

Symbolic Computer Language for Multibody Systems

Michael W. Sayers

University of Michigan Transportation Research Institute, Ann Arbor, Michigan 48109

Methods are developed for describing and manipulating symbolic data objects that are useful for analyzing the kinematics and dynamics of multibody systems. These symbolic objects include 1) vector/dyadic algebraic expressions, 2) physical components in a multibody system, and 3) program structures needed in a numerical simulation code. A computer algebra language based on these methods encourages the automation of multibody analyses that are versatile and simple because much of the work involved in describing the system mathematically is handled by the algebra system, rather than the analysis formalism. It also handles much of the process of converting symbolic equations into efficient computer code for numerical analysis. The language permits a dynamicist to describe forces, moments, constraints, and output variables using expressions involving arbitrary combinations of unit vectors from different moving reference frames. Kinematics and dynamics analysis algorithms have been programmed that employ these capabilities to analyze complex multibody systems and formulate highly efficient computer source code used for subsequent numerical analysis. A companion paper describes the basic multibody formalism that has been programmed.

Introduction

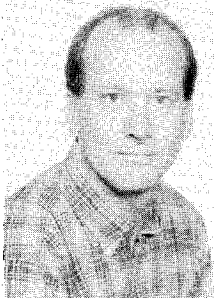
THE manual derivation of the equations of motion for even a modestly complex multibody system is a tedious undertaking that involves considerable algebra and a nagging uncertainty of the correctness of the equations. Further, a considerable programming and debugging effort may be needed to write those equations in a form suitable for numerical solution. Computer algebra has offered a means to reduce the effort and avoid simple algebraic errors, thereby allowing the dynamicist to concentrate on the analysis rather than the algebra. Most of the work reported to date has been done with the MACSYMA language,¹⁻⁴ possibly because it has been available on mainframe computers for over 15 years. Other generic symbolic languages that have been used are FOR-MAC⁵ and REDUCE.⁶ Newer languages with similar capabilities are MAPLE,⁷ MuMath,⁸ and Mathematica.⁹

Even though these mathematical languages are very powerful and have many applications, they are not ideal tools for dynamicists. Because these languages are generic (that is, they are intended for a wide range of scientific applications), they do not automatically associate algebraic expressions with elements in a multibody system. As a result, common kinematic operations cannot be automated unless the analyst engages in an extensive programming effort. Another problem is that the software packages are large and require substantial computing power to be effective.¹⁰ For example, the language MACSYMA consists of about 3000 compiled Lisp functions, accounting for over 300,000 lines of Lisp source code.⁴

At least one symbolic computation language has been developed specifically for interactive use by a dynamics expert.¹¹

With this language, called AUTOLEV, the dynamicist analyzes the mechanical system using the methodology advocated by Kane and Levinson¹² and the computer acts as an assistant that performs most of the algebra. When the analysis is complete, the equations of motion are written into a Fortran program that is ready to compile and run. Because it is specialized for dynamics and kinematics analysis, the software is reported to be simpler to use for this application than other symbolic mathematics computer languages. Another advantage is that it runs on inexpensive personal computers. However, the correctness of the equations is strongly dependent on the skill and thoroughness of the dynamicist, who must attend to many mundane details of the analysis (e.g., using kinematical relations to derive velocities) using AUTOLEV commands.

Further automation has been achieved by computer programs that formulate equations of motion based only on a description of the geometry of the multibody system (e.g., NEWEUL,¹³ SD/FAST,¹⁴ MESA VERDE,¹⁵ etc.). These programs generate subroutines that can be merged into a simulation program. The automation does not come without a price, however. With most of these programs, the dynamicist must describe the system using coordinate systems dictated by the software. Active forces and torques (those doing work) are not included directly, forcing the dynamicist to develop subroutines or equations by hand that are linked with the automatically generated equations. Inclusion of arbitrary constraints (nonholonomic, specified motions, etc.) can require considerable expertise. Simplifications that are made by a human analyst, such as lumping bodies together, or making small-an-



Dr. Sayers received B.S. and M.S. degrees in Mechanical Engineering from the Massachusetts Institute of Technology in 1975 and 1976, respectively, and a Ph.D. in Mechanical Engineering from the University of Michigan in 1990. Since 1976, he has been a researcher at the University of Michigan Transportation Research Institute, conducting research involving the dynamics of ground vehicles, vehicle-road interactions, computer-aided analysis, and the automated modeling of dynamic mechanical systems. He developed the International Roughness Index used worldwide by highway agencies to characterize road roughness, and has designed and authored numerous software packages, most recently with emphasis on symbolic computation methods.

gle approximations, are not done automatically and may not be possible at all. Consequently, the equations can be overly complicated.

Computer programs that automatically form equations of motion of multibody systems, either numerically or symbolically, incorporate a formal analysis process called a multibody formalism. In developing formalisms, most dynamicists have taken it upon themselves to specify the analysis method in such complete detail that it can be programmed numerically in existing computer languages^{16,17} or symbolically using rudimentary computer algebra.¹³⁻¹⁵

Rather than developing a complicated method that can be programmed in existing languages, an alternative approach is to 1) design a new computer language that includes symbolic operations relevant to the analysis of multibody systems, and then 2) devise simpler and more versatile multibody formalisms that can be programmed in the new language.

This paper describes the design of such a language. In this language, three aspects of the system are represented in symbolic form as computer data objects: 1) vector and dyadic algebra expressions, 2) components of the multibody system (bodies, forces, etc.), and 3) pieces of computer code that go into a numerical simulation code.

The methods described in this paper have been programmed in Lisp and are part of a software package called AUTOSIM, developed at the University of Michigan to automatically generate simulation codes for multibody systems. Although the software was developed on an Apple Macintosh, it runs on any machine that supports the Common Lisp language.¹⁸ The multibody formalism presently used in AUTOSIM is based on tree-topology systems. The basics of the formalism are described in a companion paper,¹⁹ and extensions that allow AUTOSIM to handle nonholonomic constraints and kinematical closed loops are described in Ref. 20.

Notational Conventions

Bodies in the multibody system are designated by plain capital letters, e.g., body A , body B . The inertial reference is called N . Points are designated by capital letters that often have subscripts. Origins of coordinate systems are always written with a subscript zero (e.g., B_0). When discussing bodies in the system, the current (generic) body under consideration is called B . Its movements are defined with reference to another body in the system, called the parent of B , and designated A . (The parent A can be either another body or N .) The configuration of the multibody system when all generalized coordinates are zero is called the nominal configuration.

