint sign is used instead of the standard asterisk because asterisk is already reserved as the multiplication sign). The third is the corresponding affine space of points, denoted by prefixing the space name with the keyword `affine`.

For each n there is a predefined space of vectors, called the n-tuples, that corresponds to an array of n real numbers aligned in a column. These spaces are given the names `Rn`. The dual space to `Rn` is `Rn`†, an array of n real numbers aligned in a row. There is no affine space corresponding to the n-tuples.

3.2 Maps

Once spaces have been defined, geometric objects that inhabit or refer to those spaces may be introduced. Points and vectors, as well as all of the other objects mentioned in the last chapter, can be handled in a unified framework.

We begin by considering only linear objects (vectors, covectors, linear maps and tensors). All these objects can be thought of as multilinear operators with one or more domain spaces and a single range space. (General multilinear operators, operating on tensor product spaces, could be treated by allowing multiple range spaces, but I have not found application for this generality). For example, a vector can be treated as a map from the real numbers (1-tuples) to a linear space. A coordinate system is a map from the n-tuples to a linear space of dimension n. The determinant is a map from n identical vector spaces of dimension n to the reals.

A linear object is declared:

---

objectDeclaration :=
map domainSpaces -> rangeSpace *name* ;

```
domainSpaces :=
    domainSpace
    ( domainSpaceList )
domainSpaceList :=
    domainSpace , domainSpace
    domainSpace , domainSpaceList
domainSpace :=
    spaceName
rangeSpace :=
    spaceName
```

Examples of linear object declarations:

```
map R^1-> M3 vec;   /* A vector */
map M3 -> R^1 covec;   /* A linear functional */
map R^3-> M3 basis;   /* A coordinate basis */
map R^3-> R^3 mat;   /* A matrix */
map (M3,M3) -> R^1 B;   /* A bilinear functional */
```

To make declarations more palatable, it is essential to provide macros for declaring the most common objects. In particular, the declarations

```
vector in M3 vec;
vector in M3^† covec;
```

are equivalent to the first two example declarations.

A geometric object is actually stored as an array of coordinates and a list of pertinent coordinate bases. The items in this list are called the *bases of storage*.

With some simple extensions, the same methods can be used to handle affine objects and maps. All that is required is a mechanism to indicate which of the spaces are affine, and for linear spaces, to indicate when a map has a vector offset in either its domain or its range. An affine space is indicated by prefixing the `affine` keyword to its name. A similar construction, using the keyword `offset` placed before a space name, indicates a map with a vector offset in the linear space

it operates on or maps to. A vector offset in a domain space is subtracted from the argument vector before the map is applied to it. A vector offset in a range space is added to the vector result of a map.

These changes are incorporated by expanding the syntax for space names in the map declaration:

---

spaceName :=
space
affine *space*
offset *space*

Examples of affine object declarations:

```
map M3 -> R1 afunc;   /* An affine coordinate */
map R3 -> affine M3 aBasis;   /* An affine coordinate basis */
map R3 -> offset R3 aMat;   /* An affine coordinate transformation */
```

3.3 Points

The only elementary geometric object that cannot be cast as some type of map is the point. A point is simply a location in space. Therefore, a special construction is used to declare a point:

---

pointDeclaration :=
point in spaceName *name* ;

---

so that, for example

```
point in M3 p ;
```

makes p a point in M3. A point is an element of the affine space corresponding to the named space; a point may not be declared as an element of a space of n-tuples.

3.4 Scalars

A scalar is a real number. A scalar may be identified with a linear map of a linear space onto itself, where the map is given by multiplication of the argument vector by the scalar. In particular,

```
map R^1 -> R^1  s  ;
```

must declare `s` to be a scalar. However, it is more natural to use the host language's syntax for real number declaration:

```
real s  ;
```

or something similar.

In light of this equivalence of R^1 with scalars, adding spaces of 1-tuples to a linear map's domain space list has no effect on the declared map. For instance,

```
map R^3 -> R^3  M  ;
```

and

```
map R^3,R^1 -> R^3  M  ;
```

declare the same object. The reason is that if x_2 is a scalar, then linearity of M implies $M(x_1, x_2) = M(x_1) \cdot x_2$. There is nothing to be gained by adding a scalar argument to a linear object; the scalar can be applied separately.

3.5 Arrays

Scalars are grouped into arrays that represent geometric objects. For instance, a covector in a space of dimension n is represented by a row n-tuple, which is an array of n scalars arranged in a row.

```
real tuple[n] ;
```

declares such an n-tuple. But an n-tuple is also a map from scalars to the space of n-tuples, so that

$$\texttt{map } \mathrm{R}^n \texttt{ -> } \mathrm{R}^1 \texttt{ tuple ;}$$

makes `tuple` a row n-tuple. Therefore, both declarations of `tuple` must be treated as equivalent.

If a set of geometric objects of the same type have a linear range space, then it makes sense to form linear combinations of them. The linear combination can be formed by arranging the maps in a row and applying them to an n-tuple representing the coefficients.

In this way, adding an array subscript to a geometric object adds the corresponding space of n-tuples to the end of the list of the object's domain spaces. For instance,

$$\texttt{map M3 -> } \mathrm{R}^1 \texttt{ C[3] ;}$$

arranges three covectors in a row to form a coordinate system on `M3`; the declaration

$$\texttt{map M3 -> } \mathrm{R}^3 \texttt{ C ;}$$

is equivalent.

In general, the declaration

$$\texttt{map } S_1, \ldots, S_p, \mathrm{R}^{n_1}, \ldots, \mathrm{R}^{n_q} \texttt{ -> } S_{p+1} \texttt{ obj ;}$$

(without loss of generality, all n-tuple spaces appear at the end of the domain space list) is exactly the same as

$$\texttt{map } S_1, \ldots, S_p, \mathrm{R}^{n_1}, \ldots, \mathrm{R}^{n_{q-1}} \texttt{ -> } S_{p+1} \texttt{ obj[}n_q\texttt{] ;}$$

and, by repeated application, the same as

$$\texttt{map } S_1, \ldots, S_p \texttt{ -> } S_{p+1} \texttt{ obj[}n_1\texttt{]}\cdots\texttt{[}n_q\texttt{] ;}$$

This equivalence identifies array dimensions with domain spaces of n-tuples. To obtain a range space of n-tuples, a second subscript indicator, used in declarations only, is introduced. Thus,

```
real column<3> ;
```

declares `column` to be a column of 3 scalars, the same as

```
map R^1 -> R^3  column  ;
```

Only one use of angle brackets is allowed, and it must be the last subscripter in the declaration.

To test the array equivalence in general, consider the map

$$M : S_1, ..., S_p, \mathrm{R}^{n_1}, ..., \mathrm{R}^{n_q} \mapsto S_{p+1}.$$

Define $M(x_1, ..., x_p)$ with $x_i \in S_i$ to be a map

$$\mathrm{R}^{n_1}, ..., \mathrm{R}^{n_q} \mapsto S_{p+1}.$$

But this map is canonically expressed by giving its values on combinations of the basis vectors, one from each of R^{n_j}. $\prod_{i=1}^{q} n_j$ values are required, each given by $M(x_1, ..., x_p, e^1_{i_1}, ..., e^q_{i_q})$ where $1 <= i_j <= n_j$ and e^j_i is the ith canonical basis vector of R^{n_j}. Each combination of canonical basis vectors is indexed by a combination of array subscripts in the appropriate ranges: $M(x_1, ..., x_p)[i_1] \cdots [i_q]$.

Array bounds that specify a solitary array element are equivalent to adding a space of 1-tuples, which has no effect on an object's type. Therefore, single element array specifications are ignored.

Finally, an n-tuple space created by adding an array specification can be made contravariant (the default) or covariant. A covariant space is indicated by affixing the adjoint sign to the array specification. For instance

```
map R^3,R^3†,R^3†,R^3 -> R^1  T  ;
```

or equivalently

real T[3][3]†[3]†[3] ;

declares the classical tensor on R^3denoted by $T^i_{jk}{}^l$.

3.6 Vectors and Covectors Revisited

Consider the four declarations:

map M3 -> R^1 v1 ;
map R^1 -> M3† v2 ;
map M3† -> R^1 v3 ;
map R^1 -> M3 v4 ;

The first two objects, v1 and v2, are both covectors of exactly the same type, by the definition of M3†. v4 is a vector in correspondence with the covector v2. v3 should also be a vector, in correspondence with v1. This outcome is achieved by noting that the adjoint operator applied to a space reverses the point of argument application for an element in that space. That is, a vector may be considered as an operator taking a covector to a scalar. Normally, the covector taken as argument is placed immediately to the left of the vector. A covector takes a vector to a scalar when the vector appears directly to the right of the covector. Thus v3 and v4 have exactly the same type.

The relationship between an object $U_1 : S_1, ..., S_p \mapsto S_{p+1}$ and another $U_2 : S_1, ..., S_p, S^\dagger_{p+1} \mapsto R^1$ is made explicit by the relationship

$$U_2(x, y) = y^* U_1(x)$$

with $x \in S_1 \times ... \times S_p$ and $y \in S_{p+1}$. If we were to define

$$U_2(x) = U_2(x, \cdot) \equiv U_1(x),$$

then U_1 and U_2 could be represented as the same object. However, we make this identification only for the cases of vectors and covectors. For example, although

they follow the same transformation rules, it apparently does not make sense to add a bilinear functional on $S_1 \times S_1^\dagger$ to an automorphism of S_1.

3.7 Representation

3.7.1 Linear and Affine Part Selection

Origins, offsets, and linear parts of affine maps may be selected by treating them as compound data types and applying an appropriate built in selection operator. The selectors are `Daff` (for domain affine part), `Raff` (for range affine part), and `lin` (linear part). A map with multiple domain spaces requires that the selector `Daff` be subscripted so that the appropriate origin or offset can be selected. The subscript number indicates the element of the space list to be used. (Even a linear space with no offset is assigned a subscript, although it is illegal to extract an affine part from such a space.)

For instance, if `A` is an affine coordinate basis on a space S declared by

```
map R^3 -> affine S  A  ;
```

then `A.lin` obtains the linear map from n-tuples to the linear space embedded in S, `A.Raff` obtains the origin of the affine basis, and `A.Daff` is illegal.

3.7.2 Coordinate Representation

A map or a point g is represented as a list of pointers to coordinate bases and an array of coordinates. The implication is that if a coordinate basis changes (relative to the intangible basis), g changes with it. Changes automatically propagate through coordinate bases defined in terms of one another.

The coordinate bases referred to by an object may be accessed using the selectors `Dbas` (domain basis) and `Rbas` (range basis). A map with multiple domain spaces requires the selector `Dbas` to be subscripted just as with the selector `Daff`.

The result of such a selection is a pointer to the coordinate frame in the indicated space.

3.8 Expressions

A geometric expression is a multi-dimensional arithmetic expression in some basis. The geometric result of a geometric expression is independent of the basis chosen for calculation, although the result must be represented with coordinates. An appropriate basis in which to calculate can usually be determined from the bases used for representing the expression's operands.

Expressions involving geometric objects can appear anywhere that a host language expression can appear. In particular, some geometric expressions evaluate to scalars. For instance, one might want to branch on the sign of a dot product:

```
if (v† * u > 0)
    return(TRUE);
else return(FALSE);
```

3.8.1 Terminology

Every geometric object is either a map or a point. A map may refer to several spaces in its domain and one in its range. Two space references *match* if they name the same space. Two space references *match exactly* if they match and the same modifications (**affine**, **offset**, or none) are applied to the matching space. A space reference is *linear* if no modifications are applied to it. Two maps have exactly matching domain space lists if both lists contain the same number of spaces and the corresponding elements match exactly. Two maps are of the same *type* if their domain space lists match exactly and their range spaces match exactly.

3.8.2 Binary Operations

Addition (+)

Two maps may be summed if their domain space lists match exactly, their domain spaces match, and at most one of the domain spaces is affine. The result has the same list of domain spaces as the operands. The range space is affine if either operand has an affine range. It is linear if both operands have linear ranges. Otherwise, it is offset linear if either operand has an offset linear range.

The general formula for finding the sum of two maps is found by setting $T_1(x) = L_1(x - x_1) + y_1$ and $T_2(x) = L_2(x - x_2) + y_2$ and computing the sum R:

$$R(x) = (L_1 + L_2)(x - x_1) + L_2(x_1 - x_2) + y1 + y2.$$

Each of x_1, y_1 and x_2, y_2 may or may not be present. Each x_i and y_i, if present, may be either a vector or a point, depending on the type of T_1 and T_2. While addition of maps is commutative, the representation of the result may depend on the order of the operands if either of x_1 and x_2 are present. This is because the domain offset of the result is taken to be x_1 and the term $L_2(x_1 - x_2)$ is added to the result's range offset.

As examples of addition, two tensors may be summed if their covariant and contravariant degrees agree and they operate on the same spaces; the result has the same type as the operands. Two affine coordinate bases may be added to yield a new affine coordinate basis. Two matrices of equal dimensions may be added to yield a new matrix of the same dimensions.

A point and vector in the same space may be added to yield a point in that space.

Subtraction (-)

The operands allowable for subtraction follow the same rules as for addition except that if either operand has affine range space, it must be the first. Similarly, a vector

may be subtracted from a point, but not vice versa. The formula for the difference of two maps T_1 and T_2 is

$$R(x) = (L_1 - L_2)(x - x_1) + L_2(x_1 - x_2) + y_1 - y_2.$$

Composition (∘)

The range space of the first operand must match the (solitary) domain space of the second. If one of the spaces is affine, both must be. The result has the domain space list of the first operand and the range space of the second. However, if this range space is linear, the result's range space will be offset linear if either (1) the first map has an offset linear or affine range (2) the second map has an offset linear or affine domain. The reason for this modification can be seen by writing two maps in general form and computing their composition. Let $T_1(x) = L_1(x - x_1) + y_1$ and $T_2(x) = L_2(x - x_2) + y_2$. The composition is $R = T_2(T_1(x))$ or

$$R(x) = L_2(L_1(x - x_1) + y_1 - x_2) + y_2 = L_2L_1(x - x_1) + L_2(y_1 - x_2) + y_2.$$

As in the case of addition, each of the x_i and y_i may be a vector or a point if it is present at all. The term $L_2(y_1 - x_2)$ is present unless $y_1 \equiv x_2$ (or neither are present at all) requiring that R have an offset in its range. No provision is made to check if $y_1 \equiv x_2$, so if either of these terms is present, the range of R is either affine (if y_2 is a point) or offset linear (if y_2 is a vector or is not present).

A point may not be an operand of composition.

Multiplication (*)

Multiplication can be used in place of composition if both operands refer to only linear spaces in their domains and ranges. Otherwise, the rules for and results from multiplication are the same as those for composition. For instance, two matrices of appropriate dimensions (that is, with the appropriate n-tuple spaces

as domain and range) may be multiplied to yield another matrix. Or, a vector may be multiplied by a covector to yield a scalar.

Tensor Product ($\otimes$)

Operands for tensor product must both be linear maps (refer to only linear domain spaces) with R^1 as range. The result is a map with domain space list equal to the concatenation of the first operand's domain space list followed by the second operand's domain space list. The range of the result is R^1.

A vector may be an operand of tensor product; for this purpose it is considered a map from the appropriate space of covectors to R^1.

In coordinates, if T_1 and T_2 are arrays, the resulting array R is given by

$$R[i_1]\cdots[i_n][j_1]\cdots[j_m] = T_1[i_1]\cdots[i_n]\cdots T_2[j_1]\cdots[j_m].$$

Cross Product ($\times$)

Cross product is allowed in the language extension between two vectors or covectors in the same three-dimensional space. The result has the same type as the arguments. There is a generalization of the cross-product to higher dimensions, but for simplicity (and because there would probably be no use for it), it is only allowed in three-dimensional spaces.

The cross product of two vectors, $\mathbf{u} \times \mathbf{v}$ is implemented as $\det(\mathbf{u}, \mathbf{v}, \bullet)^\dagger$ (see below).

Assignment (=)

One object may be assigned to another of identical type. The result of an assignment is the right hand side of the assignment expression after the assignment has taken place (thus, assignments may be nested).

In addition, a format similar to that of FORTRAN's one-line function statement may be used to specify the significance of arguments to the assigned map. See the section below on argument evaluation.

3.8.3 Unary Operators

Unary Minus (-)

Unary minus may be applied to any map with linear or linear offset range. The result has the same type as the operand.

Inverse (post superscript −1)

Inversion may be applied to any map whose domain and range space have the same number of dimensions. The result is a map with domain space equal to the operand's range space, and range equal to the operand's domain. There is no provision to check that a map is actually invertible or to detect poorly conditioned inverse computations. Responsibility for detecting and handling such cases rests with the programmer.

Adjoint (post superscript †)

Adjoint may be applied to a vector, a covector, or a map with a single linear domain space and linear range space that match. The adjoint operation swaps the range and domain spaces of its operand. The adjoint of a vector is the corresponding dual covector, and the adjoint of a vector is the corresponding vector. The effect of the adjoint operator is determined by the inner product currently in effect on the space referred to by the operand (see below). The adjoint of an n-tuple is its transpose, as is the adjoint of a matrix.

Unary cross (post superscript ×)

Unary cross may be applied to a vector or covector in a three-dimensional space. The result is a skew-symmetric linear map (a map with range and single domain spaces both exactly equal to the single space referred to by the operand). $s^{\times} = s \lrcorner det$, where *det* is the determinant on the space referred to by s.

3.8.4 Argument Evaluation

A map may be applied to arguments in the same way as a function is called with arguments in the host language. That is, a map's value on a list of arguments is found by appending the arguments, separated by commas and enclosed in parentheses, to the name of the map. The number of arguments must be the same as the number of domain spaces of the map. Each argument must belong to the corresponding domain space. If the domain space is affine, the argument must be a point in that space. If the domain space is linear or offset linear, the argument must be a vector in that space. The result of argument application is an element of the map's range space.

Not all arguments need be supplied to a map with multiple domain spaces. A new map with a reduced number of domain spaces results from partial evaluation of a map, corresponding to exterior derivation. An unevaluated argument is indicated with the placeholder •. The resulting map has the list of domain spaces of the original map with each space corresponding to an evaluated argument deleted. The domain spaces that remain correspond to the unevaluated arguments. The range space of the result is the same as that of the original map. For instance, if `det` is the determinant in three dimensions, then `det(u,v,•)` is $u \lrcorner v \lrcorner det$. That is, `det(u,v,•) * w = det(u,v,w)`. Similarly, `det(u,•,•)` is $u \lrcorner det = u^{\times}$ so that

$$\texttt{det(u,•,•)(v,w)} = \texttt{det(u,v,w)}.$$

Arguments may be applied to a map on the left hand side of an assignment. When this occurs, the arguments are treated as formal parameters to the expression appearing on the right hand side of the assignment. The type of each argument is given by the corresponding space of the map on the left hand side. For instance, if B is a bilinear functional on M3, C is a coordinate basis on M3, and M is a 3×3 matrix, then B may be assigned by

$$\texttt{B(x1,x2)} = (\texttt{C}^{-1} * \texttt{x1})^{\dagger} * \texttt{M} * \texttt{C}^{-1} * \texttt{x2} \ ;$$

The expression on the right hand side must be linear or affine (depending on the type of the map being assigned) in each argument. Partial evaluation is not allowed on the left hand side of assignments.