Vectors are written with bold type. Unit vectors that are parallel with axes in coordinate systems are written with lowercase letters that are the same as the body in which the unit vector is fixed, and subscripted with an index of 1, 2, or 3. For example, the three directions of the coordinate system of B are the unit vectors b_1 , b_2 , and b_3 . Other unit vectors used to define directions of interest are written with the letter d . Position vectors are written with the letter r , superscripted with the names of the endpoints of the vector. For example, a vector connecting the origin of $B(B_0)$ to its mass center (B^*) is $r^{B_0B^*}$. A similar convention is used for velocity, except 1) the letter v is used, and 2) only one point is contained in the superscript, e.g., v^{B_0} .

Names of computer data types are written in the Helvetica typeface, e.g., indexed-sym. Formal arguments to computer procedures and names of slots in structures (defined later) are shown in italics.

Symbolic Computation

Symbolic computation is used to derive expressions when the values needed for numerical computation are not known. Arithmetic operations are not performed on the symbolic expressions, but are used to build new expressions that can be applied later. That is, the arithmetic operations are deferred. For example, given an equation $A = 2^*(B + C) - D$, where B ,

C , and D are unknown, the expression $2^*(B + C) - D$ is stored and associated with the symbol A . Later, when values are supplied for symbols B , C , and D , a value for A is calculated.

An example of a symbolic computation program is a compiler. Translating procedures from a high-level language such as Fortran to a low-level language such as machine code involves symbol manipulation and the generation of instructions to perform operations. One way to view a symbolic multibody analysis program is as a compiler: the high-level language is the input from the dynamicist, and the low-level language is a target language such as Fortran.

The reason that symbolic computation can be very useful for developing efficient equations of motion is that knowledge about some of the terms can be used to simplify equations. For example, the equation $A = 0^*(B + C) - D$ can be simplified once and for all to $A = -D$. In addition to pure algebraic manipulation, equations can be simplified based on engineering judgments if certain terms are known to be numerically negligible. For example, if the symbol X is known to apply to a variable that is very small, the expression $1 - X^2$ can be simplified to unity, and the expression $\sin X$ can be simplified to X .

Basis-Free Vectors and Dyadics

Generic computer algebra languages are unable to automatically manipulate vector and dyadic expressions that involve unit vectors from a variety of moving reference frames. For example, consider a spring between two points P_A , fixed in body A , and point P_B , fixed in body B , as shown in Fig. 1. The vector connecting points P_A and P_B can be written simply as

$$r^{P_A P_B} = L_2 b_1 - L_1 a_1 \quad (1)$$

Let the distance between the points be designated x , where

$$x = |r^{P_A P_B}| = (r^{P_A P_B} \cdot r^{P_A P_B})^{1/2} = (L_1^2 + L_2^2 - 2L_1 L_2 a_1 \cdot b_1)^{1/2} \quad (2)$$

An expression for the dot product $a_1 \cdot b_1$ can be formulated if information is available concerning how the bodies associated with these unit vectors are related kinematically. From the figure, we can deduce that the dot product is the sine of the angle between the two bodies. However, if such information is not available, all that can be done symbolically is to generate an expression such as $a_1 \cdot b_1$.

In most existing computer algebra languages, vectors are handled as three-element arrays in a prescribed coordinate system. If the unit vectors a_1 and b_1 are represented by such arrays, then the dot product is the inner product of the two arrays. Two problems with choosing a coordinate system for each vector expression are 1) the analysis is made more complicated because the coordinate systems must be monitored, and 2) it is not always clear right away which is the best coordinate system to choose. To represent the vector expression of Eq. (1), a dynamicist would probably choose either the

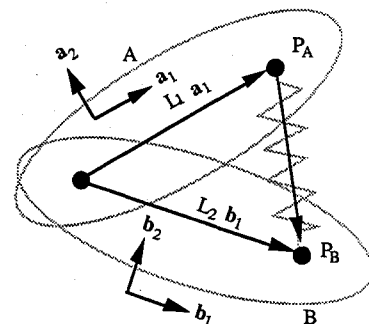


Fig. 1 Use of unit vectors to describe spring.

coordinate system of *A* or *B*. Either way, dot products are likely to be needed between the vector $r^{P_A P_B}$ and vectors described in the other coordinate system. Coordinates that were first transformed to the chosen coordinate system are later transformed back. The process of transforming coordinates back and forth can add to the complexity of the resulting equations. However, if the vector $r^{P_A P_B}$ is stored as written in Eq. (1), a minimal transformation is needed to convert to any coordinate system.

Numerical Efficiency

A simulation code is a computer program that simulates a physical system by numerically integrating differential equations of motion. The integration is performed by using a numerical approximation to integrate the equations over a very small increment of time. The state variables are computed in a simulation run for discrete times that are stepped from a start time to a stop time. Numerical efficiency can be estimated by the number of arithmetic operations needed to compute derivatives of the state variables of the multibody system at each time step. This efficiency derives from several factors. The method used to derive the equations of motion is, of course, a primary factor. The merits of various methods (Newton-Euler, Lagrange, Kanés equations, etc.) have been covered extensively in the literature and will not be repeated here. However, within the scope of a given method, there are several techniques that can be taken by the analyst to simplify the equations and also techniques that can be taken when coding the equations into a computer program. These techniques include the following.

- 1) Terms that are zero for the specific system (but that could be nonzero for a more general formulation) are omitted from the equations.
- 2) Equations are written in factored form, involving products and ratios of sums of terms. For example, the expression $(A + B + C)^2$ requires two additions and one integer power; the expanded form $(A^2 + 2AB + B^2 + 2AC + 2BC + C^2)$ requires five additions, six multiplications, and three integer powers.
- 3) Terms involving products or powers of quantities known to be small are dropped if they are of order 2 or higher. Trigonometric functions of small quantities are replaced with truncated Taylor series expansions.
- 4) Complicated expressions that occur in several places are replaced with intermediate variables. This technique is particularly important for multibody systems because the equations of motion are inherently redundant, even when highly recursive dynamics analysis methods are used.

5) Expressions involving only constants are identified and precomputed as part of the program initialization to avoid the repeating of identical computations at each time step.

6) Unnecessary equations are removed. For example, a term might be introduced that is later multiplied by zero. Equations that compute the term can be safely eliminated.

All of these techniques are independent of the method used to form the equations of motion and can, therefore, be made a part of the computer algebra language.

Representing Symbolic Data

The methods required to manipulate symbolic expressions are derived from the design of the computer data types that are used to represent algebraic expressions and other entities. Given that the AUTOSIM implementation was written in Lisp, Lisp terminology is used in the following descriptions. However, the basic concepts could be applied in other languages.

Lisp includes over 40 types of data objects. In addition, new types are included by the use of structures. Figure 2 shows a hierarchy of data types used in AUTOSIM, as they relate to data types already in Lisp. Each type of data object inherits from the type to its immediate left in the figure. For example, an object of type *cos* is also of types *trig*, *func*, and *expression*. Characteristics of the types *trig*, *func*, and *expression*

are inherited by objects of type *cos*, and most Lisp functions developed to work with objects of type *trig*, *func*, or *expression* work without modification with objects of type *cos*.

The data objects in the figure are shown in four groups related to 1) computer algebra, 2) the multibody system, 3) the numerical simulation program, and 4) additional native Lisp objects. Native Lisp forms are shown in italics, and those used extensively in AUTOSIM are shown in bold italics. Each type of data object is associated with a specialized function used to print that type of object. When the object represents an algebraic expression, it is printed according to the conventions of the target language. The present version of AUTOSIM prints outputs in one of three target languages: Fortran, ADSIM (a simulation language used for real-time simulation with computers made by Applied Dynamics International, Inc.), and RTF (rich text format, used in the Microsoft Word word processor for the Apple Macintosh and the IBM PC). For example, an indexed-sym object that is printed as Z_{10} when the target language is RTF is printed as $Z(10)$ when the target language is Fortran.

Computer Algebra

Expressions in AUTOSIM can represent scalars, vectors, or dyadics. They are composed of numbers and expression structures, whose characteristics are listed in Table 1. The examples show how they are printed in this paper (or when the target language is RTF). Three of the expressions defined in the table are *elementary* types from which *compound* types are built. The elementary types are the **number**, the **sym** (and a subtype, the **indexed-sym**), and the **uv**. The Lisp *structure* object contains a number of variables, called *slots*, that are defined within the structure. Each slot has a name and can be assigned a value. The number of slots and the associated names are defined for each data type. The meta-type *expression* defines a repertoire of qualities associated with all expression types. For example, the *type* slot tells whether an expression is a scalar, vector, or dyadic. The *const-or-var* slot tells whether an expression is a constant or a variable. The *small-order* slot defines an order of smallness for the expression. Nested expressions are implicit in the design of the compound

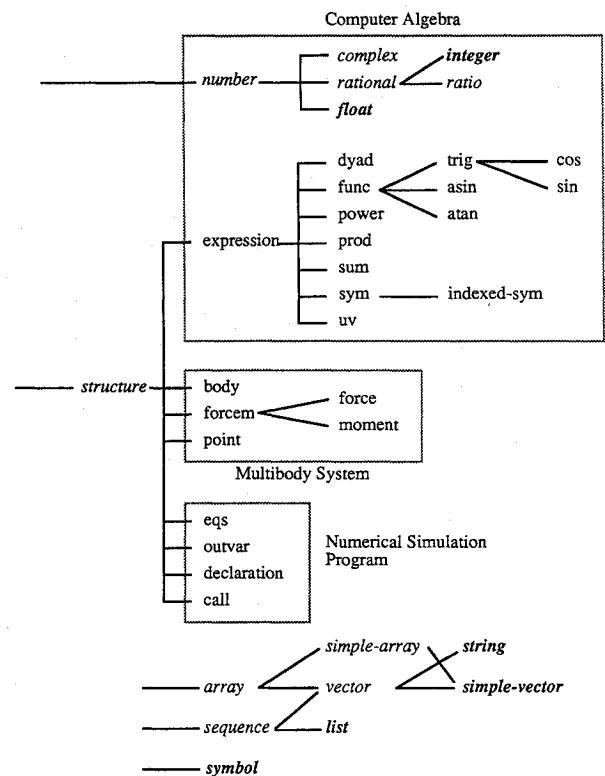


Fig. 2 Hierarchy of AUTOSIM and Lisp data objects.

Table 1 Summary of AUTOSIM expression types

| Type | Primary slots | Definition | Examples |
|-------------|---|---|------------------|
| number | | number | 2, 1/3, -0.3333 |
| expression | <i>type, small-order, sort-code, dxdt, const-or-var</i> | meta-type for all expression objects | |
| sym | <i>symbol, default, hide, exp</i> | symbol for a scalar parameter or variable | M, M_C |
| indexed-sym | <i>i</i> | indexed symbol for a scalar parameter or variable (this is a subtype of sym) | q_2 |
| uv | <i>symbol, body, dot products, cross products</i> | unit vector | a_1 |
| dyad | <i>uv1, uv2</i> | dyad | $(a_1 b_2)$ |
| power | <i>base, exponent</i> | base expression raised to power | $(L_1 + L_2)^2$ |
| prod | <i>coef, factors</i> | product of numerical coefficient and list of expressions | $2M \sin(q_2)$ |
| sum | <i>terms</i> | sum of expressions | $I + M^* L^2$ |
| func | <i>function, args</i> | function that will be written into numerical program | TIRE(F_Z, A) |
| trig | <i>symbol</i> | sin or cos | |
| cos | | cos | $\cos(q_2)$ |
| sin | | sin | $\sin(q_2)$ |
| asin | | arc-sine | $\sin^{-1}(q_2)$ |
| atan | | arc-tangent | ATAN2(X, Y) |

Table 2 Some of the slots in a body

| Slot name | Definition |
|-----------------|--|
| <i>symbol</i> | symbol for user to reference the body |
| <i>name</i> | descriptive name of body |
| <i>parent</i> | parent body in tree topology |
| <i>level</i> | level of the body in tree |
| <i>children</i> | list of bodies that have this body as their parent |
| <i>cm-point</i> | location of mass center |
| <i>mass</i> | mass associated with body |
| <i>inertia</i> | inertia dyadic (with respect to the mass center) |
| <i>abs-w</i> | absolute rotational velocity of the body |
| <i>abs-v0</i> | absolute velocity of the origin of the body |

Table 3 Some of the slots in a point

| Slot name | Definition |
|--------------------|---|
| <i>symbol</i> | symbol used to identify the point |
| <i>name</i> | descriptive name of the point |
| <i>body</i> | body structure in which this point is fixed |
| <i>coordinates</i> | array of three coordinates in the coordinate system of body |

Table 4 Some of the slots in a forcem

| Slot name | Definition |
|------------------|---|
| <i>symbol</i> | symbol used to identify the forcem |
| <i>name</i> | descriptive name of the forcem |
| <i>direction</i> | vector expression that gives direction of forcem |
| <i>magnitude</i> | scalar expression that gives magnitude of forcem |
| <i>body1</i> | body on which forcem acts with + magnitude |
| <i>body2</i> | body on which forcem acts with - magnitude |
| <i>point1</i> | point on line of action of force on body 1 (force only) |
| <i>point2</i> | point on line of action of force on body 2 (force only) |

expression types. For example, the expressions in the list of factors of a prod can be sums, powers, funcs, etc. There are no limits to the level of nesting (other than computer memory). Vectors and dyadics are simply expressions that involve unit vectors (uv objects).

Multibody System

A multibody system is composed of bodies influenced by forces and moments and connected to each other by joints. Each element is represented by a corresponding Lisp structure.