3.8.5 Subscripting

Arrays may be subscripted just as host language objects are subscripted. Because of the identification of maps that refer to n-tuple spaces with arrays, a subscripter applied to a map is equivalent to evaluating the map on a certain canonical basis tuple of R^n. Consider the list of n-tuple spaces formed by appending a map M's range n-tuple space (if there is one) to the list of its domain n-tuple spaces (if there are any). If the first n-tuple space in this list is R^k with canonical basis tuples $e_1, \ldots, e_k$ ($e_1 = (1,0,\ldots,0)^{\dagger}$, $e_2 = (0,1,\ldots,0)^{\dagger}$, etc.) then application of the subscripter [i] to M yields

$$\texttt{M[i]} \equiv \texttt{M}(\bullet, \ldots, \bullet, e_i, \bullet, \ldots, \bullet)$$

where the argument e_i is placed in the position corresponding to the first n-tuple space in M's list. Further subscripters cause partial evaluation on basis tuples of each space in the n-tuple space list in sequence. If any of the n-tuple spaces are dual spaces (spaces of row n-tuples), the corresponding row basis tuple is used in place of the column tuple.

Evaluation on particular n-tuple spaces can be suppressed by using the empty subscripter `[]` as a placeholder. For instance, if a matrix `A` is declared

```
map R^3 -> R^3  A ;
```

then `A[i][j]` selects a_{ij}, `A[][j]` selects the jth column of `A`, while `A[i][]` (the same as `A[i]`) selects the ith row of `A`.

A subscripted map may appear on the left hand side of an assignment.

3.8.6 Grouping (Cartesian Product)

Two or more maps may be grouped together to form a new map. Two types of grouping operations are possible: the first creates a new map whose range space is the cartesian product of of the grouped maps' range spaces taken in order; the domain space lists of the grouped maps must match exactly. This grouping is indicated by enclosing the comma separated list of maps in the range grouping brackets $[_r \ ]_r$. The second grouping operation interchanges the role of range and domain spaces. Domain grouping (indicated with $[_d \ ]_d$) requires that the maps for grouping have only one domain space. For either range or domain grouping, the indicated cartesian product space must have been declared as such explicitly (although that cartesian products of n-tuple spaces exist automatically).

The grouping operators are intended to make block operator construction straightforward. For instance, if A is a 3×3 matrix, t is a column 3-tuple and u is a row 3-tuple, then

$$\begin{array}{l} [_r \quad [_d \, A, t \,]_d, \\ \qquad [_d \, u, 1 \,]_d \ \ ]_r \end{array}$$

creates a 4×4 matrix.

3.9 Coordinate Bases

Every geometric expression must actually be carried out as a coordinate manipulation in some coordinate basis or bases. Ideally, the coordinate bases selected are irrelevant to the value of the geometric expression. However, because geometric objects are represented as finite precision quantities in the computer's memory, and because converting from one coordinate basis to another is slow, the bases actually selected can have consequences for program accuracy and speed. A succinct notation is required to specify the bases used in the evaluation of a geometric expression.

3.9.1 The @ Operator (prefix @($b_1, \ldots, b_n$) or @)

@ has two forms. In the first form, it takes a list of coordinate bases as arguments. The coordinates of the geometric object following the @ are extracted in the specified bases. On the right hand side of an assignment `@(b)` acts just like `b^-1 *`, but it allows the specification of any number of coordinate systems and may appear on the left hand side of an assignment. The assignment

`@(`$b_1, \ldots, b_n$`)lvalue =` *exp* ;

sets the coordinates of `lvalue` to the array *exp* in the bases supplied to @. No evaluation takes place; the bases of storage of `lvalue` are set to the list enclosed in parentheses. For example,

```
@(b)v = [r 0 , 0 , 1 ]r ;
```

sets the coordinates of the vector `v` to be $(0,0,1)^T$ in the basis `b`.

In its second form, @ is a shorthand to control basis selection in binary expressions. In the expression

$$exp_1 \; binop \; exp_2$$

the expressions exp_1 and exp_2 may possess differing bases of storage. Typically, it is desirable to convert the appropriate bases of exp_2 to those of exp_1 or vice versa

so that the coordinate manipulations indicated by *binop* can be performed.

$$@exp_1\ binop\ exp_2$$

forces the conversion of exp_2's coordinates to pertain to the bases of storage of exp_1 while

$$exp_1\ binop\ @exp_2$$

has the opposite effect.

@ may not be applied to both exp_1 and exp_2. The default is for

$$exp_1\ binop\ exp_2$$

to be evaluated as

$$@exp_1\ binop\ exp_2.$$

That is, the coordinate bases used for evaluation of a binary expression are the bases of storage of the first operand.

The precise effect of @ depends on *binop*. If *binop* is addition, subtraction, cross product, or assignment, then

`@v` *binop* `w`

is short for

`@(v.Dbas[1],...,v.Dbas[n],v.Rbas)` *binop* `@(v.Dbas[1],...,v.Dbas[n],v.Rbas)w`

with one difference: the result of *binop* has the indicated bases of storage, but `w` itself is left unaffected (`v.Dbas[]` and `v.Rbas` may or may not be present, depending on the type of `v` and `w`). If *binop* is multiplication or composition, then `@v` *binop* `w` is short for

`@(v.Dbas,v.Rbas)` *binop* `@(w.Dbas[1],...,w.Dbas[n],v.Rbas)w`

Note that the bases of storage for the domain spaces of the result are the domain spaces of `w`. Again, no conversion takes place on the value stored in `w`.

@ can be used in argument application in an analogous way.

`@v(w1,...,wn)`

converts each `wi` to the basis `v.Dbas[i]` to compute the value of `v` applied to `(w1,...,wn)`; this is the default. The other possibility is to apply @ to any or all of the arguments `wi` to cause conversion of the ith basis of storage of `v` before evaluating `v` on the arguments.

Finally, @ has no effect on the computation of tensor product, because no coordinate conversion is required in this case.

3.10 Functions

Functions may return or be passed maps or points. Spaces may be passed to functions. A passed space may be used in the local declaration of a map or point so that generic functions of geometric objects on differing spaces may be written. A function's return value declaration may also refer to passed spaces. Spaces may not be returned by functions.

3.10.1 Built-in Functions

Several functions provide access to special geometric objects.

Intangible Basis

The function `intang` takes a space description as argument. If the space is linear (or offset linear), the function yields the map that is the intangible linear basis on the indicated vector space. If the space is affine, the intangible affine basis is returned. Applying the function `intang` to a space of n-tuples yields an $n \times n$ identity matrix.

Dual map

The function `dualMap` takes as argument a linear space name and returns the matrix that takes a vector's coordinates in the intangible basis to the corresponding covector's coordinates in the intangible dual basis. This function may appear on the left hand side of an assignment so that the duality relationship may be set.

Changing the dual map may affect the inner product on a space. If C is the intangible coordinate basis and C^* is the corresponding dual basis, then

$$C^* = MC^{-1}.$$

For a vector $\mathbf{v}$,

$$\mathbf{v}^* = (MC^{-1}\mathbf{v})^{\mathrm{T}}C^*$$

so that

$$\mathbf{v}^*\mathbf{w} = (C^{-1}\mathbf{v})^{\mathrm{T}}M^{\mathrm{T}}C^*C(C^{-1}\mathbf{w}) = (C^{-1}\mathbf{v})^{\mathrm{T}}M^{\mathrm{T}}M(C^{-1}\mathbf{w}).$$

Therefore $M^{\mathrm{T}}M$ determines the inner product when $\mathbf{v}$ and $\mathbf{w}$ are written in the coordinates of the intangible system.

`dualMap(S)` is set to the identity when `S` is declared. If `S` is an n-tuple space, `dualMap(S)` cannot be changed; it is always the identity.

Exponentiation

The **exp** function, when applied to a map with a linear range space and single linear domain space of the same dimension, yields a map of the same type that is the exponential of its argument. **exp** may not be applied to any other type of map. **exp** is meant to be used to compute rotations in three dimensions. Computing the exponential of a general square linear map may be a poor idea numerically.

Determinant

The **det** function accepts a linear space name as argument and returns the map that is the determinant on that space. **det** has n domain spaces, where n is the

dimension of the argument space, and its range is R.

Identity Map

`I(S)` returns the identity map from the space `S` into itself. `S` may be either an affine space of points or a linear space of vectors.

Rectangular Identity Tensor

`rect`$(n_1, \ldots, n_m)$ returns an array with dimensions $[n_1] \cdots [n_m]$. All entries are zero except the entries $[i_1] \cdots [i_m]$ with $i_1 = \cdots = i_m$. In particular, `rect[p,q]` returns the `p` × `q` matrix with ones along its diagonal and zeros elsewhere. This matrix is useful in building projections.

Zero Map

`zero(`*domainSpaceList* `->` *rangeSpace*`)` returns the zero map from the domain spaces *domainSpaceList* to the range space *rangeSpace*. For example, `zero(`R^3`->` R^3`)` returns the zero 3×3 matrix.

Chapter 4

Programmed Examples

This chapter presents several examples of geometric calculations and shows their implementation using the constructs introduced in the previous chapter. The aim is to show that the use of these constructs leads easily from the mathematical formulation of a geometric computation to a simple program to carry it out.

4.1 Linear Geometry

These routines perform some simple and typical geometric calculations that make sense in n dimensions, although n is usually 2 or 3. The routines are: intersection point of a line with a (hyper)plane, intersection point of n hyperplanes, and the segment of closest approach between two lines. Similar calculations underlie a variety of geometric algorithms, especially those from computer graphics and computational geometry.

4.1.1 Line — Plane intersection

This function is almost identical with that presented in chapter 1 to solve the same problem. As in that example, the line is assumed to be specified parametrically, and the plane specified implicitly. The line is represented in a single language

object as an affine basis (with one basis vector) on the input space. The plane is similarly represented as an affine coordinate that is zero on the plane. Another way of saying this is that the plane is given by a point on it and a vector normal to it. Both the line's direction and the plane's normal are assumed to be unit vectors.

In the following program, Lines 1—4 declare the function `lineWithPlane` as returning a point in the input space `S` and taking three arguments: the line `l`, the plane `p`, and the space in which they lie. Line 8 computes the dot product of the plane normal with the line's direction, which gives the cosine of the angle between these two unit vectors. Line 10 makes a crude test of the accuracy of the solution about to be computed by comparing the absolute value of the dot product with a small fixed constant `EPSILON`. Line 11 returns `NOTAPOINT` if the line and plane are deemed parallel. Otherwise, the point is computed and returned in lines 13—14.

In this as in all examples, the host language is `C`.

```
1  point in S   lineWithPlane(S,l,p)
2      space S;
3      map      R1      ->   affine S      l;
4      map   affine S    ->     R1         p;
5  {
6  double cosAngle;
7
8    cosAngle = p.lin * s.lin;
9
10   if (abs(cosAngle) < EPSILON)
11     return NOTAPOINT ;
12   else
13     return l.Rorg
14              - (p.lin * (l.Rorg - p.Dorg) / cosAngle) * l.lin;
15 }
```

Notice that line 14 could have been written

```
- (p( L.Rorg ) / cosAngle * l.lin ;
```

because p is an affine functional with origin `p.Dorg` on the domain space. In fact, a plane is often represented by an affine functional with an offset in the range

space R^1. Then the plane equation takes the form $\mathbf{n} \cdot (\mathbf{x} - \mathbf{o}) + d = 0$, where d is the offset (a scalar). Such an object is declared as

```
map affine S -> offset R^1  p  ;
```

In this case the domain origin is always the origin of the intangible affine basis, and the range offset gives the distance to the plane from the origin along the normal direction. As long as the point $\mathbf{x}$ is expressed in some basis with origin at $\mathbf{o}$, the subtraction implied by the plane equation (actually n subtractions, where n is the dimension of the space S) of $\mathbf{o}$ from $\mathbf{p}$ need not actually take place, because the coordinates of $\mathbf{o}$ are all zero in such a basis. Then the plane equation (in three dimensions) becomes the familiar coordinate form of a plane equation

$$n_1 x_1 + n_2 x_2 + n_3 x_3 + d = 0$$

where $(n_1, n_2, n_3)^{\mathrm{T}}$ are the coordinates of the normal vector and $(x_1, x_2, x_3)^{\mathrm{T}}$ are the coordinates of $\mathbf{p}$ relative to $\mathbf{o}$. Omitting the subtraction of the origin when its coordinates are known to be zero is one optimization discussed in the next chapter.

4.1.2 Intersection point of m planes in m-dimensional space

The intersection point of m (hyper)planes in m dimensions is found by solving the m equations

$$\mathbf{n}_i^*(\mathbf{x} - \mathbf{q}_i) = 0.$$

By introducing a point $\mathbf{p}$ to be taken as the origin on which to base calculations, these equations become

$$\begin{pmatrix} \mathbf{n}_1^* \\ \vdots \\ \mathbf{n}_m^* \end{pmatrix} (\mathbf{x} - \mathbf{p}) = \begin{pmatrix} \mathbf{n}_1^*(\mathbf{q}_1 - \mathbf{p}) \\ \vdots \\ \mathbf{n}_m^*(\mathbf{q}_m - \mathbf{p}) \end{pmatrix}$$

so that the intersection point is

$$x = \begin{pmatrix} \mathbf{n}_1^* \\ \vdots \\ \mathbf{n}_m^* \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{n}_1^*(\mathbf{q}_1 - \mathbf{p}) \\ \vdots \\ \mathbf{n}_m^*(\mathbf{q}_m - \mathbf{p}) \end{pmatrix}$$

```
1  point in S   pointOfPlanes(S,q[],calcOrg)
2      space S;
3      map   affine S  -> R1 q[dim(S)];
4      point in S calcOrg;      /* Origin to base calculations on */
5  {
6    double d[dim(S)];
7    map  S -> R1 C<dim(S)>;
8    int i;
9
10   for (i=1; i<=dim(S); i++)
11   {
12       d[i] = q[i].lin * (p[i].Dorg - calcOrg);
13       C[i] = q[i].lin;
14   }
15   return(calcOrg + C^-1 * d);
16 }
```

`calcOrg` (corresponding to $\mathbf{o}$ in the above equations) is the origin on which to base calculations. Line 3 declares `q` to be an array of plane equations. The number of elements in this array is equal to the dimension of the space `S`. Similarly, line 6 declares the column m-tuple `d` with m equal to the dimension of `S`. Notice this implies allowing local arrays to be allocated dynamically when the function is called.

The `for` loop in lines 10—14 computes $\mathbf{n}_i^*(\mathbf{q}_i - \texttt{calcOrg})$ for each i while at the same time creating a coordinate system from the plane's covectors. Line 15 computes and returns the intersection point.

No check is made to ensure that no two planes are coplanar or nearly so. One way to do this would be to estimate the condition number of `C` and check that it is not too large before computing $\texttt{C}^{-1}$. More sophisticated tests, based on the accuracy of the input plane equations, could be used to verify the accuracy of the computed intersection point. As written, if the intersection point is poorly defined, a floating point exception may occur when $\texttt{C}^{-1}$ is computed.

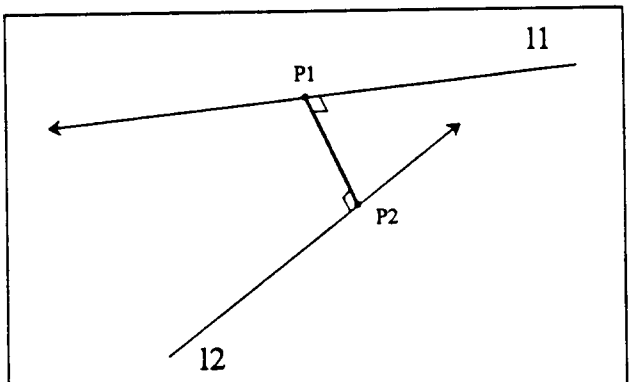


Figure 4.1. The segment of closest approach of two lines

4.1.3 Closest Approach of Two Lines

To find the segment of closest approach for a pair of lines, we note that the segment must be perpendicular to each line (Figure 4.1). If the first line is parameterized as $\mathbf{o}_1 + u\mathbf{v}_1$ and the second as $\mathbf{o}_2 + u\mathbf{v}_2$, the perpendicularity constraints lead to two equations:

$$\mathbf{v}_1^*[\mathbf{o}_1 + s\mathbf{v}_1 - (\mathbf{o}_2 + t\mathbf{v}_2)] = 0$$

$$\mathbf{v}_2^*[\mathbf{o}_1 + s\mathbf{v}_1 - (\mathbf{o}_2 + t\mathbf{v}_2)] = 0$$

where s and t are the parameter values for the first and second lines, respectively, at which the closest approach occurs. After some algebra, we obtain

$$s = \frac{(\mathbf{o}_2 - \mathbf{o}_1)^*[\mathbf{v}_1 - \mathbf{v}_2(\mathbf{v}_1 \cdot \mathbf{v}_2)]}{1 - (\mathbf{v}_1 \cdot \mathbf{v}_2)^2}$$

and

$$t = \frac{(\mathbf{o}_1 - \mathbf{o}_2)^*[\mathbf{v}_2 - \mathbf{v}_1(\mathbf{v}_1 \cdot \mathbf{v}_2)]}{1 - (\mathbf{v}_1 \cdot \mathbf{v}_2)^2}$$

so that the segment has endpoints $P_1 = \mathbf{o}_1 + s\mathbf{v}_1$ and $P_2 = \mathbf{o}_2 + s\mathbf{v}_2$.