A data structure called a **body** is used to represent each body in the system. Table 2 lists a few of the major slots in a body. Other slots used to support mathematical operations are listed later in other tables. Mass and inertias can be expressions involving variables to account for dynamically varying mass properties. Massless bodies can be used to introduce intermediate reference frames. Also, bodies with zero degrees of freedom can be used to add (or subtract) mass or inertia to an existing body. As will be seen later, some of the slots in a body are set directly by the analyst, such as the *symbol* and the *parent*. Others, such as the slots *abs-w* and *abs-v0*, are set by analyses performed automatically as the body is defined.

A structure called a **point** is used to define a location of interest in a body, such as the origin of the coordinate system, the mass center, an attachment point, etc. Points are defined as needed by the analyst to identify attachment points for forces, as points of interest for output variables, or as center of mass locations for bodies and composite bodies introduced in a dynamics analysis. Table 3 defines the major slots used to define a point. The coordinates of a point can include expressions involving variables to facilitate simple descriptions of forces that act on moving points.

Force-producing elements are represented by objects called **forces**, and moment-producing elements are represented by **moments**. Both types, which inherit from the meta-type **forcem**, are summarized in Table 4. The *point1* and *point2* slots in a force are used to obtain expressions for the moment

applied to a body about its mass center. That is, the moment is defined as

$$T = r^{B^*P} \times f \quad (3)$$

where r^{B^*P} is the position vector going from the center of mass B^* to the point P on the body through which the force passes, and f is the force vector (i.e., the product of the expressions in the *direction* and *magnitude* slots of the force object).

Numerical Simulation Program

In addition to expressions and the multibody system, the numerical simulation program produced as output by AUTOSIM is represented with objects. A sequence of assignment statements is represented by an object called an **eqs**. Some of the sequences that are generated and manipulated are the kinematical equations, the dynamical equations, the trigonometric functions used in other equations, and the output variables. Each equation in an **eqs** is a **sym** (or **indexed-sym**) whose *exp* slot is assigned to an expression. As with other types of objects, the eqs prints in a form appropriate to the target language. Each **sym** is printed in a form similar to the following: *symbol* = *exp*. Information about a variable that will be produced as output by the simulation code is represented by the **outvar** object. It includes a short name, a long name, a generic name, an expression, and units. Before the simulation code is written, the list of **outvars** is processed to ensure that statements are generated to compute all dependent variables specified by the dynamicist. A list of all variables of a certain type (REAL, INTEGER, etc.) that must be declared in a specific subroutine module of the simulation code is represented in a **declaration** object.

Many real-world multibody systems cannot be fully described using only differential and algebraic equations. The behavior of certain components may require semiempirical models that involve table lookups, convoluted numerical algorithms, and even hardware-in-the-loop. Variables defined in these ways are included in the equations of motion through the use of external subroutines. Procedures that return a single variable as a function of one or more arguments are represented with the **func** structure (see Table 1 and Fig. 2). Procedures that return several variables at once are also used. In Fortran, a procedure of this sort is called a subroutine and is invoked with a **CALL** statement. When code is written, it is essential that 1) values needed as inputs to a subroutine are computed before the subroutine is called, and 2) all references to values computed by the subroutine appear after the subroutine is called. External subroutines are represented in AUTOSIM with a type of structure called a **call**. Each call has slots that indicate 1) where the subroutine appears in the simulation code, 2) the name of the subroutine, and 3) its arguments.

Manipulating Symbolic Data

The manipulation of symbolic data to generate efficient numerical analysis algorithms for multibody systems involves algebraic operations, interactions with the multibody system, and automated programming.

Making Expression Objects

Algebraic operations are implicitly performed when a compound expression object is created. For example, a **prod** represents the multiplication of expressions. The functions that create objects check their arguments and create simpler objects when possible. In fact, significant algebraic simplifications are performed in these operations. Table 5 summarizes simplifications that are performed by creator functions.

Most of the small quantity simplifications occur when a **sum** is created. The term with the minimum order of smallness is used as a reference and all other terms are compared to it. Terms whose order of smallness is more than the reference by some threshold are dropped. Normally, the threshold for dropping small terms is 2. However, this value can be modi-

fied if needed to perform alternate analyses that require higher-order terms. For example, AUTOSIM has been used to generate equations needed for a bifurcation stability analysis in which all state variables are small and terms are kept up to the fifth order.²¹ Small simplifications can also occur when a trigonometric object is created, in which case a truncated Taylor series is used.

Care has been taken to ensure that equivalent occurrences of a compound expression always are created the same way. Sums nested within sums and prods within prods are removed. For example, the sum $(A + B) + C$ yields $(A + B + C)$, rather than $[(A + B) + C]$. Terms and factors are sorted when creating **prod** or **sum** structures. For example, the product of B and $A * C$ is $A * B * C$ rather than $B * A * C$. A sign convention for sums is used that results in a repeatable formulation for a given sum. For example, the expression $(-A - B - C)$ would never be generated; instead, that result is always represented as $-(A + B + C)$.

Algebra Operations

Conventional scalar operations such as multiplication and addition are performed by applying simple rules to create new data objects from data in the arguments. Table 6 summarizes the algebra operations. Most of the scalar operators in the table work as would be expected. One unusual operator is the **constant-part** function, which returns zero unless 1) the expression is a constant, or 2) it is a sum with at least one constant term. It is used when symbolically solving for dependent variables to avoid expressions that are likely to become singular. That is, when division is necessary, an expression is preferred that involves a divisor whose **constant-part** is not zero. (This capability is important when dealing with constraints that occur in systems that do not follow a tree topology.²⁰)

Operations that involve unit vectors involve novel interactions between the computer-algebra part of the software and the representation of the multibody system.

The dot-product operation is valid for two vectors, a vector and a dyad, or two dyads. The operation is applied by recursively expanding expressions into multiplications and additions of subexpressions, until an expression is obtained that

Table 5 Simplifications performed by creator functions

| Function | Simplifications ^a |
|----------------|--|
| asin, cos, sin | <ul style="list-style-type: none"> • simplify if argument is the inverse function [e.g., $\sin(\sin^{-1}x) \rightarrow x$] • if argument is a number, evaluate • if small order > 0, return truncated Taylor expansion |
| atan | <ul style="list-style-type: none"> • same simplifications as for asin • if there are two arguments, divide both by greatest common factor [e.g., $\tan^{-1}(ax, ay) \rightarrow \tan^{-1}(x, y)$] |
| power | <ul style="list-style-type: none"> • if base is a power, change exponent • if base is number, evaluate • if base includes small terms, drop if possible |
| prod | <ul style="list-style-type: none"> • if the coefficient is 0, return 0 • if the coefficient is 1 and there is one factor, return the factor • if any numbers are included as factors, remove them from the list of factors and multiply them with the coefficient • if any factors are prods, multiply coefficients and combine lists of factors (i.e., expand nested prods) • if any factors can be combined into a power, make the substitution • else, sort factors and create prod object |
| sum | <ul style="list-style-type: none"> • compare small-order values of terms and remove those that are negligible • check for trig identities: $\sin^2x + \cos^2x \rightarrow 1$; $1 - \sin^2x \rightarrow \cos^2x$; etc. • if any terms are sums, remove them and append terms from nested sums to existing list (i.e., expand nested sums) • if sym-value of sum would be negative, negate all terms and return negative sum (prod with coefficient of -1) • else, sort terms and create sum object |

^aSimplifications marked with •• mean that after the simplification is performed, the creator function is called recursively, using updated arguments.