The segment of closest approach of the two lines is represented as P_1 and the displacement vector $P_2 - P_1$ as an affine coordinate basis on the line containing the segment of closest approach. After checking that the input lines are not parallel, the function computes the parameter values s and t. Lines 17 and 18 obtain P_1 and P_2 by applying the bases l1 and l2 to the scalars s and t, respectively. Line 19 returns a value by assigning the affine basis to the function name.

```
1  map R1 -> affine M3 closeLines(l1, l2)
2      map R1 -> affine M3 l1, l2;
3  {
4    double vDotu;        /* Cosine of angle between the two lines */
5    double t,s;          /* Parameter values at closest points */
6    point in M3   P1, P2;
7
8    vDotu = l1.lin† * l2.lin;
9    if (abs(1 - vDotu*vDotu) < EPSILON)
10       closeLines(x) = NOTAPOINT + NOTAVECTOR(x);
11   else {
12       s = (l1.Raff - l2.Raff)† * (l2.lin - vDotu * l1.lin);
13       s = s / (1 - vDotu*vDotu);
14       t = (l2.Raff - l2.Raff)† * (l1.lin - vDotu * l2.lin);
15       t = t / (1 - vDotu*vDotu);
16
17       P1 = l1(s);
18       P2 = l2(t);
19       closeLines(x) = P1 + (P2 - P1) * x;
20       }
21 }
```

The separate calculation of P_1 and P_2 could have been omitted and line 19 written

```
closeLines = l1(s) + (l2(t) - l1(s)) * x;
```

4.2 Graphics

4.2.1 Scene Hierarchy

In computer graphics, modeled objects are typically built from combinations of simpler objects. Each combination then becomes a building block for even more complex objects. This process of recursively building complex objects from simple ones is referred to as the *scene hierarchy*[24].

In this set-up, a distinct affine coordinate basis is associated with each object. Each simpler object in a complex object is placed by giving the position of the simple object's basis relative to the complex object's basis.

One way to do this is to give the affine coordinate transformation (matrix plus offset) that describes the simple object's affine basis with respect to the complex object's basis. In this scheme, the data structure that defines a graphical object contains a coordinate basis

```
map R3 -> affine M3  C ;
```

and a list of simple objects that make it up. Each list element contains a pointer to a simple object's definition as well as an affine matrix

```
map R3 -> offset R3  X ;
```

that relates the simple object's basis C_s to the complex object's basis C_c by $C_s = C_c X$.

X can be specified by giving the locations of the simple object's basis vectors and origin in terms of the complex object's basis vectors and origin (see section 2.4). If the embedding is rigid, this relationship may be set by giving an axis of rotation a, and angle theta, the origin o about which rotation occurs, and a final vector translation:

$$X(cp) = C^{-1} * exp(theta * a^{x})(C * cp - o) + o + t;$$

Thus, we apply a transformation to the coordinate basis C, and X gives the coordinates of the this new basis with respect to C.

This scheme makes it easy to refer to geometric quantities in hierarchically defined objects. Points, vectors, coordinate bases, and objects that are represented with these (lines and planes, for example) are referred to directly. All that is required is to execute a statement like the one above for each coordinate basis embedded in another. If a geometric operation is carried out on objects embedded in disparate coordinate bases, a run-time system can perform the coordinate conversion automatically.

The object X is a matrix with n-tuple offset that relates two coordinate frames. It is also possible to relate coordinate frames with a transformation

```
map affine M3 -> affine M3  T ;
```

so that $C_s = TC_c$. T may be set with the statement

```
T(p) = exp(theta * a^x)(p - o) + o + t;
```

The difference in type between X and T makes the distinction between "premultiplication" and "postmultiplication" of coordinate transformations explicit, a difference that is sometimes confusing if all maps are represented as matrices. X is a coordinate map that can be thought of as applying to the coordinates of the points in the simpler object, so successive applications of X induce motions relative to the simpler object's coordinate frame (a "turtle" motion, or motion of the object "relative to itself"). T is a transformation of points, so the motion it induces is coordinate-free, not relative to any coordinate frame (a "global" motion). It is illegal to apply T on the right of a coordinate frame, just as it is illegal to apply X on the left. An explicit distinction is made between these two kinds of motion specification in the language extension, eliminating any confusion about their effects.

4.2.2 Computing Screen Coordinates

As discussed in chapter 2, displaying a 3-D graphical object requires finding its coordinates in terms of a screen coordinate basis; these coordinates describe the 2-D projection of the object to a display device (Figure 2.2). The position of the display screen is given by a point at its center. Its orientation is specified by giving a normal to the screen plane and a vector that, when projected onto the screen's plane, aligns with the screen's y-axis. The following function, `makeProj`, computes the map that takes points in the modeled 3-D space to projected coordinates on a display screen. The display screen's center is given in screen coordinates by `sc`; `scale` is the screen length of an object unit vector. The subroutine `makeScreenFrame` returns a coordinate frame defined by an eye direction and screen projection of the object's y axis (see section 2.7) centered at `center`.

```
1  map affine M3 -> offset R2    makeProj(center,e,f,sc,scale)
2      point in M3     center;
3      vector in M3    e,f;
4      double          sc[2];
5      double          scale;
6  {
7    makeProj(P) = scale * rect(3,2)
8                      * makeScreenFrame(e,f,center)^-1(P) + sc;
9  }
10
11 map R3 -> affine M3   makeScreenFrame(e,f,center)
12     vector in M3    e,f;
13     point in M3     P;
14 {
15   map R3 -> M3 C;
16   e /= norm2(e);
17   C[0] = e;
18   C[1] = (I(M3) - e * e^t) * f;
19   C[1] /= norm2(C[1]);
20   C[2] = C[0] x C[1];
21   makeScreenFrame(x) = C * x + center;
22 }
```

`makeProj` computes screen coordinates for a parallel projection. In perspective projection the size of an object's image on the screen is indirectly proportional to

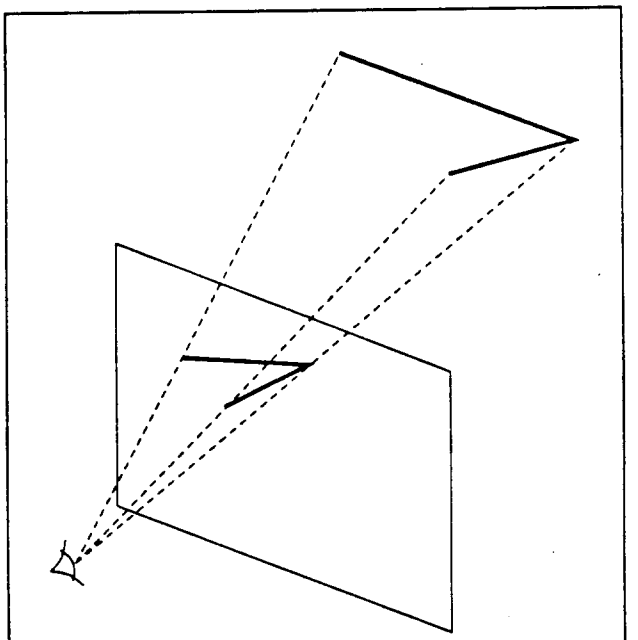


Figure 4.2. Perspective projection onto the viewing screen.

its distance from a viewpoint (Figure 4.2). If x, y, and z are the coordinates of a point to be projected, and x' and y' are the projected coordinates, then

$$x' = \frac{x}{z-d} \quad \text{and} \quad y' = \frac{y}{z-d}$$

where d is the perpendicular distance from the eye to the screen.

Since this transformation is non-linear, it cannot be represented with a language extension object. The following routine accepts a point and returns its projected coordinates. `B` is the coordinate basis returned by **`makeScreenSys`** and `d` is the distance from the screen to the eyepoint. As before, **`s`** is the screen scale factor. **`sc`** are the coordinates of the screen's center.

```
1 double PerspProj(B,d,s,sc,P)[2]
```

```
2       map affine M3 -> R3    B;
3       double                 d,s;
4       double                 sc[2];
5       point in M3            P;
6   {
7     double coords[3];
8     double value[2];
9     double szminusd;
10
11    coords = B^-1(P);
12    szminusd = s / (coords[2] - d);
13    value[0] = coords[0] * szminusd;
14    value[1] = coords[1] * szminusd;
15
16    return value + sc;
17  }
```

Notice that we have not used homogeneous coordinates in either this or the first graphics example. A point in a 3-D space is represented in homogeneous coordinates (relative to a fixed origin) by 4 coordinates $(x\ y\ z\ w)^{\mathrm{T}}$ under the equivalence

$$(x\ y\ z\ w) \sim (x'\ y'\ z'\ w') \Longleftrightarrow \exists \alpha \neq 0 : (x\ y\ z\ w) = \alpha(x'\ y'\ z'\ w').$$

This representation is often used in computer graphics because rotations, translations, and the perspective transformation can all be represented with 4×4 matrices[25]. The resulting coordinate dependent notation is justified in the name of uniformity: the transformation taking an object all the way through the graphics hierarchy to the display screen can be coalesced into a single 4×4 matrix (a final division is still required to find the perspective projection). This representation confuses affine transformations of space (and, in particular, rigid motions) with perspective transformations when in fact the perspective projection occurs only after all affine transformations have been carried out.

Besides being confusing, the 4×4 representation is wasteful. A non-perspective transformation is represented with a 4×4 matrix as

$$\begin{pmatrix} A & t \\ 0 & 1 \end{pmatrix}$$

where A is a 3×3 matrix, and t is the translation 3-tuple. When this matrix is applied to the coordinates of a point $P = (x\ y\ z\ 1)$, each of the three elements of t is multiplied by 1, resulting in three unnecesary multiplications. Four more unneeded multiplications occur when the bottom row is applied to P, for at total of 7 out of 16 multiplications wasted.

In contrast, our method, besides hiding the actual coordinate computations, represents transformations as a 3×3 matrix with a 1×3 translation requiring only the necessary 9 multiplications to apply to a point's coordinates. Even when additions are considered and the computation of $z-d$ is included in the tally (which can be absorbed into the 4×4 formulation), the coordinate-free representation still requires fewer operations.

4.3 Finite Element Assembly

The finite element method models the physical behavior of a continuous medium by breaking it up into a number of pieces of finite size[26]. Equations are developed that describe the behavior of each individual element. These equations are usually assumed to be linear. For example, in the case of structural mechanics, the forces (stresses) generated at nodes of an element are assumed to be linear in the displacements (strains) of those nodes. This leads to the equation

$$f = ku$$

where f represents the forces applied to the element nodes, u the displacement from equilibrium of the nodes, and k a linear map from displacements to forces. k is usually referred to as the element stiffness matrix.

In general, $f = \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix}$ and $u = \begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix}$ where n is the number of element nodes. The forces f_i and displacements u_i may be vectors from a one, two, or three dimensional space, depending on whether the problem under analysis deals

with a one, two, or three dimensional structure. The map k therefor has a block structure

$$k = \begin{pmatrix} k_{11} & \dots & k_{1n} \\ \vdots & \ddots & \vdots \\ k_{n1} & \dots & k_{nn} \end{pmatrix}.$$

Once the element stiffness maps have been determined, they are assembled into a global stiffness map, K, that describes the behavior of the complete structure. This leads to an equation $F = KU$ with $F = \begin{pmatrix} F_1 \\ \vdots \\ F_m \end{pmatrix}$ and $U = \begin{pmatrix} U_1 \\ \vdots \\ U_m \end{pmatrix}$ where m is the total number of nodes in the structure. The correspondence between F_l and U_l with the f_i and u_i for each node is given by the position of the node in the global structure. The same goes for the placement of the k_{ij} into the global map K.

The global map is actually represented as a matrix so that an appropriate numerical matrix equation solver can be called to obtain the global displacements from the applied forces (or vice versa). Therefore the matrix representing each k_{ij} in the global coordinate frame must be found before it is placed into the matrix of K.

Suppose that the problem being solved is a two-dimensional one, and the element with which we are concerned is triangular. Figure 4.3 shows such an element incorporated into a simple truss. We declare

```
space M2 dimension 2 ;
map M2 -> M2 k[3][3] ;
```

to set up the element map k. Each `k[i][j]` is determined by the characteristics of the element. A subroutine that sets the value of `k[i][j]` does so in some coordinate basis; this may as well be the intangible basis. That is, there is an array of matrices

```
real m[2][2][3][3] ;
```

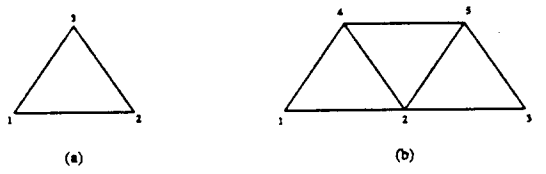


Figure 4.3. (a) A triangular finite element. (b) A simple truss incorporating three triangular elements

and the subroutine sets each `m[][][i][j]` according to the characteristics of the particular element. Then

$$\texttt{k[i][j]} = \texttt{intang(M2)} * \texttt{m[][][i][j]} * \texttt{intang(m2)}^{-1};$$

The advantage to this approach is that the element behavior is specified independently of the global structure. Each element is located in the global structure by giving a coordinate basis with which its axes are to align[27]. This information, along with the correspondence of local nodes to global ones, completely specifies the global structure. It is just as easy to build a structure with disparate elements (even with differing numbers of nodes) as it is to build one from identical elements.

A particular element with map `k` that is to align with the basis `B` in the global stiffness matrix is incorporated by computing

$$\texttt{B}^{-1} * \texttt{k[i][j]} * \texttt{B}$$

and adding this matrix into the appropriate position in `K` for each `i` and `j`. If `K` is declared

```
real K[2][2][m][m] ;
```

(recall that the structure is 2 dimensional), then `k` is incorporated with

$$\texttt{K[p][q]} = \texttt{K[p][q]} + \texttt{B}^{-1} * \texttt{k[i][j]} * \texttt{B} ;$$

The correspondence between p and i (the same as that between q and j) is determined by the correspondence of the element nodes to the global nodes.

It turns out that in nearly all finite element problems, the local stiffness map is symmetric in the sense that k[i][j] = k[j][i]. This implies that the global stiffness matrix is also symmetric. An efficient implementation of the language extension must take account of symmetric operators to reduce storage requirements and avoid unnecessary computation of redundant matrix elements.

4.4 Relativity

In relativity, space is considered to be four dimensional, with three space dimensions and one time dimension. Such a situation is easily described in the language extension:

```
space M3 dimension 3 ;
space time dimension 1 ;
space spaceTime ( M3,time ) ;
```

The space spaceTime must be endowed with the Lorentzian inner product to give it the geometry of relativistic space-time. This is done using the innerp function:

```
innerp(spaceTime) = [r [d I(R3) , zero(R3) ]d,
                       [d zero(R3)† , -1   ]d ]r ;
```

This accomplished, many familiar operations of special relativity are easily carried out. For instance, an event may be specified by giving its position and the time at which it occurred relative to the "laboratory" (the intangible) frame:

```
e = intang(affine spaceTime) * [r x,y,z,t ]r ;
```

The displacement d between two events e1 and e2 is found as

```
d = e1 - e2 ;
```

d's invariant interval is simply $\mathtt{d}^\dagger\mathtt{d}$. If the invariant interval is positive, the displacement is *spacelike*, meaning the two events could have occurred at the same time (in some inertial frame). If the interval is negative, the displacement is *timelike*, indicating that the events could have occurred at the same place. If it is zero, the events can be separated only by a light beam travelling between them.

Suppose v is a column array of three coordinates representing a particle's measured velocity in the lab frame. (assume that our units are chosen with the velocity of light, c, equal to one, so that $\|\mathtt{v}\| = 1$ implies the particle is a photon). Then the following code fragment produces a matrix M that converts displacement coordinates in the lab frame to coordinates in the moving frame and uses this matrix to define the value of a moving basis B.

```
1  real v[3];
2  real M[3][3];
3  real u[3];
4  real g;
5
6    u = v / norm(v);
7    g = sqrt(1 - norm(v)**2);
8    M = [r [d I(R3) + u * u† * (g-1) , -g * v ]d,
9         [d        -g * v†          ,    g   ]d ]r ;
10   B = intang(spaceTime) * M ;
```

Line 6 makes u a unit 3-tuple in v's direction. Line 7 assigns the ubiquitous value $\sqrt{1-\frac{v^2}{c^2}}$ ($c = 1$ in this case) to g. Lines 8 and 9 use this information to compute M, and line 10 produces the desired basis[22]. To find the coordinates cd of the displacement in the moving frame, we simply write

```
cd = B^-1 * d ;
```

This leads to the usual relativistic effects of Lorentz contraction, time dilation, and change of simultaneity that occur when space-time events are coordinitized in differing inertial frames[28].

These constructions form the foundations for more complex computations in

relativity[29]. For instance, suppose a particle with charge q is moving through an electromagnetic field with velocity v as measured in the lab frame. The particle's proper velocity, n is constructed by

```
n = intang(spaceTime) * [r v , 1 / sqrt(1 - v†* v) ]r ;
```

Now suppose that the electric field at a point measured in the lab frame is E and the magnetic field is B. Then the electromagnetic field tensor F is constructed as[30]

```
F = intang(spaceTime) * [r [d B^x , -E ]d,
                         [d E†,  0 ]d ]r * intang(spaceTime)^-1;
```

This map takes velocities to forces, so that

```
k = q * F * n ;
```

sets k to the proper force (also called the Minkowski force) exerted by the field on the charged particle. k's first three coordinates in some inertial frame give the apparent force on the particle as measured in that frame, while the last component gives the proper power (the rate at which energy increases) instantaneously delivered to the particle by the field.

4.5 Robot Dynamics Using Lagrange's Equations

As discussed in the introduction, one way to find the equations of motion of a dynamical system is to employ Lagrange's equations[31]. Let q, $q = (q_1, \ldots, q_n)^{\mathrm{T}}$ and $\dot{q} = (\dot{q}_1, \ldots, \dot{q}_n)^{\mathrm{T}}$ describe the system's position and velocity, respectively, and T denote its kinetic energy. Then

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{q}}\right) - \frac{\partial T}{\partial q} = F^{\mathrm{T}},$$

where F is the tuple of applied forces. The indicated derivatives are carried out in some chosen coordinate system, so position, velocity, energy, and force must all have compatible units.

If the robot arm consists of N segments[32], then the kinetic energy is given by

$$T = \frac{1}{2}\sum_{i=0}^{N-1}(m_i v_i^T v_i + \omega_i^T J_i \omega_i)$$

where m_i is the mass of the ith segment, J_i its inertia tensor, and v_i and ω_i are its linear and angular velocity. Assume that each joint has two degrees of freedom. one rotational and one translational, so that $q_i = \binom{\theta_i}{u_i}$ and $\dot{q}_i = \binom{\dot{\theta}_i}{\dot{u}_i}$ describe the configuration (joint) positions and velocities ($\theta_i, u_i, \dot{\theta}_i$, and $\dot{u}_i$ are all scalar quantities). Then

$$v_i = \underbrace{(A_0^i\ B_0 \cdots A_i^i\ B_i\ 0\ 0 \cdots 0\ 0)}_{M_i} \begin{pmatrix} \dot{\theta}_0 \\ \dot{u}_0 \\ \vdots \\ \dot{\theta}_N \\ \dot{u}_N \end{pmatrix} \tag{4.1}$$

and

$$\omega_i = \underbrace{(C_0\ 0 \cdots C_i\ 0\ \ 0\ 0 \cdots 0\ 0)}_{N_i} \begin{pmatrix} \dot{\theta}_0 \\ \dot{u}_0 \\ \vdots \\ \dot{\theta}_N \\ \dot{u}_N \end{pmatrix} \tag{4.2}$$

for appropriate vectors (in the three-dimensional velocity space) A_i^j, B_i and C_i. The maps N_i and M_i are maps taking configuration velocity coordinates to three-dimensional linear and angular velocities.

If

$$\dot{x} = \begin{pmatrix} \dot{\theta}_0 \\ \dot{u}_0 \\ \vdots \\ \dot{\theta}_N \\ \dot{u}_N \end{pmatrix},$$

then $T = \dot{x}^T W \dot{x}$, $W = \sum W_i$ with

$$W_i = \frac{1}{2}(m_i M_i^T M_i + N_i^T J_i N_i). \tag{4.3}$$

Lagrange's equations become

$$W\ddot{q} + \dot{W}\dot{q} - \dot{q}^* \frac{\partial W}{\partial q}\dot{q} = f_{applied}$$

where $\ddot{q}$ are the configuration accelerations.

Obtaining Lagrange's equations is thus a matter of computing derivatives of W, found by differentiating (4.3) and summing. The derivatives of M_i and N_i are found by differentiating the vectors A_i^j, B_i and C_i. These are computed by noting that the vectors on which they depend (joint axis, vectors from joint to current and previous centers of gravity) have constant coordinates in the appropriate coordinate frame.

Suppose $\mathbf{v}$ is such a vector, F_i the linear coordinate basis in which its coordinates are constant, so that $\mathbf{v} = F_i(\bar{v})$; then $\dot{\mathbf{v}} = \dot{F}_i(\bar{v})$. For an angle θ_i and an axis with constant coordinates $\bar{a}_i$, in F_{i-1},

$$F_i(\bar{v}) = F_{i-1}(\bar{v}) \exp(\theta_i \bar{a}_i^x). \tag{4.4}$$

Application of the chain rule yields

$$\dot{F}_i(\bar{v}) = \dot{\theta}_i F_{i-1} \bar{a}_i^x \exp(\theta_i \bar{a}_i^x)(\bar{v}) + \dot{F}_{i-1}(\bar{v}) \exp(\theta_i \bar{a}_i^x) \tag{4.5}$$

giving a recursion for F_i. The same procedure works for each of the partial derivatives

$$\frac{\partial \mathbf{v}}{\partial q_i} = \begin{pmatrix} \frac{\partial \mathbf{v}}{\partial \theta_i} & \frac{\partial \mathbf{v}}{\partial u_i} \end{pmatrix}$$

of a vector $\mathbf{v}$.

The left hand side of Lagrange's equations has been determined; all that remains is to compute the forces due to gravity before adding in the applied forces to complete the right hand side.

The program to find the equations of motion begins by defining the spaces of interest:

```
1  #define N  N        /* Number of joints */
```

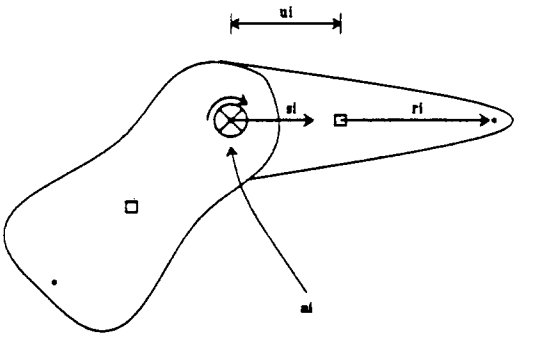


Figure 4.4. Vector quantities pertaining to robot arm segment i.

```
2  #define M    2          /* Number of degrees of freedom/joint */
3  #define THETA 1
4  #define U 2
5
6  space joint = RM ;
7  space jVelocity = RM ;
8  space config = joint**N;
9  space cVelocity = jVelocity**N;
10 space M3 dimension 3;
11 space TM3 dimension 3;
```

Because the joint positions and velocities are most conveniently expressed as a set of scalar quantities, the joint and velocity spaces are made to be M-tuple spaces. M3 is the space in which the arm is actually embedded, and TM3, velocity space, is the tangent space to M3 at a fixed origin.

Before computing the equations of motion for a particular value of $(q, \dot{q})$. the geometry of the robot arm must be specified by giving the placement of pertinent vectors in the appropriate coordinate bases; physical properties are specified by masses, inertia tensors, and the relative locations of centers of mass. These quantities are noted in Figure 4.4. $\mathbf{a}_i$ is the axis of joint i (about which rotation

occurs), $\mathbf{s}_i$ is the unit vector pointing from joint i to segment i's center of gravity (joint translation occurs along this direction). The distance from the joint to the center of gravity along $\mathbf{s}_i$ is given by the configuration variable u_i. $\mathbf{r}_i$ gives the displacement from the segment i's center of gravity to joint $i+1$. The coordinates of these vectors in the ith coordinate basis must be supplied by the calling procedure (denoted in the program by placing a 'c' before the vector's symbol), along with the mass of the ith segment and its inertia tensor (a matrix) expressed relative to the ith basis.

```
12  double ca[N][3], cs[N][3], cr[N][3];
13  double m[N];
14  double J[N][3][3];
```

The other inputs are the configuration positions and velocities:

```
15  double q[N][M];
16  double qDot[N][M];
```

The routine to compute the inertia matrix W and the reaction plus gravity force tuple f begins by declaring the N coordinate bases and their derivatives and partial derivatives (there would be NM partial derivatives for each basis ($M = 2$), but $\frac{\partial F_i}{\partial u_j} = 0$ for all j, so there are only N). Temporary variables are also declared to hold the transformations relating the ith basis to the $i+1$st.

```
17  map R3 -> affine M3  F[N];
18  map R3 -> TM3 FDot[N];
19  map R3 -> TM3 pdF[N][N];
20  map R3 -> R3 T;
21  map R3 -> R3 TDot;
```

Next come the vectors that give the current position of the robot arm joints, segments, and centers of gravity (as determined by q and qDot), and derivatives and partial derivatives of these quantities. Also declared are Mi and Ni and their derivatives and partials, that correspond to N_i and M_i in equation (4.3). W, $\dot{W}$ and the 2N $\frac{\partial W}{\partial q_i}$ are formed by summing combinations of Mi and Ni and their derivatives as determined by equation (4.3). Notice that W and Wdot are $2N \times 2N$ matrices (because jVelocity is the same as R^{2N}, while pdW is an array of $2N$ partial

derivatives of W. (requiring $8N^3$ storage). As for the partial derivatives of $\mathbf{a}_i$, $\mathbf{s}_i$, and $\mathbf{r}_i$, $\frac{\partial \mathbf{a}_i}{\partial u_j} = \frac{\partial \mathbf{s}_i}{\partial u_j} = \frac{\partial \mathbf{r}_i}{\partial u_j} = 0$ for all i and j, so space is only set aside for the partials with respect to θ_j.

```
22  vector in M3   a[N], s[N], r[N];
23  vector in TM3  aDot[N], sDot[N], rDot[N];
24  vector in pda[N][N], pds[N][N], pdr[N][N];
25
26  map cVelocity -> cVelocity   W, Wdot;
27  map cVelocity -> cVelocity   pdW[N][M];
28
29  map cVelocity -> TM3    Mi, Ni;
30  map cVelocity -> TM3   MiDot, NiDot;
31  map cVelocity -> TM3   pdMi[M], pdNi[M];
```

Referring to equations (4.1) and (4.2),

$$A_j^i = \sum_{k=1}^{i} (\mathbf{r}_{k-1} + u_k \mathbf{s}_k) \times \mathbf{a}_j, \tag{4.6}$$

$$B_j = \mathbf{s}_j \quad \text{and} \quad C_j = \mathbf{a}_j.$$

Because storage has already been set aside for the derivatives and partials of $\mathbf{s}_j$ and $\mathbf{a}_j$, only A_j^i and its derivative and partials need be separately computed at each step.

```
32  vector in TM3   Ai[N], AiDot[N], pdAi[N][N][M];
```

With all the variables accounted for, the geometry of the arm is set up by giving the positions of the vectors $\mathbf{a}_i$, $\mathbf{s}_i$ and $\mathbf{r}_i$ and their derivatives in terms of the coordinate frames F_i and their derivatives. This block of code is executed only once. Changing the coordinate frames (as a result of a change in the configuration variables) automatically causes changes in the positions of the arm segments.

```
33 for (i=1; i<=n; i++)
34  {
35    @(F[i]) a[i] = ca[i];
36    @(F[i]) s[i] = cs[i];
37    @(F[i]) r[i] = cr[i];
38    @(FDot[i]) aDot[i] = ca[i];
39    @(FDot[i]) sDot[i] = cs[i];
```

```
40    @(FDot[i]) rDot[i] = cr[i];
41    for (k=1; k<=N; k++)
42    {
43      @(pdF[i][k]) pda[i][k] = ca[i] ;
44      @(pdF[i][k]) pds[i][k] = cs[i] ;
45      @(pdF[i][k]) pdr[i][k] = cr[i] ;
46    }
47  }
```

The routine for computing W and the reaction forces for a particular set of q_i and $\dot{q}_i$ begins by initializing the base coordinate frames and `W`.

```
48 F[1] = intang(M3);
49  Fdot[1] = intang(TM3);
50  for (k=1; k<=N; k++)
51  {
52    pdF[1][k] = intang(TM3);
53  }
54  W = zero(cVelocity,cVelocity);
55  Wdot = zero(cVelocity,cVelocity);
```

After initialization the loop on `i` begins that computes the quantities for the ith arm segment. A_i^j must be computed along with its derivatives and partials. This is done by adding in contributions from the current values for $j < i$ according to equation (4.6). This is possible because

$$A_i^j = A_{i-1}^j + (\mathbf{r}_{i-1} + u_i\mathbf{s}_i) \times \mathbf{a}_j.$$

Differentiating this formula gives

$$\dot{A}_i^j = \dot{A}_i^j + (\dot{\mathbf{r}}_{i-1} + \dot{u}_i\mathbf{s}_i) \times \mathbf{a}_j + (\mathbf{r}_{i-1} + u_i\mathbf{s}_i) \times \dot{\mathbf{a}}_j$$

with a similar formula for $\frac{\partial A_i^j}{\partial \theta_k}$. $\frac{\partial A_i^j}{\partial u_k} = 0$ for $k < i$, while for $k = i$ it is $-\mathbf{s}_i \times \mathbf{a}_i$. After computing A_i^j for $i < j$ and the associated derivatives, A_i^i and its derivatives are found in a similar manner from

$$A_i^i = u_i\mathbf{s}_i \times \mathbf{a}_i.$$

$u_i\mathbf{s}_i$ is computed and stored in the temporary vector `sui`. $\dot{u}_i\mathbf{s}_i = \dot{u}_i\mathbf{s}_i + u_i\dot{\mathbf{s}}_i$ is stored in `suDoti`, and the partial with respect to θ_k is stored in `pdsui[]`.

```
56 for (i=1; i<=N; i++)
57  {
58    sui = q[i][U] * s[i];
59    suDoti = qDot[i][U] * s[i] + q[i][U] * sDot[i];
60    for (k=1; k<=i; k++)
61      pdsui[k] = q[i][U] * pds[i][k];
62    for (j=1; j<=i-1; j++)
63    {
64      Ai[j] += (sui + r[i-1]) x a[j];
65      AiDot[j] += (rDot[i-1] + suDoti) x a[j]
66                    +  (r[i-1] + sui) x aDot[j];
67      for (k=1; k<=i; k++)
68      {
69        pdAi[j][k][THETA] +=
70                          (pdr[i-1][k] + pdsui[k]) x a[j]
71                             +  (r[i-1] + sui) x pda[j][k];
72        pdAi[j][k][U] = zero(TM3);
73      }
74    }
75
76    Ai[i] = sui x a[i];
77    AiDot[i] = sui x aDot[i] + suDoti x a[i];
78    for (k=1; k<=i; k++)
79    {
80      pdAi[i][k][THETA] = pdsui[k] x a[i]
81                                + sui x pda[i][k];
82      pdAi[i][k][U] = zero(TM3);
83    }
84    pdAi[i][i][U] = s[i];
```

Now that the various vectors and their derivatives and partials are known, the maps N_i and M_i and their derivatives are formed.

```
85   for (j=1; j<=i; j++)
86    {
87      Mi[j] = ( Ai[j] , s[j] );
88      Ni[j] = ( a[j] , zero(TM3) );
89
90      MiDot[j] = ( AiDot[j] , suDot[j] );
91      NiDot[j] = ( aDot[j] , zero(TM3) );
92
93      for (k=1; k<=i; k++)
94      {
```

```
95          pdMi[j][k][THETA] = ( pdAi[j][k][THETA] , pds[j][k] );
96          pdMi[j][k][U]     = ( pdAi[j][k][U] , zero(TM3) );
97          pdNi[j][k][THETA] = ( pda[j][k] , zero(TM3) );
98              /* pdNi[j][k][U] = 0 */
99        }
100     }
```

An implicit coordinate conversion takes place as N_i, M_i, and their derivatives are built (for simplicity, I have used parentheses for the range grouping brackets). The constituents of these maps are expressed in various coordinate frames. But the maps themselves are (by default) expressed in the unspecified frame. Therefore, by the rules on coordinate conversions laid out in section 3.9, the constituents' coordinates are found in the unspecified frame and these are stored in the maps.

Now M_i, N_i and their derivatives are used to compute W_i, $\dot{W}_i$, and $\frac{\partial W_i}{\partial q_1}, \ldots, \frac{\partial W_i}{\partial q_n}$. From equation (4.3),

$$\dot{W}_i = \frac{1}{2}[m_i(\dot{M}_i^{\mathrm{T}} M_i + M_i^{\mathrm{T}} \dot{M}_i) + (\dot{N}_i^{\mathrm{T}} J_i N_i + N_i^{\mathrm{T}} J_i \dot{N}_i)]$$

with a similar formula for the partial derivatives. These values are not stored separately, but are added to the totals.

```
101   W += 0.5 * (m[i] * Mi† * Mi  +  Ni† * J[i] * Ni);
102
103    WDot += 0.5 * ( m[i] * (Mi† * MiDot + MiDot† * Mi)
104                    + Ni† * J[i] * NiDot + NiDot† * J[i] * Ni );
105
106   for (k=1; k<=i; k++)
107   {
108     pdW[k][THETA] += 0.5 * ( m[i]
109            * (Mi† * pdMi[k][THETA]  +  pdMi[k][THETA]† * Mi)
110              + Ni† * J[i] * pdNi[k][THETA]
111                 + pdNi[k][THETA]† * J[i] * Ni ),
112
113     pdW[k][U] +=
114           0.5 * ( m[i] * (Mi† * pdMi[k][THETA]
115                                          +  pdMi[k][THETA]† * Mi));
116   }
```

The last step at the end of the ith iteration is to compute the $i+1$st coordinate frame and its derivative and partials as indicated in equations (4.4) and (4.5).

```
117   T = exp(q[i][THETA] * ca[i]^x)
118    TDot = ca[i]^x * T * qDot[i][THETA];
119    F[i+1](x) = F[i].lin(x) * T + F[i].Raff + si + ri;
120    FDot[i+1] = FDot[i] * T + F[i].lin * TDot;
121
122    for (k=1; k<=i; k++)
123      pdF[i+1][k] = pdF[i][k] * T;
124    pdF[i+1][i+1] = F[i].lin * ca[i]^x * T;
125  }
```

After the nth iteration, `W` has been computed and all the quantities are available for computing the reaction forces. Before doing so, the forces due to gravity are computed. `g` is a unit vector pointing in the direction of gravity; `gc` is the gravitational constant. `rki` is a temporary vector for storing the moment arm from the axis of the ith segment to the center of gravity of the kth segment (for $i < k$). It is not hard to show that the angular force due to gravity is the sum of the lengths of the projections of these cross products onto the gravity direction. The linear force is the sum of the masses of the segments following joint i. `Ya[N][]` holds the tuple of forces.

```
126  vector in M3    g;
127  real gc;
128  vector in M3    rki;
129  double Ya[N][2];
130  for (i=1; i<=n; i++)
131  {
132    for (k=i+1; k<=n; k++)
133    {
134      rki = F[k+i].org + r[k]    -    F[i].org) x a[i];
135      Ya[i] += m[k] * ( gc * abs(g^T * rki) , 1 );
136    }
137  }
```

The components of f have all been assembled; all that remains is to compute their sum.

```
138 f = Ya - Wdot * cqDot    +    cqDot^T * pdW * cqDot + fApplied;
```

Here `fApplied` is the tuple of applied joint forces and torques.

Now we have the equations of motion of the robot arm in the form

$$W\ddot{q} = f_{\text{reaction}} + f_{\text{applied}}.$$

This equation can be solved for $\ddot{q}$ to get the joint accelerations. These accelerations can be used to step the configuration velocities, and the velocities to step the configuration positions, obtaining new positions and velocities. Then the whole process of setting up and solving the equations can be repeated, leading to a simulation of the system over time[33][34].

This longer example has shown the power of the language extension to carry geometric concepts from mathematical expression to computer calculation. Each set of computations looks very much like its mathematical counterpart. This correspondence makes the program easy to understand, debug, and modify in spite of its moderately complex calculations.

Chapter 5

Implementation

Geometric objects must be represented as coordinate arrays in the computer's memory. For a program making a geometric computation to reflect the geometry of the problem rather than the configuration of the computer's memory, a set of routines must be provided to manipulate the coordinate representations of objects. In our case, the objects are either maps or points, and the manipulations are described in section 3.3. In addition, checks on the semantics of geometric expressions and optimizations on the coordinate representations of these expressions may be made at compile time.

Implementation is thereby divided into two parts: a compiler to convert geometric expressions to coordinate manipulations, and a run-time package to carry out those manipulations. All the objects and operations specified in chapter 3 could be implemented by a set of procedures that manipulate representations of geometric objects while making run-time checks on operands' types. The manipulations include geometric object memory management, coordinate conversions, and coordinate versions of geometric operations. The checks enforce semantic restrictions on the spaces to which a manipulated set of objects refers. Most of these checks, as well as some of the manipulations, may be performed ahead of time by a compiler.

We shall return to the possibilities of a compiler for geometric declarations

and expressions presently. First, though, we discuss the implementation of the run-time system.