Table 6 Summary of primitive mathematics operations

| Operation | Argument(s) | Description |
|---------------------|--|--|
| add | x, y | $\text{gcf}(x, y) [x/\text{gcf}(x, y) + y/\text{gcf}(x, y)]$ |
| angle | $v_1, v_2, \{v_3\}$ | angle between v_1 and v_2 , with sign determined by optional v_3 , as illustrated in Fig. 3 |
| const-or-var | x | is x constant or variable? |
| constant-part | x | constant part of expression |
| convert-coordinates | $\text{coordinates}, \text{oldbody}, \text{newbody}$ | convert coordinates from <i>coordinate</i> system of <i>oldbody</i> to the coordinates system of <i>newbody</i> |
| cross | v_1, v_2 | $v_1 \times v_2$ |
| dir | v | direction of vector, i.e., |
| div | x, y | x/y (y must be scalar) |
| dot | v_1, v_2 | $v_1 \cdot v_2$ |
| dot-plane | v_1, v_2 | project v_1 onto plane normal to v_2 , i.e., $\text{dot-plane}(v_1, v_2) = v_1 - \frac{v_1 \cdot v_2 v_2}{v_2 \cdot v_2}$ |
| dxdt | x | $\frac{dx}{dt}$ |
| gcf | x, y | symbolic greatest common factor |
| inv | x | $1/x$ (x must be scalar) |
| mag | v | scalar magnitude of vector, $ v \rightarrow \sqrt{v \cdot v}$ |
| mul | x, y | xy (either x or y must be a scalar) |
| neg | x | $-x$ |
| nominal | exp | set all generalized coordinates in <i>exp</i> to zero |
| partial | y, x | $\partial y / \partial x$ (x is scalar) |
| sub | x, y | $x - y$ |

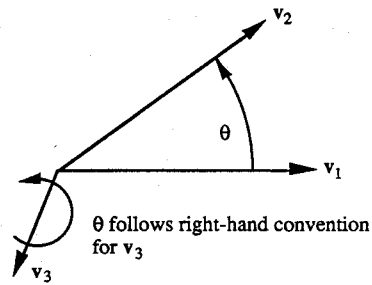


Fig. 3 Angle convention.

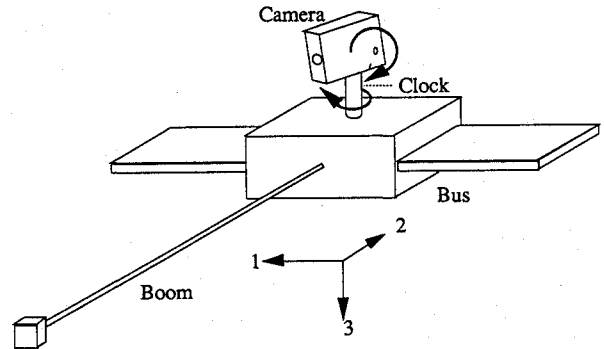


Fig. 4 Satellite multibody system.

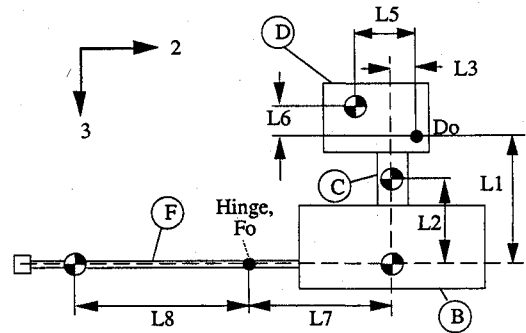


Fig. 5 Dimensions of satellite.

involves operations defined for scalar algebra, together with dot products between unit vectors. Thus, the primitive dot-product operation is defined for two uv arguments. Recall that the uv contains a slot called *dot-products*. This contains a table with all pairs of uvs whose dot product is known. Initially, each table contains three entries for the three uvs in the body in which the uv is defined. (The values are 1 for the dot product of the uv with itself and 0 for the other two uvs of the trio.) If the table in the first uv contains the answer, it is used. If not, the table in the second uv is checked. Again, if the answer is in the table, it is used. If not, the following analysis is performed.

The uv whose body is farthest down the topology tree is identified by comparing the values from the *level* slots of the bodies. That uv is transformed into an expression involving the three uvs of its parent body by using the direction cosine matrix (from the *cos-matrix* slot), as will be described shortly. The dot product is then taken between the new expression and the uv that was up the tree. This method is recursive—the dot operator is defined in terms of itself. It works because with each recursion the expressions being considered are simpler and/or the uvs are from bodies that are closer in the tree. Eventually, the process stops when both arguments are uvs associated with the same body. (In the most complicated case possible, both uvs would be transformed to the inertial reference.) The results of the process are stored in the table of dot products for one of the uvs, so that the tree-climbing and matrix multiplications are not required the next time the dot product is needed. This method of tree climbing ensures that the minimum number of direction transformations is performed for each dot product operation.

The cross-product operation is performed using the same recursive approach as just described for the dot product. A uv crossed with a uv is obtained from the table of values in the cross-product slot of either uv if available (with a multiplica-

tion by -1 if the table of the second uv is used). Otherwise, the cross product is formulated using the expansion:

$$a_i \times b_j \rightarrow [(a_i \cdot b_1)b_1 + (a_i \cdot b_2)b_2 + (a_i \cdot b_3)b_3] \times b_j \quad (4)$$

where a_i is the first uv, b_j is the second, and $b_1, b_2,$ and b_3 are the unit vectors for the body containing b_j . An angle function is included, which uses dot-product and cross-product operations to define an angle between two arbitrary vectors, with a right-handed sign convention (see Fig. 3).

The derivative of an arbitrary expression is determined using elementary rules of calculus to recursively expand the expression into products and sums of simpler expressions and their derivatives. The expansion stops when a *sym*, number, or uv is reached. The time derivative of a *sym* is zero if the expression is a constant, otherwise it is obtained from the *dxdt* slot. (When *sym* structures are created to represent state variables, the *dxdt* slots are assigned to the appropriate *sym* for the derivative.)

The time derivative of a uv (u) is defined as

$$\dot{u} \rightarrow \omega^B \times u \quad (5)$$

where ω^B is the absolute rotational velocity of the body containing u . Note that Eq. (5) is always valid, even if simplifications have been made involving small angles and small speeds.

Example Multibody System

Figure 4 shows an example multibody system that will be used to illustrate the multibody operations and the general use of AUTOSIM. The system is a satellite with a main body *B* (called the bus), a flexible boom *F*, and a camera *D*, mounted on a clock *C*. Dimensions and locations of significant points are shown in Fig. 5. The bus has six degrees of freedom relative to the inertial reference. The clock is a shaft that rotates relative to the bus, the camera is a body attached to the clock with a hinge joint, and the flexible boom is modeled as a rigid body attached to the bus with a two-degree-of-freedom hinge at a point *Fo*, with torsional stiffness *KB* and torsional damping rate *DB* in the directions 1 and 3. Movements of the clock and camera are controlled. The controller is modeled as a torque applied to the clock through a massless element with torsional stiffness *KC* and torsional damping rate *DC*. The torque applied to the camera is also through a massless element with the same stiffness and damping properties.

The complete description of this system is listed in Fig. 6. Although space limitations prevent a thorough discussion of Fig. 6, the following hints may prove helpful for understanding the inputs. AUTOSIM commands are lists enclosed by parentheses. Items in a list are separated by white space or by appearing on different lines. Regardless of how many lines are covered, the list ends when the closing parenthesis is encountered. The first symbol in the list (a name, possibly hyphenated) is the name of a procedure—a Lisp macro or function.

Table 7 Parameters and degrees of freedom of a body/joint

| Parameter | Description |
|--|--|
| $r^{A_0 B J}$ | position of joint point of <i>B</i> relative to origin of parent |
| $\{\{d_{r1}^B, \{d_{r2}^B, \{d_{r3}^B\}\}\}\}$ | list of 0, 1, 2, or 3 directions for translational degrees of freedom of <i>B</i> , fixed in the coordinate system of the parent |
| $\{\{i_1, \{i_2, i_3\}\}\}$ | list of 0, 1, or 3 axis indices in <i>B</i> for sequential rotations |
| d_{rot}^B | orientation of first rotation axis of <i>B</i> (fixed in the coordinate system of the parent) |
| d_{ref}^B | reference direction for first rotation of <i>B</i> (fixed in the coordinate system of the parent) |
| $\{\{d_{r1}^B, \{d_{r2}^B, d_{r3}^B\}\}\}$ | list of 0, 1, or 3 directions of rotations for <i>B</i> . This list is derived from the above parameters. |

Other items in the list are arguments for that procedure. The AUTOSIM macros add-body and add-moment have numerous arguments that are optional. If optional arguments are not

```
(reset)
(add-body B :translate (1 2 3) :body-rotation-axes (1 2 3)
:small-angles (t t t) :small-translations (t t t))
(add-body c :parent b :inertia-matrix (ic ic 0)
:body-rotation-axes 3 :cm-coordinates (0 0 -L2))
(add-body d :parent c :joint-coordinates (0 L3 -L1)
:cm-coordinates (0 -L5 -L6) :body-rotation-axes 1)
(add-body e :parent b :Joint-coordinates (0 -L7 0)
:small-angles t
:inertia-matrix 0 :mass 0 :body-rotation-axes 3)
(add-body f :parent e :inertia-matrix (if1 if2 if1)
:small-angles t
:cm-coordinates (0 -L8 0) :body-rotation-axes 1)
;;; Moments due to flexing of the boom
(add-moment bt1 :direction [e3] :body1 f :body2 b
:magnitude "-kb*q(9) - db*u(9)")
(add-moment bt2 :direction [f1] :body1 f :body2 b
:magnitude "-kb*q(10) - db*u(10)")
;;; add moments from clock and camera motors
(add-variables difeqn real clkcmd camcmd)
(add-subroutine difeqn cmd (t clkcmd camcmd))
(add-moment clockt :direction [c3] :body1 c :body2 b
:magnitude "kc*(-q(7) + clkcmd) - dc*u(7)")
(add-moment camt :direction [d1] :body1 d :body2 c
:magnitude "kc*(-q(8) + camcmd) - dc*u(8)")
;;; add moments from thrusters
(add-moment tt1 :direction [b1] :body1 b
:magnitude "l1t1*thrust(t, 1, (g*u(4) + q(4)))")
(add-moment tt2 :direction [b2] :body1 b
:magnitude "l1t2*thrust(t, 2, (g*u(5) + q(5)))")
(add-moment tt3 :direction [b3] :body1 b
:magnitude "l1t3*thrust(t, 3, (g*u(6) + q(6)))")
;;; set labels and default values of parameters
(mks)
(dynamics)
```

Fig. 6 AUTOSIM inputs to describe spacecraft example.

Table 8 Values assigned to slots in the bodies of the satellite example

| Slot | Bus | Clock | Camera |
|-------------------------|---|---|---|
| symbol | B | C | D |
| name | Bus | Clock | Camera |
| parent | N | B | c |
| level | 1 | 2 | 3 |
| children | C | D | |
| translation-coordinates | q_1, q_2, q_3 | | |
| translation-directions | n_1, n_2, n_3 | | |
| uvs | b_1, b_2, b_3 | c_1, c_2, b_3 | c_1, d_2, d_3 |
| basis | $b_1 b_1 + b_2 b_2 + b_3 b_3$ | $c_1 c_1 + c_2 c_2 + b_3 b_3$ | $c_1 c_1 + d_2 d_2 + d_3 d_3$ |
| rotation-coordinates | q_4, q_5, q_6 | q_7 | q_8 |
| rotation-directions | $n_1, (c_6 b_2 + s_6 b_1), b_3$ | b_3 | c_1 |
| cos-matrix | $c_5 c_6, (c_4 s_6 + c_6 s_5 s_4), (s_4 s_6 - c_4 c_6 s_5)$ $- c_5 s_6, (c_4 c_6 - s_5 s_4 s_6), (c_6 s_4 + c_4 s_5 s_6)$ $s_5, - c_5 s_4, c_4 c_5$ | $c_7, s_7, 0$ $- s_7, c_7, 0$ $0, 0, 1$ | $1, 0, 0$ $0, c_8, s_8$ $0, - s_8, c_8$ |
| abs-w | $u_4 b_1 + u_5 b_2 + u_6 b_3$ | $u_4 b_1 + u_5 b_2 + (u_6 + u_7) b_3$ | $u_4 b_1 + u_5 b_2 + (u_6 + u_7) b_3 + u_8 c_1$ |
| abs-v0 | $u_3 b_3 + (P_1 u_5 + u_1) b_1 - (P_1 u_4 - u_2) b_2$ | $- [(P_1 u_4 - u_2) b_2 - (P_1 u_5 + u_1) b_1 - u_3 b_3]$ | $[- (L_1 - P_1) u_5 + u_1] b_1 + [(L_1 - P_1) u_4 + u_2] b_2 - L_3 (u_6 + u_7) c_1 + [u_3 + L_3 (u_4 c_7 + u_5 s_7)] b_3$ |