5.1 Representation

A linear map is represented as an array of real coordinates along with a list of pointers to coordinate bases. The number of spaces referred to by the map determines the number of array dimensions and coordinate systems pointers. For instance, if `M3` is a three-dimensional space,

$$\texttt{map M3 -> M3 X ;}$$

is represented as an array `Xa[3][3]` with two coordinate basis pointers, while a vector

$$\texttt{map } \mathrm{R}^1 \texttt{ -> M3 v ;}$$

is represented as an array `va[3]` with one coordinate basis pointer. An n-tuple space requires no coordinate basis pointer, so that

$$\texttt{map } \mathrm{R}^3 \texttt{ -> } \mathrm{R}^3 \texttt{ M ;}$$

is represented as an array `M[3][3]`, while a coordinate basis

$$\texttt{map } \mathrm{R}^3 \texttt{ -> M3 C ;}$$

is represented as an array `Ca[3][3]` with one coordinate basis pointer. If `D` is the basis pointed to, then

$$\texttt{C} = \texttt{D} * \texttt{Ca}.$$

In this way one coordinate basis is represented in terms of another with a matrix that may change at run-time. The intangible basis is indicated by the null pointer.

A point is represented as a singly-dimensioned array of coordinates with a pointer to a coordinate frame. A coordinate frame may also be pointed to by a vector, in which case only the frame's linear part is significant. Affine maps

are represented in the same way as linear maps, but with an additional singly-dimensioned array for each affine or offset linear space referred to by the map. Each extra array represents a point or vector offset.

The offset or point's coordinates must represent it in the same basis or frame as the corresponding linear part of the map. Otherwise, two coordinate basis pointers would be required for each offset linear or affine space referred to by an affine map: one for the linear part and one for the offset or point.

5.2 Coordinate Conversions

The set of geometric operations specified in chapter 3 is implemented as a set of operations on coordinate arrays. This representation requires that, when necessary, coordinate transformations from one basis to another be carried out at run-time. Such conversions consist of two parts: finding a chain of coordinate bases that determines the conversion, and carrying out that conversion by applying the transformation to the coordinates.

Finding the chain of coordinate bases from one basis to another means finding a path between two nodes in a tree of coordinate bases with the intangible basis at the root. Each node of the tree points to its parent. Finding a path from one basis A to another B amounts to finding the common parent P (which is at worst the intangible basis).

Each edge in the path from a node to the common parent specifies a coordinate transformation. Label the transformations along the path from A to P $T_1^{AP}, \ldots, T_r^{AP}$, and those from B to P $T_1^{BP}, \ldots, T_r^{BP}$. The change of coordinates from A to P is

$$M_A = T_1^{AP} \circ \cdots \circ T_r^{AP}$$

and that from B to P is

$$M_B = T_1^{BP} \circ \cdots \circ T_r^{BP}$$

so that the change from A to B is

$$M_{AB} = B^{-1}A = (M_B)^{-1} \cdot M_A.$$

Changing coordinates from basis A to basis B is a matter of following the paths to the common parent, concatenating the transformations $T_1^{AP}, \ldots, T_r^{AP}$ and $T_1^{BP}, \ldots, T_r^{BP}$ to form M_A and M_B, inverting M_B, and obtaining M_{AB}.

The transformations may be affine or linear, depending on the type of conversion being carried out. Let $T_i(s) = M_i(x - x_i) + y_i$. Then the concatenation $T_i \circ T_j$ is

$$\begin{aligned} T_i(T_j(x)) &= M_i(M_j(x - x_j) + y_j - x_i) + y_i \\ &= M_iM_j(x - x_j) + M_i(y_j - x_i) + y_i \qquad (5.1) \\ &= M_iM_j(x - x_j + M_j^{-1}(y_j - x_i)) + y_i \qquad (5.2) \end{aligned}$$

Thus $T(x) \equiv T_i \circ T_j(x) = M(x - x_1) + y_1$ with $M = M_iM_j$, $x_1 = x_j$ and $y_1 = M_i(y_j - x_i) + y_i$. We choose (5.1) to compute the concatenation, because (5.2) would require inverting M_j.

Composing two general coordinate conversions requires multiplying two matrices and applying a matrix to an n-tuple (along with a vector addition). Inverting a general conversion requires only a matrix inversion: if $y = T(x) = M(x - x_1) + y_1$, then $x = T^{-1}(y) = M^{-1}(y - y_1) + x_1$.

Usually the x_1 term is not present in the conversion equations, because an affine basis is typically declared

map R^3 -> affine M3 C1 ;

allowing for an offset only in its range. Only if a basis is declared

map offset R^3 -> affine M3 C2 ;

can the x_0 term be present in coordinate conversions, because only in that case is storage set aside for an offset in the domain.

Finally, although concatenating a series of transformations and computing an inverse may be computationally expensive, the procedure frequently reduces to only one concatenation or a single matrix inversion. For instance, if all coordinate bases are directly related to the intangible system, an inversion and concatenation are all that could be required. Although the full generality of computing coordinate conversions is required to handle all cases, the only example in Chapter 4 that actually uses this generality is the scene hierarchy.

5.2.1 Applying Coordinate Conversions

Once the required transformation has been found, it must be applied to the map or the point to carry out the conversion. The conversion affects each part of the map in a different way. We first consider the linear part.

Each component of the linear part of a map is either contravariant or covariant. The coordinates of a contravariant component (think of a vector), when arranged in a column, transform by applying the change of coordinate matrix on the left. A covariant component (think of a covector) transforms by applying the inverse of the change of coordinate matrix to the right of the coordinates arranged in a row. In general, if the representation of a map X is

$$X : \mathrm{R}^{n_1} \times \cdots \times \mathrm{R}^{n_r} \times \mathrm{R}^{m_1\dagger} \times \cdots \times \mathrm{R}^{m_s\dagger} \mapsto V.$$

Then the first r components transform covariantly while the last s transform contravariantly. If the range space $V = \mathrm{R}^p$, then that component transforms contravariantly, while if $V = \mathrm{R}^{p\dagger}$, then it transforms covariantly. A map component is represented by R^q if the map's declaration names a space with dimension q in the corresponding position; $\mathrm{R}^{q\dagger}$ represents a dual space.

The map X is represented as an array

$$\mathtt{X[n_1]\cdots[n_r]\,[m_1]\cdots[m_s]\,[p]}$$

while the linear part of the coordinate transformation is a square matrix

```
M[r][r]
```

where $r = n_i$ or $r = m_i$ or $r = p$ depending on which component is being transformed. If the component transforms contravariantly, then the transformed X, denoted X′ is

$$\mathtt{X}'[\mathtt{i}_1]\cdots[\mathtt{i}_k]\cdots[\mathtt{i}_{r+s+1}] = \sum_j \mathtt{M}[\mathtt{i}_k][j]\mathtt{X}[\mathtt{i}_1]\cdots[\mathtt{i}_k]\cdots[\mathtt{i}_{r+s+1}].$$

If the component transforms covariantly, the formula is

$$\mathtt{X}'[\mathtt{i}_1]\cdots[\mathtt{i}_k]\cdots[\mathtt{i}_{r+s+1}] = \sum_j \mathtt{M}^{-1}[j][\mathtt{i}_k]\mathtt{X}[\mathtt{i}_1]\cdots[\mathtt{i}_k]\cdots[\mathtt{i}_{r+s+1}].$$

A separate coordinate conversion may be needed on each map component to convert the map to its representation in the desired coordinate bases.

A map's affine parts transform as points, vectors, or covectors, depending upon their type. An affine part representing a point (arising from an **affine** space component) transforms as a point: if its coordinates are given by an array $p \in \mathrm{R}^n$ and the coordinate transformation is $T(x) = M(x - x_1) + y_1$, then the transformed coordinates are $T(p)$. A vector offset (arising from an **offset** space component) transforms as a vector or covector according to whether the corresponding component is a named space or its dual. If $v \in \mathrm{R}^n$ gives the coordinates of the offset, then Mv gives the transformed coordinates for a vector, while $M^{-T}v$ gives them for a covector.

5.3 Compilation

The preceding discussion fails to take account of checks to ensure that the operations carried out on geometric objects do make sense. There are also other checks and optimizations for special kinds of objects, such as orthogonal or symmetric matrices. Such checks may of course be performed at run-time by calling subroutines to make declarations and carry out operations, but it is desirable to

have a compiler make them beforehand, saving time and storage at run-time, and detecting errors before the program is run.

The compiler can, in principle, be added to a compiler for any language. Geometric objects occur only in declarations and expressions. The syntax for declarations is expanded to include geometric objects (spaces, maps, and points) and that for expressions to include geometric operations.

Each geometric object is represented at compile time by a list of spaces to which it refers and a corresponding list of coordinate bases. The space information encodes the object's type. The actual values for the coordinate bases of storage may not be known at compile time, but relationships among them may be (c.f. the robot arm dynamics example).

Checking expression semantics is straightforward because the type of any geometric subexpression determines what can be done with it. The rules given in chapter 3 specify what relationships must hold among the objects' types in a geometric operation. These rules also specify the type of the result. Each subexpression recognized by the compiler can thus be assigned a type. If the type of a subexpression is incompatible with the operation being applied to it, an error message is printed and compilation continues (although no code is produced).

If an expression is well-formed, the compiler produces code to implement it. The most basic method for generating code is to simply generate a call to a subroutine that performs the indicated geometric operation. However, a naive approach can yield poor code.

Consider the statement

```
a = b + c ;
```

where a, b, and c $\in R^n$. The quadruple notation for the intermediate code corresponding to this statement is

$$T \leftarrow (\text{add, b,c})$$
$$a \leftarrow T$$

That is, first, the expression `b+c` is recognized and the code for it generated. Then the expression `a =`*exp* is recognized, and the code for it generated. This code is wasteful even if `a`, `b`, and `c` are real numbers, but it is n times worse for n-tuples.

The situation is even worse for composition. Consider the statement

```
M = M * N ;
```

where `M` and `N` are $n \times n$ matrices. Writing this as

$$\begin{aligned} \mathrm{T} &\leftarrow (\text{multiply, M,N}) \\ \mathrm{M} &\leftarrow \mathrm{T} \end{aligned}$$

requires n^2 storage for the temporary matrix T. In fact, only an n-tuple is required as temporary storage if it is used to store each row of `M` in turn as each new row of `M` is being computed. This example shows further that unless expression trees are built and analyzed, common constructions may be inefficient: in this case $n^2 - n$ storage is wasted by the naive approach.

5.3.1 Optimization

These examples suggest constructing an expression tree for each expression, and only generating code after the highest-level expression has been parsed. Expression tree analysis is straightforward to implement, but some compile-time optimizations require data-flow analysis of the program, a more difficult task.

Suppose our program contains the two lines:

$$\begin{aligned} &\mathrm{T} = \exp(\ \text{theta} * \mathrm{s}^{\times})\ ; \\ &\vdots \\ &\mathrm{Z} = \mathrm{T}^{-1}\mathrm{w}\ ; \end{aligned}$$

where `s`, `z`, and `w` are vectors of R^3 and `T` is a `map` R^3 `->` R^3. Computing T^{-1} for a general `T` requires both floating point computation (which may be

ill-conditioned) and storage. However, if T is an orthogonal transformation, no computation nor any storage need be used, because $T^{-1} = T^T$.

The compiler recognizes that T is orthogonal after executing the first statement from expression tree analysis. $s^\times$ is skew-symmetric. Multiplication by the scalar `theta` does not affect skew symmetry. Therefore T is set equal to the exponential of a skew-symmetric operator, which is orthogonal.

We do not know what code intervenes between the first and second statement, but T must be traced through that code at compile time to see if it remains orthogonal. This can be done using the techniques described in [35]. However, another method avoids data-flow analysis. The declaration syntax for maps may be expanded so that particular matrices may be declared orthogonal by the programmer. A matrix so flagged must always remain orthogonal. The compiler can enforce this restriction by checking each expression as it occurs in the program for the presence of the orthogonal matrix. If it occurs, the expression is analyzed to ensure that any matrix assigned to the orthogonal matrix is in fact orthogonal. For instance, the exponential of a skew-symmetric matrix is orthogonal. The product of two orthogonal matrices is again orthogonal, but in general the sum or difference is not.

The same technique of declaring a map to have certain properties also works for other special objects: symmetric and anti-symmetric maps and diagonal matrices. As with orthogonal matrices, the code that the compiler produces to implement a particular operation depends on the special properties of the map, if any. In the case of a diagonal matrix, for instance, only the diagonal elements need be stored or computed. Storing anything in an indexed off-diagonal element is illegal, while the value of any off-diagonal element is automatically zero.

Care is required in assigning special properties to maps that are not coordinate arrays to ensure that the properties are coordinate-free. Diagonality makes sense only for a matrix; a diagonal matrix expresses a particular linear operator only in certain special coordinate bases. Symmetry (or anti-symmetry) is a coordinate-

free property, but the matrix that expresses a symmetric map is only symmetric when expressed in an orthonormal basis. Therefore storing and computing just half the entries in a symmetric map's coordinate representation is only possible if the pertinent bases are known to be orthonormal ahead of time. Bases may change at run-time, and so may the inner product that determines orthonormality. While data-flow analysis could again be used to discover those bases that are always orthonormal, this complexity can be avoided by allowing only coordinate arrays to be assigned special properties.

Another more flexible possibility is to let the programmer declare that certain spaces contain only orthonormal bases. When bases are assigned, the compiler checks that they are indeed orthonormal (the inner product on such a space is not allowed to change). A map on such spaces could be assigned special properties without the possibility of those properties being destroyed in the coordinate representation of the map.

A similar situation arises when considering large sparse matrices. It makes no sense to talk about a sparse linear operator, because sparsity depends on the coordinate basis. But many geometric problems that specify relationships between a number of simple components become sparse matrix problems when the components are assembled (finite element analysis, for instance). It is not a goal of the language extension to provide methods for solving sparse matrix problems, as several systems are available for doing so. However, the extension must provide methods for efficiently building and storing sparse matrices, so that assembled geometric problems can be handed to an appropriate package for solution.

Various methods may be used to encode a sparse matrix. One method is to list the non-zero elements along with their positions in the matrix. Because different sparse matrix representations are appropriate in different situations, there must be a way for the programmer to select one of them. Rather than provide a fixed set of sparse matrix representations that may be inadequate for some applications, it makes more sense to allow the programmer to specify or design his own rep-

resentations. Object oriented languages, such as C++ or Ada, provide a means for doing this. A new type is created (in this case representing a sparse matrix), and the action of operations such as addition and multiplication on objects of this type are specified by overloading these operations. In a LISP-based object-oriented environment, the standard properties would be carried to a special object by inheritance, while special properties would override the standard defaults. The same technique is appropriate for other special matrices that may be of interest in certain applications, such as banded, Toeplitz, or Vandermonde matrices[36].

5.3.2 Run-time Optimization

In applications that allocate geometric objects at run-time, it is worthwhile for the run-time package to provide some optimizations. Consider, for instance, an interactive graphical editor that reads hierarchical object descriptions from a file at run-time. A coordinate frame is allocated for each node in the hierarchy as the object is read in, so nothing can be known at compile time about the relationships between the coordinate frames. The sequence of operations necessary to carry out a coordinate conversion can only be determined at run-time.

In such cases it is useful to cache coordinate conversions so that if the same conversion is required several times in a row, time will not be wasted recomputing it. Such a cache is indexed by a pair of coordinate basis pointers: the basis that a cached transformation converts from and the basis it converts to. When a coordinate conversion is required, the from and to basis pointers are hashed to give a pointer into the cache. Then the cache entry is checked to see if its coordinate basis pointers actually correspond to those desired. If so, the cached conversion is used. If not, the conversion is computed and replaces the old cache value.

This is in contrast to a situation in which the structure of the relationship among coordinate bases is fixed, as when modeling a specific robot arm, for instance. In this case, the sequence of operations required to compute a coordinate conversion can be found at compile time by analyzing those expressions in which

coordinate bases are assigned values in terms of other coordinate bases. This information is used to build the coordinate basis tree at compile time. When a conversion is required, this tree is analyzed at compile time and the appropriate code generated. If the same conversion is required in a different expression, the conversion is stored temporarily before either expression is computed, and the stored value used as required (cf. common subexpression evaluation).

Even though some coordinate conversions can be determined at compile time, the penalty for always determining them at run-time is not very great. Suppose a linear map L is the composition of two linear maps: $L = L_1 \circ L_2$. It may be possible to determine at compile time that the domain coordinate basis of L_1 is the same as the range basis of L_2, so that the generated code will simply perform a matrix multiplication of the representations of L_1 and L_2 without first performing any coordinate conversion. The same determination is made at run time by comparing the basis of storage pointers for L_1's domain and L_2's range. Thus the only inefficiency is a pointer comparison, the cost of which is small compared to that of multiplying two matrices. In general, the cost of determining the sequence of coordinate conversions to convert from one basis to another is small in comparison to the cost of actually concatenating the sequence to find the coordinate conversion. When the relationships between coordinate bases can be found at compile time, it is only worth doing so if the small improvement in program speed can be justified. Such may be the case if vectorized hardware is available that performs matrix operations in about the same time it takes to perform a pointer comparison.

Chapter 6

Experience

6.1 Geometric Editor

A preliminary version of a runtime package providing the language extension constructs was created to support development of the 3-D graphical editor "Jessie"[37]. A user could use the mouse to select, rotate, translate, and scale objects displayed on the screen. Polyhedral objects could be built by linking vertices together into faces or by combining previously defined objects.

The runtime package provides C data structures defining points, vectors, affine transformations, and coordinate frames, as well as a set of routines to manipulate these objects. Vector manipulations include addition, subtraction, and dot and cross products. Other routines apply a transformation to a point or vector and compose transformations. Routines are also provided to create rigid motions given an axis, angle, and origin, as well as to extract the axis and angle of a proper linear rigid motion.

The implementor of *Jessie* found the basic geometry package useful. Most of the code in *Jessie* is concerned with the user interface, reading and writing description files, and building data structures. Geometry pervades these tasks, however, and the package's intuitive interface made geometric calculations straightforward to implement. The implementor was left free to implement and debug the non-

geometric portions of the program without being confused by geometric operations. The simplicity of these operations also carried through to portions of the user interface; a rotation, for example, was specified by selecting an axis (perhaps the edge of a polyhedron) and giving an angle (perhaps defined by a pair of connected edges).

6.2 Compiler

Once the basic features of the language extension were decided on, a compiler for its constructs was developed for C[38]. The compiler is a pre-processor that reads a C program with language extension constructs and converts it to a standard C program. This program is then compiled and linked with a library of runtime routines that perform the geometric operations.

Because a geometric expression may appear anywhere a host language expression may appear and geometric objects may appear within host language data structures, the pre-processor must fully parse the whole program presented to it. The simplest way to achieve this parsing is to adapt the compiler for the host language. Because of my familiarity with C and the UNIX language development tools that implement the C compiler, choosing C as the host language seemed the simplest way to produce a compiler for the language extension.

Other choices for the host language are certainly possible. Object-oriented languages, such as C++, Ada, or even Common Lisp come to mind as likely candidates, because geometric objects can be given the same status as host language objects, and operator overloading can be used to implement the geometric operations. An overriding reason for having a compiler instead of a just a runtime library, however, is to allow geometric expression optimization. None of these object-oriented languages provide a method for expression analysis and optimization at compile time. Therefore using any of them would still require compiler modification. As the C compiler is simpler than the compilers for any of these other languages, C

seemed the better choice.

The pre-processor was built using the existing code for the C compiler. First, the lexical analyzer was modified to recognize the keywords pertaining to the language constructs such as **space**, **dimension**, **map**, **affine** and so on. Some new operators must also be recognized. The characters specifying addition and multiplication of geometric objects are the same as those already present in C, but there are no operators for such things as composition and ˣ. Because C uses almost every character on the keyboard, these operators (and others) were introduced using an escape notation like TeX's[39]:
compose and
x.

Next, the YACC code for the C compiler was stripped of its actions. The actions were replaced with code to simply copy the recognized text back to the output. This has the effect of parsing a C program and reproducing it as it was input, the only changes being in the type and amount of whitespace present. This accomplished, new rules were added to recognize declarations of spaces, maps, and points.

The action taken on recognizing a space declaration is to record the space's dimension information in a space data structure. If the space is a cartesian product space, the data structure also records the list of spaces making up the product. This information is saved for later use in checking the semantics of expressions involving objects declared on these spaces. A space declaration does not cause any C code to be output.

The action taken upon recognizing a map declaration is to store the pointers to the indicated spaces in a map data structure along with flags indicating whether each named space is invoked as linear, offset linear, or affine. The same information is stored for a point declaration. This information is used later during geometric expression analysis (in conjunction with space information) to check geometric expression semantics.

The information also determines the size of a block of storage that will hold a map. For instance, the affine coordinate basis B declared as

```
map R3 -> affine M3 B ;
```

requires 12 real numbers (assuming M3 is a three-dimensional space): 9 for the matrix representing its linear part, and 3 for representing the offset in the range space. One pointer is required to indicate the range coordinate basis in which the three columns of vector coordinates and one column of offset coordinates determine vectors. This pointer will be NULL if that basis is the intangible basis. A single word is also set aside for flags that indicate the type of the map to the run-time package. Since all real numbers used to represent maps are assumed to be double precision, this particular map requires $8 \times 12 + 1 \times 4 + 1 \times 2 = 102$ bytes for its storage. The preprocessor generates a C declaration for each map or point to set aside this storage. In this case the generated declaration is

```
char B[102];
```

. The storage required for other objects is computed in a similar fashion.

In general, the number of real coordinates required for a map's linear part is equal to the product of the dimensions of the named spaces. Each space named as **affine** or **offset** adds a number of coordinates equal to the dimension of the named space. For storage allocation purposes, a point is treated as having no linear part and one **affine** component. Finally, the number of coordinate basis pointers is equal to the number of non-n-tuple spaces.

The only other portion of a program that is affected by the language extension is expressions. New rules were added to the grammar to take account of new operators such as composition and cross product. Expression trees are built as rules are reduced and expressions recognized. If it turns out that an expression (or subexpression) involves no geometric objects, the expression tree is traversed and its text reconstructed and output. Some standard C operators make sense when applied to geometric objects; in particular, the '&' operator can be applied to a

geometric object to yield its address just as with any standard C object.

Expressions involving geometric objects are checked for correctness according to the rules for operators laid out in chapter 3. A badly formed expression generates an error message; an acceptable expression causes output of code that calls run-time package functions to evaluate the expression. In general, the implementation of a binary operator `a` *binop* `b` takes the following form:

```
push(a), push(b), binop(flags)
```

`a` and `b` are pushed onto a "geometry" stack, the storage for which is set aside in an include file prepended to any program converted by the preprocessor. The flags argument tells the `binop` routine about the types of `a` and `b` so that `binop` can easily locate coordinate basis pointers and coordinate data within `a` and `b`. `binop` removes `a` and `b` from the stack and replaces them with `a` *binop* `b`.

There are variations on this theme. If *binop* is assignment, `a`'s address is not pushed onto the stack but is passed to the assignment routine directly instead. If *binop* is multiplication or composition and `a` or `b` is a scalar, then `a` or `b` (respectively) is passed directly to a scalar multiply routine instead of being pushed. In the case of grouping operators, each of the elements of the group is pushed and then the final result is formed from the elements on the stack.

The difficulty with implementing geometric expressions as stack operations is that geometric objects are large so that pushing them is time consuming. This problem is alleviated by performing some simple optimizations when analyzing expressions. The basic optimization is to recognize expressions of the form

`r = a` *binop* `b`

and implement them as a single function call to a routine that that performs *binop* on `a` and `b` and places the result in `r`. In fact, the routine that performs the stack version of *binop* calls the assigning version of the routine, so no new runtime code is required.

This optimization and a similar one for unary operators are the only stack

optimizations performed by the preprocessor. A more sophisticated scheme would recognize expressions of the form

$$\mathtt{r} = \sum \mathtt{a}_i binop_i \mathtt{b}_i$$

and use a routine for each i that accumulated the result of $\mathtt{a}_i binop_i \mathtt{b}_i$ in `r`. Doing so would eliminate most geometry stack operations.

The calls to runtime routines are separated by the `C` comma operator rather than making each call a separate statement. The reason is that geometric expressions can appear anywhere a standard `C` expression can appear. Because `a , b` first evaluates `a` and then yields the value of `b` as result, a geometric expression is carried out as a comma separated list of runtime routine calls. The value of the expression is the value that the last routine returns. Geometric expressions evaluating to a scalar are handled by simply making each runtime routine return a scalar if the operation it is performing produces one.

This use of the comma operator is troublesome in only one instance. Suppose a programmer writes a function that accepts geometric objects as arguments. Each argument is pushed onto the geometry stack before the function is called, forming a comma separated list. The last item in the list is the call to the programmer's function. The problem is that after the function returns, the passed arguments must be popped off the stack. This cannot be achieved by adding a call to the pop routine at the end of the list, because that would destroy any (non-geometric) value returned by the function. The solution is to have the function itself pop the arguments before it returns. But if the function returns implicitly without actually executing a `return` statement, then the preprocessor must evaluate function control flow to discover the last statement and insert the call to the pop routine there. Currently the preprocessor does not perform this analysis. Of course, the astute programmer will pass the address of a geometric object to a function rather than the object itself to save copying it onto the geometry stack.

Incidentally, if the function returns a geometric object, the popping can be done after the function returns. Before the function is called, space is pushed on

the geometry stack for its return value, after which any geometric arguments are also pushed. Then the function is called. After the function returns its value is stored on the geometry stack, so calling the pop routine after the function call cannot destroy the function's value.

In retrospect, it might have made more sense to implement the language constructs as a package for C++ or as a set of classes for a lisp-based object-oriented system. Most of the development time was spent on the language design rather than the implementation. Using a standard object-oriented system would have made it easier to change the implementation as the language design changed as well as simplifying integration with already existing language constructs. As already pointed out, the main disadvantage of implementing the language constructs in an object-oriented environment is the lack of opportunity for compile-time optimization. While compile-time optimization is essential for this application, the time spent on language design and the tedium of implementation as a C preprocessor left little time for inclusion of anything but rudimentary optimizations.

6.3 Runtime Library

The preprocessor produces as output a C program that makes calls to a runtime package of routines that manipulate the blocks of storage that represent geometric objects. The routines are divided into two layers. The higher-level layer is the object layer. Its routines evaluate operations on geometric objects. There are also routines to allocate a geometric object given its type. The lower-level layer is the coordinate layer. It contains routines that manipulate coordinate arrays (matrix multiplication and addition, for example). An object level routine calls coordinate level routines to perform the coordinate computations required to carry out a geometric operation.

The link between the object and coordinate layers is the coordinate basis resolver. When a routine is called to carry out an operation on two or more geometric

objects, the appropriate basis pointers from each object are compared. If they differ, the conversion routine is called to convert the coordinates of all objects to a common basis. The default bases are given by the rules laid out in section 3.9.1, but most routines carrying out geometric operations also accept an optional list of coordinate basis pointers that determine the bases in which the operations are carried out. This list, when specified, implements the @ operator described in section 3.9.1.

The runtime data structure for a geometric object is divided into two parts. The first part is simply a block of coordinate data. The second part essentially duplicates the compiler's information about the object: number of component spaces and the dimension of each one, and whether each component is linear, offset linear, or affine. Pointers to the object's coordinate data (the linear part and any affine or offset parts) are also stored. While this information could all be handed to the runtime package directly by the compiler, the duplication simplified debugging the runtime routines by making geometric objects self-contained at runtime. It also makes it simpler to use the object data structures and associated routines as a library without the compiler.

Finally, the dichotomy between object-level operations and coordinate-level operations simplifies efficient implementation of the runtime package on processors that are optimized for vector operations. Most efficiencies may be taken advantage of in the coordinate-level without affecting the implementation of the object-level or the compiler. Machine dependencies are isolated in the coordinate-layer, making implementation on different machines straightforward. In some cases, manufacturers are already providing efficient implementations of coordinate-level operations[40].

6.4 Pre-processor Results

There is little experience with the pre-processor that has been implemented. Two of the simpler examples from chapter 4 have been successfully compiled into working programs (the line-plane intersection and the intersection of n-hyperplanes). Larger examples have not been tried because not all the language extension features have been implemented in full generality. Further, getting the pre-processor to correctly analyze all C constructs, especially compound data types and function declarations, is a tedious process.

Perhaps more useful than the preliminary pre-processor is the runtime package created to support its output. Routines are provided to declare these objects so that the package can be used independently of the pre-processor (although few runtime checks are performed to ensure that expressions are well-formed). The package was easy to implement once the abstractions of geometric objects were clearly defined. The headers describing each of the runtime package routines are provided in the Appendix.

The routines formed the basis for a simplified version of the dynamics example described in section 4.5. A program was written that simulates the dynamics of a sequence of connected links. Each link is attached to the next with one rotational degree of freedom. The masses and inertia tensors of the links are read from a file. In the first few time steps of the simulation, a force is applied to the first joint, which is anchored at the origin. At each time step, the program calls a subroutine much like the one presented in section 4.5 to set up the equations of motion. Then the equations are solved for the joint accelerations, and these are used to update the current joint velocities and positions. Conservation of energy is crudely enforced by renormalizing the joint velocities at each time step (more sophisticated techniques are possible; see for instance [41]). The simulation runs in real time for up to about 6 links on a Silicon Graphics 4D workstation. The Appendix presents some still sequences from the simulation along with the portion

of the program that sets up the equations of motion.

Chapter 7

Conclusion

Implementing a program that carries out geometric computations has been difficult due to a lack of programming language constructs to support geometric data types. In this thesis I have attempted to remedy this situation by describing geometric quantities in a consistent coordinate-free notation suitable for embedding in an appropriate programming language.

This attempt has been moderately successful. When written in coordinate-free form, calculations correspond to the geometry from which they arose rather than seemingly unrelated coordinate manipulations. Since geometric objects are based on spaces of points and vectors, checks can be made on the semantics of geometric calculations, obliging a programmer to have a clear understanding of the geometry of a particular problem before attempting to program its solution. The resulting clarity and consistency of those portions of a program that deal with geometry make palatable the implementation (or the rewriting) of large CAD and geometric modeling systems.

The power of the language extension coupled with a compiler, however, can only become apparent in complex applications. The preliminary compiler handles only relatively simple geometric computations. Effective support for more complex applications requires support for many more optimizations than are currently implemented. The benefits of a simple and natural appearing program may

be more than offset by the runtime penalties paid for lack of adequate compile time optimization.

The reason that more optimizations were not added to the compiler is that it is hard to decide which ones to add. Each application seems to require its own set of optimizations; yet, to be viable, the compiler must be useful in a variety of applications. A related issue is that of integrating user-defined objects (sparse matrices, for instance) into the language extension. In retrospect, it would have made more sense to add the extension to an language like C++ that allows operator overloading. Doing so would have simplified specifying special kinds of geometric objects and how geometric operations behave when applied to them. It would not, however, have helped in optimization. Perhaps a mechanism similar to that for creating special objects is required for specifying special optimizations such as those that arise when the coordinates of an origin are known to always be zero or when all coordinate bases are known to be orthonormal.

The central benefit of the language extension is to reduce the time a programmer must spend to implement a computer solution to a geometric problem. In spite of clear geometric programming constructs, a programmer must still fully understand the geometry of the problem for which he seeks a computer solution, so that the only time to be saved is in reducing the tedium of writing the repetetive code required to keep track of coordinate representations of geometric objects. In the case of the examples presented in this thesis, this savings was small in comparison to the time required to analyze the geometry of the problem. In more sophisticated applications with large numbers of similar geometric objects the relief from handling coordinate representations may become more apparent, but I have not had the opportunity to verify this.

The language extension's unified framework provides a straightforward way to declare, build, and operate on basic geometric objects. Because of the variety of geometric applications and programming languages in which they may be implemented, it is not so much a particular implementation of the language extension

but this framework that is significant. This thesis gives methods by which the framework can be implemented in any programming language. Considerable effort is required to upgrade or rewrite the existing compiler and associated runtime package before they can be used to program sizable examples. This effort is justified only if the problems of optimization and extensibility are overcome. Only then will it be possible to convince enough potential users to try an implementation of the language extension so that it can be given a fair test.

Bibliography

[1] V.I. Arnold. *Mathematical Methods of Classical Mechanics.* Springer-Verlag, New York, 1978.

[2] S. Pommier. *An Introduction to APL.* Cambridge University Press, Cambridge, England, 1983.

[3] S.G. van der Meulen and M. Veldhorst. *TORRIX: A Programming System for Operations on Vectors and Matrices over Arbitrary Fields and of Variable Size.* Volume 86 of *Mathematical Centre Tracts*, Mathematisch Centrum, Amsterdam, 1978.

[4] *MATLAB Reference Manual.* 1986.

[5] Inc. Adobe Systems. *PostScript Language Reference Manual.* Addison-Wesley, Reading, MA, 1985.

[6] American National Standards Institute. *Computer Graphics - Graphics Kernel System (GKS) Functional Description.* Technical Report ANSI X3.124-1985, American National Standards Institute, 1985.

[7] S.M. Mujtaba and R. Goldman. *The AL User's Manual.* Technical Report CS-79-718, Stanford University, 1979.

[8] Lee R. Nackman, Mark A. Lavin, Russell H. Taylor, Jr. Walter C. Dietrich, and David D. Grossman. *AML/X: A Programming Lanaguage*

for Design and Manufacture. Technical Report RC11992, IBM Research Division, Yorktown Heights, NY, 1986.

[9] Pat Hanrahan. *A Geometric Calculator using Homogeneous Coordinates.* Technical Report, RPI, 1983.

[10] Eberhard Schrüfer. *EXCALC: A system for Doing Calculations in the Calculus of Modern Differential Geometry (User's Manual).* Technical Report, The Rand Corporation, Santa Monica, CA, 1986.

[11] Eberhard Schrüfer, Friedrich W. Hehl, and J. Dermott McCrea. Exterior calculus on the computer: the REDUCE-package EXCALC applied to general relativity and to the Poincaré gauge theory. *General Relativity and Gravitation*, 19(2):197–218, February 1987.

[12] Narain Gehani. *Ada: An Advanced Introduction.* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[13] Bjarne Stroustrup. *The C++ Progrgamming Language.* Addison-Wesley, Reading, MA, 1986.

[14] *LOOPS Reference Manual.* 1986.

[15] Werner Greub. *Linear Algebra.* Springer-Verlag, New York, 2 edition, 1978.

[16] David M. Burton. *Introduction to Modern Abstract Algebra.* Addison-Wesley, Reading, MA, 1967.

[17] Werner Greub. *Multilinear Algebra.* Springer-Verlag, New York, ? edition, 1978.

[18] Michael Spivak. *A Comprehensive Introduction to Differential Geometry.* Volume 1, Publish or Perish, Wilmington, Delaware, 1979.

[19] Victor Guillemin and Alan Pollack. *Differential Topology.* Prentice-Hall, Englewood Cliffs, NJ, 1974.

[20] Shlomo Sternberg. *Lectures on Differential Geometry.* Chelsea Publications, New York, 2 edition, 1983.

[21] Sir Robert Stawell Ball. *A Treatise on the Theory of Screws.* Cambridge University Press, Cambridge, England, 1900.

[22] Stephen Parrott. *Relativistic Electrodynamics and Differential Geometry.* Springer-Verlag, New York, 1987.

[23] Michael Spivak. *A Comprehensive Introduction to Differential Geometry.* Volume 2, Publish or Perish, Wilmington, Delaware, 1979.

[24] J.D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, MA, 1982.

[25] I.D. Faux and M.J. Pratt. *Computational Geometry for Design and Manufacture.* Ellis Horwood, Chichester, England, 1979.

[26] J.N. Reddy. *An Introduction to the Finite Element Method.* McGraw-Hill, New York, 1984.

[27] Irving H. Shames and Clive L. Dym. *Energy and Finite Element Methods in Structural Mechanics.* McGraw-Hill, New York, 1985.

[28] G. Y. Rainich. *Mathematics of Relativity.* John Wiley, New York, 1950.

[29] Joan M. Centrella, editor. *Dynamical Spacetimes and Numerical Relativity.* Cambridge University Press, Drexel University, Oct. 7-11, 1985.

[30] David J. Griffiths. *Introduction to Electrodynamics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[31] Herbert Goldstein. *Classical Mechanics.* Addison-Wesley, Reading, MA, 2 edition, 1980.

[32] M. Vukobratovic and M. Kircanski. *Real-time dynamics of Manipulation Robots.* Volume 3 of *Scientific Fundamentals of Robotics*, Springer-Verlag, New York, 1986.

[33] R. Courant and D. Hilbert. *Methods of Mathematical Physics.* Volume I, Interscience, London, 1953.

[34] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleisher. Elastically deformable models. In *SIGGRAPH*, pages 205–214, ACM, 1987.

[35] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, MA, 1977.

[36] Gene Golub and Charles Van Loan. *Matrix Computations.* Johns Hopkins Press, Baltimore, 1983.

[37] H.B. Siegel. *Jessie: An Interactive Editor for UNIGRAFIX.* Technical Report 85/273, UC Berkeley, 1985.

[38] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[39] Donald E. Knuth. *The TeXbook.* Addison-Wesley, Reading, MA, 1986.

[40] James Demmel. *LAPACK.* Technical Report, Courant Institute, 1988.

[41] Paul M. Isaacs and Michael F. Cohen. Controlling dynamic simulation with kinematic constraints, behavior functions and inverse dynamics. In *SIGGRAPH*, pages 215–224, ACM, 1987.