Table 9 Slots in a body that establish its orientation

| Slot name | Definition |
|-----------------------------|---|
| <i>uvs</i> | three unit-vectors that define the 1-2-3 axis directions in <i>B</i> |
| <i>basis</i> | a dyadic that transforms an arbitrary vector expression into the basis of this body, e.g., $\vec{B} = b_1b_1 + b_2b_2 + b_3b_3$ |
| <i>rotation-directions</i> | directions associated with each joint rotational degree of freedom |
| <i>rotation-coordinates</i> | generalized coordinates introduced for each joint rotational degree of freedom |
| <i>cos-matrix</i> | direction cosine matrix between <i>B</i> and <i>A</i> |

Table 10 Indices for three possible rotation axes

| First index | <i>i</i> | <i>j</i> | <i>k</i> |
|-------------|----------|----------|----------|
| $i_1 = 1$ | 2 | 3 | 1 |
| $i_1 = 2$ | 3 | 1 | 2 |
| $i_1 = 3$ | 1 | 2 | 3 |

provided, default conditions are assumed. Symbols beginning with a colon are keywords that identify specific optional arguments. For example, the keyword `:translate` indicates that the list (1 2 3) defines directions of translation. The `add-body` macros at the top of Fig. 6 provide a complete description of the system topology needed to support the symbolic multibody operations. Note that the bus is represented as body *B*. This should not be confused with the generic body *B* that is used to signify the current body of interest.

Multibody Operations

In order for the dot-product operation to work, a direction cosine matrix is required for each body. A few other pieces of data are also needed in order to define vector speeds and positions. The direction cosine matrix and several useful expressions are formed automatically when the dynamicist uses the macro `add-body`. The kinematical relationship between a generic body *B* and its parent *A* is defined by the dynamicist using quantities listed in Table 7. In the table, optional items are enclosed in curly braces. Figure 7 illustrates the geometry of a joint having one degree of freedom for rotation and one for translation. Table 8 shows the cosine matrices, unit-vector triads, and other values of slots of the body objects for three of the components of the example spacecraft. Table 9 lists and defines slots that contain information needed to establish the orientation of *B*.

The unit vectors of *B* are related to those of *A* by the direction cosine matrix ${}^B C^A$, defined as,

$$\begin{Bmatrix} b_1 \\ b_2 \\ b_3 \end{Bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} \quad (6)$$

$$C_{ij} = b_i \cdot a_j \quad (7)$$

If generic body *B* has zero rotational degrees of freedom, then its direction cosine matrix is a 3 × 3 identity matrix. The contents of the *basis* and *uvs* slots are simply copied from *A* to *B*.

If *B* can rotate with respect to *A*, the first rotation is about an axis whose direction d_{rot}^B is fixed in *A*. In the nominal configuration, the three orthogonal unit vectors that establish the coordinate system of *B* are defined in terms of the inputs from the dynamicist as follows:

$$b_k = d_{rot}^B \quad b_j = b_k \times d_{ref}^B \quad b_i = b_j \times b_k \quad (8)$$

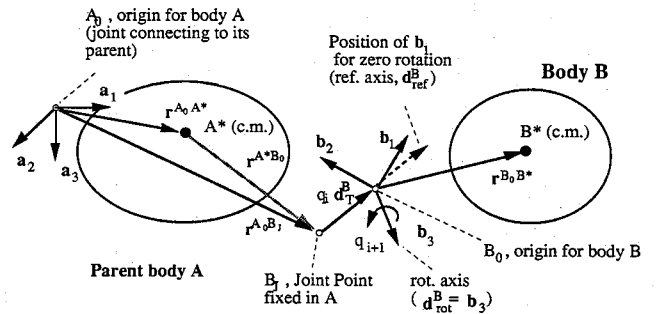


Fig. 7 Geometry of body relative to its parent.

Table 11 Slots in a body that locate its origin

| Slot name | Definition |
|--------------------------------|--|
| <i>0-point</i> | origin of coordinate system and joint attachment point in this body |
| <i>joint-point</i> | joint attachment in parent body |
| <i>translation-coordinates</i> | generalized coordinates introduced for joint translational degree of freedom |
| <i>translation-directions</i> | directions associated with joint translational degree of freedom |

The set of unit vectors introduced for *B* are nominally designated $b_1, b_2,$ and $b_3,$ and are identical to the unit vectors $b_i, b_j,$ and $b_k,$ where the definitions of the indices *i, j,* and *k* are obtained from Table 10. In the table, the first index is the first axis of rotation specified by the dynamicist. (In Fig. 6, the first index for the `add-body` inputs for the bus is 1.) A direction cosine matrix is defined as follows. Calling the rotation angle $\theta,$ two terms, *s* and *c,* are introduced as the sine and cosine of θ to account for the rotation. (The creator functions for a `sin` and `cos` make small-angle approximations appropriately.) The elements of the direction cosine matrix are defined for each row using the same *i, j,* and *k* indices obtained from Table 10:

$$C_{ri} = c(a_r \cdot b_i) + s(a_r \cdot b_j) \quad (r = 1, 2, 3) \quad (9a)$$

$$C_{rj} = -s(a_r \cdot b_i) + c(a_r \cdot b_j) \quad (r = 1, 2, 3) \quad (9b)$$

$$C_{rk} = a_r \cdot b_k \quad (r = 1, 2, 3) \quad (9c)$$

If the rotation axis is parallel to one of the unit vectors of *A*, then the corresponding unit vector is also used for *B*. In the satellite example, the rotation axis of body *C* is $b_3.$ Thus, the unit vectors of the clock are $c_1, c_2,$ and $b_3.$

A body with three rotational degrees of freedom is subject to three consecutive rotations. Starting with the nominal orientation, after each of the three rotations, the orientation coincides with 1) a reference frame $B'',$ 2) a reference frame $B',$ and 3) body *B*. The method just described to obtain a direction cosine matrix for a body with one rotational degree of freedom is applied three times to obtain cosine matrices relating *B* to $B',$ B' to $B'',$ and B'' to the parent:

$${}^B C^A = {}^B C^{B'} {}^{B'} C^{B''} {}^{B''} C^A \quad (10)$$

Slots that contain information needed to locate the origin of *B* are listed and defined in Table 11. The origin is a point object created automatically for the body with coordinates (0, 0, 0) and assigned to the *0-point* slot of the new body object. The coordinates of the *joint-point* are provided by the dynamicist and define the vector $r^{A_0 B_j}.$ The list of *translation-directions* is also provided by the analyst. The generalized coordinates are indexed-sym structures, created automatically, and put in a list assigned to the *transla-*

tion-coordinates slot. For the satellite example (Table 8), the bus has three translational degrees of freedom. The translation directions are parallel to axes in the inertial reference frame and are indicated by numerical indices in the input. If a direction is not aligned with an axis, a list of coordinates is provided to the add-body macro instead of an index.