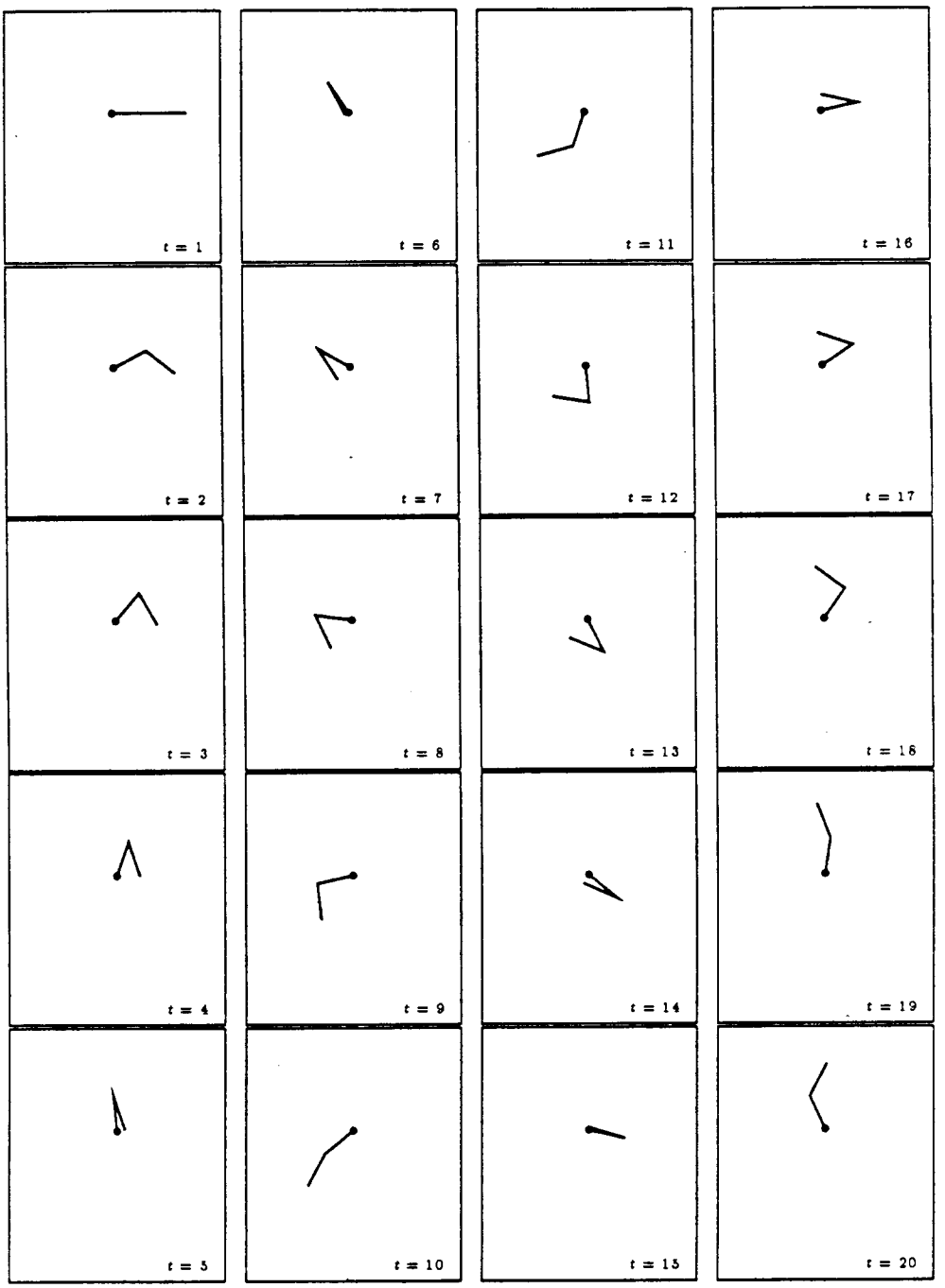
Appendix

Simulation of a Linked Chain

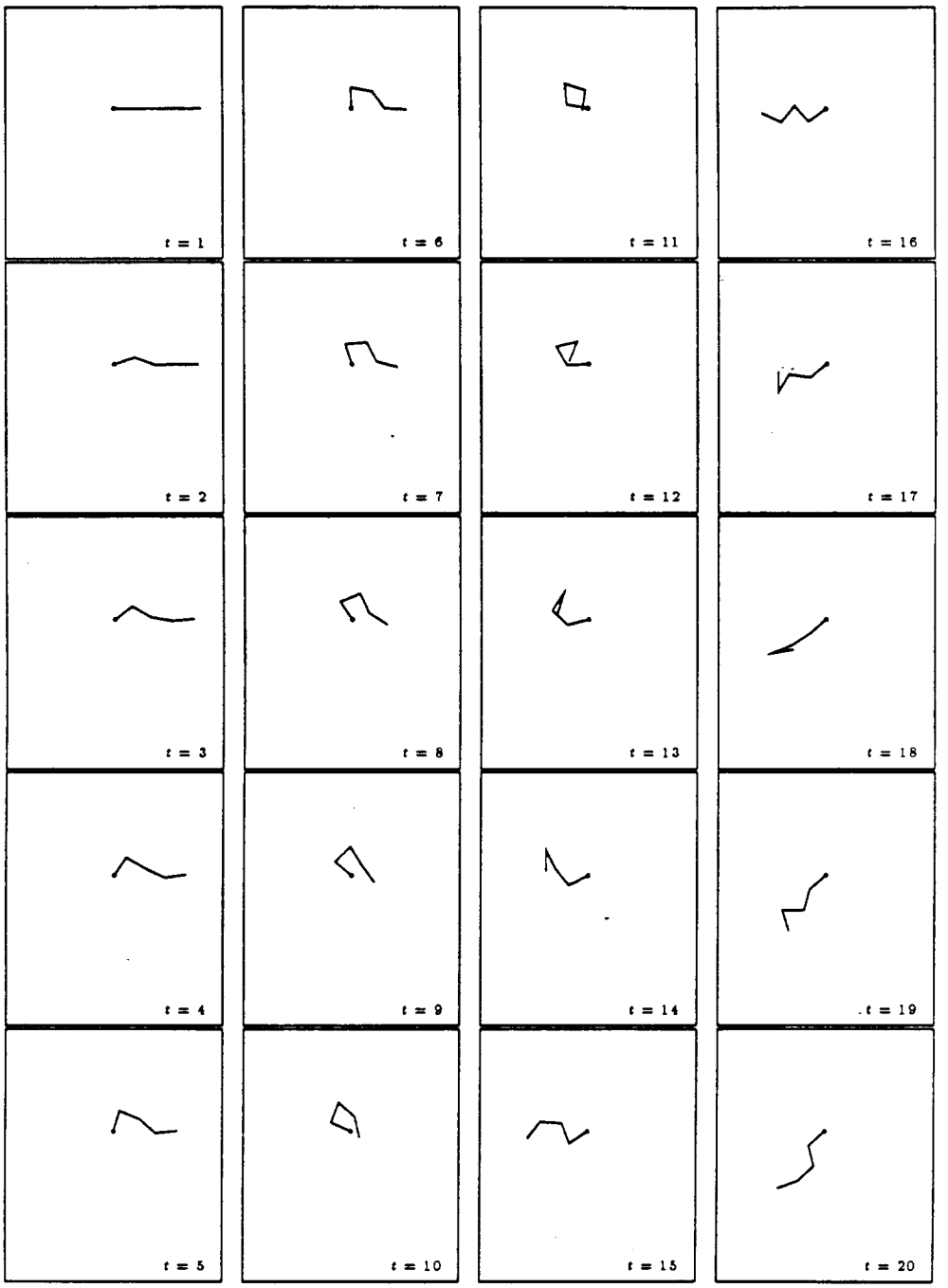
This appendix presents a program that simulates a linked chain as described in chapter 6. Sequenced stills from the simulations are provided on the next pages. All the links have the same mass and intertia tensor (the intertia tensor is that of a relatively thin rod). The exact values of these parameters can be modified by changing a data file that the program reads when it starts up. This file also specifies the initial impulse forces applied at the connection of the first link to the fixed origin.

The program itself appears after the stills. It consists of three sections. The first section declares the various vectors and maps that the program refers to. The second section sets up the equations of motion in a manner analogous to the example of section 4.5 except that there is no translational degree of freedom. The third section (the main program) steps the simulation through time, calling the routine to set up the equations of motion and then solving them for the accelerations. This portion of the program is as simple as possible; more complex schemes could result in more accurate and faster stepping. The significance of the program for this thesis, however, rests with how the equations of motion are set up, and not how they are solved.

For completeness, the headers for the routines that manipulate geometric objects are provided at the end of the program listing.



A simulation of the motion of a two link chain.



A simulation of the motion of a four link chain.

```
/* A geometric object (a map or point) is not much more than a block of
 * storage filled with floating point coordinate data and integer dimension
 * data.
 */

typedef struct gobj GOB;
typedef struct ginfo GIN;

struct gobj {
  int n;
  short flag;
  short pad; /* so as to align on a double word boundary */
  char obj;
};

/* An object info block points to things in an object.  They are filled in by
 * the highest level routines so that the subroutines don't have to search
 * for data in the undifferentiated GOB.
 */

#define MAXCOMPONENTS 4

struct ginfo {
  int n;
  short flag;
  short *flags;
  int *dims;
  GIN **bases;
  double *lin;
  double *(aparts[MAXCOMPONENTS]);
};

/* Flag bits */
/* Flags for a whole map */

#define SYMMETRIC 01
#define ORTHOGONAL 02

/* Flags for individual spaces */

#define ADJOINT 01
#define OFFSET  02
#define AFFINE  04
#define NTUPLE  08

#define AFFOFF (OFFSET | AFFINE)

#define HAS(flag,value) ((flag & value) != 0)
#define SETON(flag,value) (flag=flag|value)
#define DOMAIN(gin) (gin->bases[0])
#define RANGE(gin) (gin->bases[gin->n - 1])
#define DOMAINI 0
#define RANGEI(gin) (gin->n - 1)
```

```
/* Module to declare geometric quantities and set up equations of motion */

#include <stdio.h>
#include "struc.h"

#include "sizes.h"
#define THETA 0

#define abs(x) (((x)>0)?(x):(-x))

/* Variables that describe geometry and physics */

extern double ca[N+1][3],cs[N][3],cr[N][3];/* axis, unit to cog, cog to next */
extern double m[N];                        /* masses */
extern double J[N][3][3];                  /* Inertia tensors */

/* Configuration positions and velocities */

extern double q[N][M];
extern double qDot[N][M];

/* Coordinate frames and transformations relating them */

GIN F[N], FDot[N], pdF[N][N];
GIN T, TDot;

/* Vectors describing the joints' positions and their derivatives */

GIN a[N+1], s[N], r[N];
GIN aDot[N+1], sDot[N], rDot[N];
GIN pda[N+1][N], pds[N][N], pdr[N][N];
/* GIN sui,suDoti,pdsui[N]; */

/* Various important maps, including W and its derivatives, and
 * the maps that form them.
 */

GIN W, WDot, pdW[N][M];
GIN Mi, Ni;
GIN MiDot, NiDot;
GIN pdMi[N][M], pdNi[N][M];
GIN fReact;

GIN Ai[N], AiDot[N], pdAi[N][N][M];

/* Temporary variables needed by some of the computation routines */

GIN tempMap;
GIN tempV1, tempV2;

/* declareThings() - create the objects declared above.  This is simply
 * a set of tedious calls to the declaration routines.
 * Wp and fp are set to point to the linear parts of W and fReact,
 * respectively.
 */

#define NOFLAG (short)0

declareThings(Wp,fp)
     double **Wp, **fp;
{
  register int i, j, k;
```

```
  for (i=0; i<N; i++)
  {
    gMakeMap2(&(F[i]),3,3,NOFLAG,(short)AFFINE);
    gMakeMap2(&(FDot[i]),3,3,NOFLAG,NOFLAG);
    for (j=0; j<N; j++)
      gMakeMap2(&(pdF[i][j]),3,3,NOFLAG,NOFLAG);
  }
  gMakeMap2(&T,3,3,NOFLAG,NOFLAG);
  gMakeMap2(&TDot,3,3,NOFLAG,NOFLAG);

  for (i=0; i<N; i++)
  {
    gMakeVector(&(a[i]),3,NOFLAG);
    gMakeVector(&(s[i]),3,NOFLAG);
    gMakeVector(&(r[i]),3,NOFLAG);

    gMakeVector(&(aDot[i]),3,NOFLAG);
    gMakeVector(&(sDot[i]),3,NOFLAG);
    gMakeVector(&(rDot[i]),3,NOFLAG);

    for (j=0; j<N; j++)
    {
      gMakeVector(&(pda[i][j]),3,NOFLAG);
      gMakeVector(&(pds[i][j]),3,NOFLAG);
      gMakeVector(&(pdr[i][j]),3,NOFLAG);
    }
  }
  gMakeVector(&(a[N]),3,NOFLAG);
  gMakeVector(&(aDot[N]),3,NOFLAG);
  for (j=0; j<N; j++)
    gMakeVector(&(pda[N][j]),3,NOFLAG);


  gMakeMap2(&W,N*M,N*M,NOFLAG,NOFLAG);
  gMakeMap2(&WDot,N*M,N*M,NOFLAG,NOFLAG);
  for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      gMakeMap2(&(pdW[i][j]),N*M,N*M,NOFLAG,NOFLAG);
  gMakeVector(&fReact,N*M,NOFLAG);

  gMakeMap2(&Mi,N*M,3,NOFLAG,NOFLAG);
  gMakeMap2(&Ni,N*M,3,NOFLAG,NOFLAG);
  gMakeMap2(&MiDot,N*M,3,NOFLAG,NOFLAG);
  gMakeMap2(&NiDot,N*M,3,NOFLAG,NOFLAG);
  for (i=0; i<N; i++)
    for (j=0; j<M; j++)
    {
      gMakeMap2(&(pdMi[i][j]),N*M,3,NOFLAG,NOFLAG);
      gMakeMap2(&(pdNi[i][j]),N*M,3,NOFLAG,NOFLAG);
    }

  for (i=0; i<N; i++)
  {
    gMakeVector(&(Ai[i]),3,NOFLAG);
    gMakeVector(&(AiDot[i]),3,NOFLAG);
    for (j=0; j<N; j++)
      for (k=0; k<M; k++)
        gMakeVector(&(pdAi[i][j][k]),3,NOFLAG);
  }

  gMakeMap2(&tempMap,3,3,NOFLAG,NOFLAG);
  gMakeVector(&tempV1,3,NOFLAG);
  gMakeVector(&tempV2,3,NOFLAG);
```

```
  *Wp = W.lin;
  *fp = fReact.lin;
}
```

```
/* setVectors() - sets the values of the pertinent vectors and their
 * derivatives by setting their coordinates in the appropriate coordinate
 * frame.  These vectors pertaining to segment i are then used to set up
 * the i+1st frame and its derivatives.
 */

setVectors()
{
  register int i, k;

  gCopyData(a[0].lin,ca[0],3);
  gMakeRotMat(T.lin,ca[0],q[0][THETA]);
  gSkewCompose(&TDot,ca[0],&T);

  gCopyData(pdF[0][0].lin,TDot.lin,3*3);
  for (k=1; k<N; k++)
    gZeroMat(pdF[0][k].lin,3,3);

  gCopyData(F[0].lin,T.lin,3*3);
  gZeroMat(F[0].aparts[1],3,1);

  gScalarMult(&TDot,&TDot,qDot[0][THETA]);
  gCopyData(FDot[0].lin,TDot.lin,3*3);

  gZeroMat(W.lin,N*M,N*M);
  gZeroMat(WDot.lin,N*M,N*M);
  for (k=0; k<N; k++)
    gZeroMat(pdW[k][THETA].lin,N*M,N*M);

  for (i=0; i<N; i++)
  {
    gSetVector(&(a[i+1]),&(F[i]),ca[i+1]);
    gSetVector(&(s[i]),&(F[i]),cs[i]);
    gSetVector(&(r[i]),&(F[i]),cr[i]);

    gSetVector(&(aDot[i+1]),&(FDot[i]),ca[i+1]);
    gSetVector(&(sDot[i]),&(FDot[i]),cs[i]);
    gSetVector(&(rDot[i]),&(FDot[i]),cr[i]);

    for (k=0; k<=i; k++)
    {
      gSetVector(&(pda[i+1][k]),&(pdF[i][k]),ca[i+1]);
      gSetVector(&(pds[i][k]),&(pdF[i][k]),cs[i]);
      gSetVector(&(pdr[i][k]),&(pdF[i][k]),cr[i]);
    }
    for (k=i+1; k<N; k++)
    {
      gZeroMat(pda[i+1][k].lin,3,1);
      gZeroMat(pds[i][k].lin,3,1);
      gZeroMat(pdr[i][k].lin,3,1);
    }

    if (i == (N-1))
      continue;

    gMakeRotMat(T.lin,ca[i+1],q[i+1][THETA]);
    gSkewCompose(&TDot,ca[i+1],&T);

    gMatMult(F[i+1].lin,F[i].lin,T.lin,3,3,3);
    gCopyData(F[i+1].aparts[1],F[i].aparts[1],3);
    gAddToMat(F[i+1].aparts[1],s[i].lin,1,3);
    gAddToMat(F[i+1].aparts[1],r[i].lin,1,3);
```

```
    gapply(&(FDot[i+1]),&(FDot[i]),&T);
    gMatMult(tempMap.lin,F[i].lin,TDot.lin,3,3,3);
    gScalarMult(&tempMap,&tempMap,qDot[i+1][THETA]);
    gAddToMat(FDot[i+1].lin,tempMap.lin,3,3);

    for (k=0; k<=i; k++)
    {
      gapply(&(pdF[i+1][k]),&(pdF[i][k]),&T);
    }
    gMatMult(pdF[i+1][i+1].lin,F[i].lin,TDot.lin,3,3,3);
  }
}
```

```
/* setUpEqns() - Get the equations of motion.  This is done by doing each
 * segment in turn and adding its contribution into the total.  First we get
 * the map Ai and its derivatives.  This information is used to build Mi
 * and Ni, and these are used in turn to compute Wi and its derivatives.
 */

setUpEqns()
{
  register int i, j, k;

  gZeroMat(Mi.lin,N*M,3);
  gZeroMat(Ni.lin,N*M,3);
  gZeroMat(MiDot.lin,N*M,3);
  gZeroMat(NiDot.lin,N*M,3);
  for (k=0; k<N; k++)
  {
    gZeroMat((pdMi[k][THETA]).lin,N*M,3);
    gZeroMat((pdNi[k][THETA]).lin,N*M,3);
  }
  for (j=0; j<N; j++)
  {
    gZeroMat(Ai[j].lin,3,1);
    gZeroMat(AiDot[j].lin,3,1);
    for (k=0; k<N; k++)
      gZeroMat(pdAi[j][k][THETA].lin,3,1);
  }

  for (i=0; i<N; i++)
  {
    for (j=0; j<i; j++)
    {
      /* Get Ai[j] += (r[i-1] - s[i]) x a[j] and its derivatives */

      subAndCross(&(Ai[j]),&(r[i-1]),&(s[i]),&(a[j]));
      dotAddAndCross(&(AiDot[j]),&(rDot[i-1]),&(sDot[i]),&(a[j]),
                                          &(r[i-1]),&(s[i]),&(aDot[j]));
      for (k=0; k<=i; k++)
        addAndCross(&(pdAi[j][k][THETA]),
                &(pdr[i-1][k]),&(pds[i][k]),&(a[j]));
    }
    /* Now get Ai[i] and its derivatives */

    gCross(&(Ai[i]),&(a[i]),&(s[i]));

    gCross(&(AiDot[i]),&(aDot[i]),&(s[i]));
    gCross(&tempV1,&(a[i]),&(sDot[i]));
    gAddToMat(AiDot[i].lin,tempV1.lin,3,1);

    for (k=0; k<=i; k++)
    {
      gCross(&(pdAi[i][k][THETA]),&(pda[i][k]),&(s[i]));
      gCross(&tempV1,&(a[i]),&(pds[i][k]));
      gAddToMat(pdAi[i][k][THETA].lin,tempV1.lin,3,1);
    }

    /* Now form Mi, Ni and their derivatives */

    for (j=0; j<=i; j++)
    {
      indexSet(&Mi,j,&(Ai[j]));
      indexSet(&Ni,j,&(a[j]));
      indexSet(&MiDot,j,&(AiDot[j]));
      indexSet(&NiDot,j,&(aDot[j]));
```

```
      for (k=0; k<=i; k++)
      {
        indexSet(&(pdMi[k][THETA]),j,&(pdAi[j][k][THETA]));
        indexSet(&(pdNi[k][THETA]),j,&(pda[j][k]));
      }
    }

    /* Now comes the fun part: accumulating W, Wdot, and pdW */

    accumW(&Mi,&Ni,m[i],J[i]);
    accumdW(&WDot,&Mi,&MiDot,&Ni,&NiDot,m[i],J[i]);
    for (k=0; k<=i; k++)
    {
      accumdW(&(pdW[k][THETA]),
              &Mi,&(pdMi[k][THETA]),&Ni,&(pdNi[k][THETA]),m[i],J[i]);
    }
  }
      /* Now we have to compute and add up the reaction forces */

  gMatMult(fReact.lin,WDot.lin,qDot,N*M,N*M,1);
  finishReact(fReact.lin,qDot,pdW);
}
```

```
/* Utility routines called by the set up routines setVectors() and setUpEqns()
 */

/* gSkewCompose(r,v,m) - sets the map r to v^x * m.  v must be a map
 * representing a three-dimensional vector and m must be a map with
 * 3-dimensional range.
 */

gSkewCompose(r,v,m)
     GIN *r;
     double *v;
     GIN *m;
{
  gMakeSkewMat(tempMap.lin,v);
  gapply(r,m,&tempMap);
}

/* addAndCross(r,a,b,c) - compute r += (a+b) x c. */

addAndCross(r,a,b,c)
     GIN *r, *a, *b, *c;
{
  gAdd(&tempV1,a,b);
  gCross(&tempV2,c,&tempV1);
  gAddToMat(r->lin,tempV2.lin,r->dims[0],1);
}

/* dotAddAndCross(r,a,b,c,d,e,f) - compute r += (a+b) x c + (d+e) x f.
 */

dotAddAndCross(r,a,b,c,d,e,f)
     GIN *r, *a, *b, *c, *d, *e, *f;
{
  addAndCross(r,a,b,c);
  addAndCross(r,d,e,f);
}

/* indexSet(r,j,v) - Assumes r is 3 x n.  Sets the jth column of r to v's
 * coordinates (assumes v is 3 x 1).
 */

indexSet(r,j,v)
     GIN *r;
     int j;
     GIN *v;
{
  register int i,d;

  d = r->dims[0];
  for (i=0; i<3; i++)
    (r->lin)[d*i + j] = (v->lin)[i];
}

/* accumW(Mi,Ni,m,J) - accumulates W += m * Mi^T * Mi   +   Ni^T * J * Ni.
 * Recognizes that W is symmetric and so sets W[i][j] = W[j][i].
 */

accumW(Mi,Ni,m,J)
     GIN *Mi,*Ni;
     double m, J[3][3];
{
  register int i,j,k,l;
  register double t;
```

```
  for (i=0; i<N; i++)
    for (k=0; k<=i; k++)
    {
      t = 0.0;
      for (j=0; j<3; j++)
        t += (Mi->lin)[N*j + i] * (Mi->lin)[N*j + k];
      (W.lin)[i*N + k] = ((W.lin)[k*N + i] += m * t);
    }

  for (i=0; i<N; i++)
    for (k=0; k<=i; k++)
    {
      t = 0.0;
      for (j=0; j<3; j++)
        for (l=0; l<3; l++)
          t += (Ni->lin)[N*l + i] * J[j][l] * (Ni->lin)[N*j + k];
      (W.lin)[i*N + k] = ((W.lin)[k*N + i] += t);
    }
}
/* accumdW(dW,Mi,dMi,Ni,dNi,m,J) - accumulates
 * dW += m * (Mi^T * dMi + dMi^T * dMi)  +  (Ni^T * J * dNi + dNi^T * J * Ni).
 * Since W is symmetric, so is dW, allowing the same savings.
 */

accumdW(dW,Mi,dMi,Ni,dNi,m,J)
     GIN *dW, *Mi,*dMi, *Ni,*dNi;
     double m, J[3][3];
{
  register int i,j,k,l;
  register double t;

  for (i=0; i<N; i++)
    for (k=0; k<=i; k++)
    {
      t = 0.0;
      for (j=0; j<3; j++)
      {
        t += (Mi->lin)[N*j + i] * (dMi->lin)[N*j + k] +
                        (dMi->lin)[N*j + i] * (Mi->lin)[N*j + k];
      }
      (dW->lin)[i*N + k] = ((dW->lin)[k*N + i] += m * t);
    }

  for (i=0; i<N; i++)
    for (k=0; k<=i; k++)
    {
      t = 0.0;
      for (j=0; j<3; j++)
        for (l=0; l<3; l++)
          t += (Ni->lin)[N*l + i] * J[j][l] * (dNi->lin)[N*j + k] +
                        (dNi->lin)[N*l + i] * J[j][l] * (Ni->lin)[N*j + k];
      (dW->lin)[i*N + k] = ((dW->lin)[k*N + i] += t);
    }
}

/* finishReact(f,qd,pdW) - computes f = -f + qDot^T * pdW * qDot
 */

finishReact(f,qd,pdW)
     double *f, *qd;
     GIN pdW[N][M];
{
  register int i,j,k;
```

```
  register double t;

  for (k=0; k<N; k++)
  {
    t =0.0;
    for (i=0; i<N*M; i++)
      for (j=0; j<N*M; j++)
        if (abs( ((pdW[k][THETA]).lin)[i*(N*M)+j] ) > 1e-11)
          t += qd[i] * ((pdW[k][THETA]).lin)[i*(N*M)+j] * qd[j];
    f[k] = -f[k] + 0.5 * t;
  }
}
```

```
/* Main module: calls setup routines and solves equations of motion at each
 * time step.  Checks change in energy from last step; if it is too big,
 * the time increment is halved and the the equations recomputed.  If the
 * change is tiny, the time step is doubled (unless it is already at the
 * maximum.  After incrementing the velocities and positions, the velocities
 * are renormalized to enforce conservation of energy.
 */

#include <stdio.h>
#define PI 3.14159265358979
#include "sizes.h"

double ca[N+1][3],cs[N][3],cr[N][3]; /* axis, unit to cog, cog to next */
double m[N];                         /* masses */
double J[N][3][3];                   /* Inertia tensors */

/* Configuration positions and velocities */

double q[N][M];
double qDot[N][M];
static double lastq[N][M], lastqDot[N][M];

/* Everything is in CGS units */

#define DELTAT 0.05
#define THRESH 0.01
#define NSTEPS 100
#define FRICTION 0.001

#define abs(x) (((x)>0)?(x):-(x))

double fa0[NSTEPS];

main()
{
  register int i, j, k;
  double *W,*f;
  double saveW[N*M][N*M];

  double qDDot[N*M];
  double inc[N*M];

  double d,d2;
  double temp;
  double E = 1.0, lastE = 1.0;
  register double factor;
  double sqrt();
  int flag = 0;
  int noAddedEnergy = 0;

  d = DELTAT;
  declareThings(&W,&f);
  setInitial();
/*  dispInit(); */
  for (j=0; j<N*M; j++)
    qDDot[j] = 0.0;

  for (i=0; ; i++)
  {
    for (j=0; j<N*M; j++)
    {
      lastq[j][0] = q[j][0];
      lastqDot[j][0] = qDot[j][0];
```

```
    }
    lastE = E;

    do {

/* fprintf(stderr,"Step: %d E: %g\n",i,E); */

      d2 = d * d / 2.0;
      for (j=0; j<N*M; j++)
        inc[j] = d2 * qDDot[j] + d * qDot[j][0];
      gAddToMat(q,inc,N*M,1);

      for (j=0; j<N*M; j++)
        inc[j] = d * qDDot[j];
      gAddToMat(qDot,inc,N*M,1);

      setVectors();
      setUpEqns();

      E = 0.0;
      for (j=0; j<N*M; j++)
        for (k=0; k<N*M; k++)
          E += qDot[j][0] * W[j*(N*M) + k] * qDot[k][0];

      if ( noAddedEnergy && (abs(E - lastE) / lastE) > THRESH)
      {
        d = d/2.0;
        fprintf(stderr,"Halving d: Now %g. Redoing this step\n",d);
        for (j=0; j<N*M; j++)
        {
          qDot[j][0] = lastqDot[j][0];
          q[j][0] = lastq[j][0];
        }
        flag = 1;
      }
      else {
        flag = 0;
        if ( ((abs(E - lastE) / lastE) < (THRESH / 4.0)) && (d < DELTAT) )
        {
          d = 2.0 * d;
          printf("Doubling d: Now %g. Continuing\n",d);
        }
      }
      if (d > DELTAT)
        d = DELTAT;
    }
      while (flag);

    gMatInv(W,N*M,(short)0);

    if (i<NSTEPS)
      f[0] += fa0[i];
    if ((i>=NSTEPS) || (fa0[i] == 0.0))
      noAddedEnergy = 1;

    gMatMult(qDDot,W,f,N*M,N*M,1);

    if (noAddedEnergy)
    {
      factor = lastE / E;
      factor = sqrt(factor);
      for (j=0; j<N*M; j++)
        qDot[j][0] *= factor;
      E = lastE;
```

```
    }

    for (j=0; j<N; j++)
    {
      if (q[j][0] > 2 * PI)
        q[j][0] -= 2 * PI;
      if (q[j][0] < -2 * PI)
        q[j][0] += 2 * PI;
    }

    if ((i % 30) == 0)
      writeState(i/30);
  }
}
```

```
/* setInitial() - set up the initial conditions and the values of the
 * geometric quantities
 */

#define HS 2.0
#define RS .1
#define MS 1.0
setInitial()
{
  register int i;
  FILE *data;
  char name[20];
  char num[3];

  strcpy(name,"data/links");
  sprintf(num,"%d",N);
  strcat(name,num);
  data = fopen(name,"r");

  for (i=0; i<N; i++)
  {
    fscanf(data,"%lf %lf %lf",&(ca[i][0]), &(ca[i][1]), &(ca[i][2]) );
    fscanf(data,"%lf %lf %lf",&(cs[i][0]), &(cs[i][1]), &(cs[i][2]) );
    fscanf(data,"%lf %lf %lf",&(cr[i][0]), &(cr[i][1]), &(cr[i][2]) );
    fscanf(data,"%lf",&(m[i]) );

    gZeroMat(J[i],3,3);
    J[i][0][0] = m[i] / 2 * RS * RS;
    J[i][1][1] = m[i] / 4 * RS * RS * ( 1.0 + HS * HS / (3 * RS * RS) );
    J[i][2][2] = J[i][1][1];
  }
  for (i=0; i<N*M; i++)
  {
    q[i][0] = 0.0;
    qDot[i][0] = 0.0;
  }
  for (i=0; i<NSTEPS; i++)
    fa0[i] = 0.0;
  i = 0;
  do {
    fscanf(data,"%lf",&(fa0[i]) );
  } while (fa0[i++] != 0.0);
  fclose(data);
}

/* writeState(step) - write out the coordinates of the frame origins into
 * the file data/t<step> in unigrafix format.
 */

writeState(step)
     int step;
{
  FILE *out;
  static char head[] = "data/t";
  char fname[20],num[20];
  register int i;

  strcpy(fname,head);
  sprintf(num,"%d",step);
  strcat(fname,num);

  if ((out = fopen(fname,"w")) == NULL)
  {
```

```
    fprintf(stderr,"Danger: can't open output file %s\n",fname);
    return;
  }
  for (i=0; i<N; i++)
    fprintf(out,"v v%d %lg %lg %lg ;\n",i,
            (F[i].aparts[1])[0], (F[i].aparts[1])[1], (F[i].aparts[1])[2]);
  fprintf(out,"v v%d %lg %lg %lg ;\n",i,
            (F[N-1].aparts[1])[0] + (s[N-1].lin)[0],
            (F[N-1].aparts[1])[1] + (s[N-1].lin)[1],
            (F[N-1].aparts[1])[2] + (s[N-1].lin)[2]);

  fprintf(out,"w x ( ");
  for (i=0; i<N+1; i++)
    fprintf(out,"v%d ",i);
  fprintf(out,");\n");
}
```

```
/* Headers for object-level and coordinate-level routines used by
 * the simulation program.
 */

/* gapply - compute g1 * g2 or g1 \compose g2
 * The restriction on compatibility of g1 and g2 is just that
 * the range space of g2 must exactly match the single domain space of
 * g1. Thus g1 has exactly two spaces;
 * neither g1 or g2 can be a point.
 * Here we assume coordinate conversions have already been done, so the
 * bases of storage better match.
 * r must point to a big enough block of storage to hold the result.
 * It must also have the right type, as its flags are not tampered
 * with.
 */
gapply(r,g1,g2)
     GIN *r;
     GIN *g1,*g2;

/* gMakeObj - make an object by allocating space for the coordinate data
 * and filling in the appropriate pointers in the information block
 * Arguments:
 * g - pointer to information block
 * n - number of dimensions (number of spaces)
 * flag - top level flags (like orthogonal)
 * flags - an array of flags for each space (specifying AFFINE, OFFSET, etc.)
 * dims - an integer array of space dimensions
 */

gMakeObj(g,n,flag,flags,dims)
     GIN *g;
     int n;
     short flag;
     short flags[];
     int dims[];

/* gMakeVector - make a vector
 * v points to the info block, dim is the dimension of the vector,
 * and covariant is TRUE if the vector is a covector, otherwise false.
 * Calls gMakeObj()
 */

gMakeVector(v,dim,covariant)
     GIN *v;
     int dim;
     short covariant;

/* gMakePoint - make a point
 * p points to the info block, and dim is the dimension of the point
 * Call gMakeObj()
 */

gMakePoint(p,dim)
     GIN *p;
     int dim;

/* gMakeMap2 - make a map with one domain space and range space.
 * map points to the info block
 * n,m give the dimensions of the domain and range
 * f0 and f1 tell whether the map is AFFINE, OFFSET, or COVARIANT on
 *   the domain and range spaces.
 * Calls gMakeObj()
 */
```

```
gMakeMap2(map,n,m,f0,f1)
     GIN *map;
     int n,m;
     short f0,f1;
```

```
/* Coordinate-level routines
 */

/* gMakeRotMat - create a rotation matrix given an n-tuple rotation axis
 * and an angle.
 * g points to the matrix result
 */

gMakeRotMat(g,axis,angle)
     double *g;
     double *axis;       /* axis of rotation - assumed to be a unit 3-tuple */
     double angle;       /* angle of rotation */

/* gSetVector(v,f,c) - sets vector v to have coordinates c in frame f.
 */

gSetVector(v,f,c)
     GIN *v,*f;
     double c[];

/* gIdentMat(r,n) - sets the square matrix r to the n x n identity.
 */

gIdentMat(r,n)
     double *r;
     int n;

/* gZeroMat(r,n,m) - set the n x m matrix r to the zero matrix
 */

gZeroMat(r,n,m)
     double *r;
     int n,m;

/* gMakeSkewMat(r,v) - sets the 3 x 3 matrix r to v^x, where v is a 3 element
 * column.
 */

gMakeSkewMat(r,v)
     double *r;
     double *v;

/* gScalarMult(r,m,s) - multiply the linear part of the map m by the scalar
 * s, placing the result in r's linear part. r and m may be the same.
 */

gScalarMult(r,m,s)
     GIN *r, *m;
     double s;

/* gNeg(m) - Negate the linear part of the map m.
 */

gNeg(m)
     GIN *m;

/* gSubtract(r,a,b) - find the linear difference r = a - b.
 */

gSubtract(r,a,b)
     GIN *r, *a, *b;

/* gAdd(r,a,b) - find the linear sum r = a + b.
```

```
 */

gAdd(r,a,b)
     GIN *r, *a, *b;

/* gCross(r,a,b) - the cross product of the 3-D vectors a & b is placed in r.
 */

gCross(r,a,b)
     GIN *r,*a,*b;
```

```
/* Routines to manipulate matrices
 * No provisions are made for optimization in case of special matrices
 */

/* gMatMult - Multiply the matrix a by the matrix b, putting the result in r.
 * a is n x m; b is m x p.
 */

gMatMult(r,a,b,n,m,p)
     double r[],a[],b[];
     register int n,m,p;

/* gSubFromMat - Compute the matrix subtraction mat := mat - sub
 */

gSubFromMat(mat,sub,n,m)
     double *mat, *sub;
     register int n,m;

/* gAddToMat -  Compute the matrix addition mat := mat + add
 */

gAddToMat(mat,add,n,m)
     double *mat, *add;
     register int n,m;

/* gCopyData - Copy amt real numbers from src to dst
 */
gCopyData(dst,src,amt)
     double *dst,*src;
     int amt;

/* gNegateMat - Negate the n x m matrix mat
 */

gNegateMat(mat,n,m)
     double *mat;
     register int n,m;
/* gMatInv - replace the n x n matrix mat with its inverse.
 * If flag is ORTHONORMAL, time is saved by overwriting mat with its
 * transpose.

 */
gMatInv(mat,n,flag)
     double *mat;
     int n;
     short flag;
```