Table 12 lists three operations that provide vector expressions for the multibody system. The pos operation derives a position vector between any two points. If the two points are in the same body, their coordinates are subtracted and the results are multiplied by the appropriate unit vectors to yield a vector expression. Otherwise, offset vectors that define the position of the origin of a body relative to the origin of its parent are added and subtracted as needed to handle displacements across bodies. The rotational velocity of B is directly available from a slot called *abs-w*. The velocity of B_0 is available from the slot *abs-v0*. The absolute velocity of a point P fixed in B is derived using the relationship:

$$v^P = v^{B_0} + \omega^B \times r^{B_0 P} + B v^P \tag{11}$$

where ${}^B v^P$ is the relative velocity of P within the reference frame of B . (This term is zero when the point is fixed in B .) The vel operation applies Eq. (11) to the two specified points and subtracts the results. Note that the expressions for ω^B and v^{B_0} generally involve unit vectors from several bodies to maintain the simplest possible form (i.e., without trigonometric functions).

Operations on Program Code

Programming simplifications are easiest to implement after the simulation code has been generated and can be inspected. This means that equations are not written as they are derived, but are kept in computer memory as eqs objects.

The simulation code generated by AUTOSIM includes two sets of intermediate symbols used to replace expressions. One set is for constant expressions and the other is for variables. (Both are called intermediate variables in the following since that is how they are implemented in a Fortran program.) A function called *intro-var-if-new* is used to process expressions and introduce new variables as needed and is indicated in this paper by enclosing the expression with the symbols « and ». The replacements are *indexed-sym* objects, designated p_i for constants and z_i for variables. A simplified version of the algorithm used to process an expression « x » is described in the following, with examples. In the example expressions, the symbols A , B , and C are constants, the symbols X and Y are variables, and symbols shown in bold type are unit vectors.

1) If x is a number, an *indexed-sym*, a *sym*, a *uv*, or a *dyad*, return x . For example,

$$\langle\langle 3 \rangle\rangle \rightarrow 3 \quad \langle\langle A \rangle\rangle \rightarrow A \quad \langle\langle a_1 \rangle\rangle \rightarrow a_1 \tag{12}$$

2) Else, if x is a vector or dyadic, collect terms so that each unit vector or dyad appears only once, and then apply the function recursively to the scalar expressions. For example,

$$\langle\langle A a_1 + B b_1 - C a_1 \rangle\rangle \rightarrow \langle\langle A - C \rangle\rangle a_1 + \langle\langle B \rangle\rangle b_1 \tag{13}$$

3) Else, if the expression is in a table of existing intermediate variables, return the corresponding *indexed-sym*.

4) Else, if the expression is a constant,
 a) make a new *indexed-sym* p_i , where the index i is incremented from the highest index used previously for that eqs.

b) put p_i at the end of the list in the eqs object for intermediate constants.

c) put x and p_i into the table of intermediate variables so that the next time expression x is encountered the symbol p_i will be found in step 3.

d) if an option to expand constants is enabled, recursively expand any intermediate variables in the expression assigned to p_i . (This is done to formulate constants that are more easily interpreted by a human and to permit possible cancellations.) If the option is disabled, this step is skipped.

e) return p_i as the result of the procedure.

For example,

$$\langle\langle A(2.3B - C^2) \rangle\rangle \rightarrow p_{10} \tag{14}$$

5) Else, if any constant expressions can be factored out, do so. Apply *intro-var-if-new* to the constant part, the variable part, and the product. For example,

$$\langle\langle A(2BX + B^2Y) \rangle\rangle \rightarrow \langle\langle AB \rangle\rangle \langle\langle 2X + BY \rangle\rangle \tag{15}$$

6) Else, x is a compound expression that is not a constant.

a) If x is a *prod*, with more than two factors, process the scalar factors two at a time. For a *prod* with coefficient c and five factors, the processing sequence is

$$\langle\langle c f_1 f_2 f_3 f_4 f_5 \rangle\rangle \rightarrow \langle\langle c \langle\langle \langle\langle f_1 f_2 \rangle\rangle f_3 \rangle\rangle f_4 \rangle\rangle f_5 \tag{16}$$

b) If x is a *prod* with just two factors, or a single factor and a numerical coefficient that is not ± 1 , then a new *indexed-sym* is introduced:

i. make a new *indexed-sym* z_i , where the index i is incremented from the highest index used previously for that eqs.

ii. put z_i at the end of the list in the eqs object for intermediate variables.

iii. put x and z_i into the table of intermediate variables so that the next time expression x is encountered the symbol z_i will be found in step 3.

iv. return z_i as the result of the procedure.

c) If x is a sum of terms t_1, t_2, \dots, t_n , the processing sequence is

$$\langle\langle t_1 + t_2 + \dots + t_n \rangle\rangle \rightarrow \langle\langle \langle\langle t_1 \rangle\rangle + \langle\langle t_2 \rangle\rangle + \dots + \langle\langle t_n \rangle\rangle \rangle \tag{17}$$

after the terms are processed, a new intermediate variable is introduced for the entire sum using the process in 6b.

d) if x is a power, the base is first processed and then the power is replaced with a new variable using the procedure from 6b

$$\langle\langle b^p \rangle\rangle \rightarrow \langle\langle \langle\langle b \rangle\rangle^p \rangle \tag{18}$$

e) if x is a function with arguments a_1, a_2, \dots, a_n , the arguments are first processed, and then the function is replaced using the procedure from 6b

$$\langle\langle f(a_1, a_2, \dots, a_n) \rangle\rangle \rightarrow \langle\langle f(\langle\langle a_1 \rangle\rangle, \langle\langle a_2 \rangle\rangle, \dots, \langle\langle a_n \rangle\rangle) \rangle \tag{19}$$

This algorithm is recursive, and results in a number of intermediate expressions being introduced for a single compound expression. Consider the example

$$\langle\langle A \sin(BX + CY)^2 / \cos(3A) \rangle\rangle \rightarrow z_6 \tag{20}$$

where

$$\begin{aligned} p_1 &= A / \cos(3A) \\ z_1 &= BX \quad z_2 = CY \quad z_3 = (z_1 + z_2) \\ z_4 &= \sin(z_3) \quad z_5 = z_4^2 \quad z_6 = p_1 z_5 \end{aligned} \tag{21}$$

Table 12 Summary of operations for bodies and points

| Operation | Argument(s) | Description |
|-----------|-------------|-----------------------------------|
| pos | P_1, P_2 | $r^{P_2 P_1} = r^{P_1} - r^{P_2}$ |
| rot | B | ω^B |
| vel | P_1, P_2 | $v^{P_1} - v^{P_2}$ |

```

C Simple control subroutines for spacecraft example
C
SUBROUTINE CMD(T, CLKCMD, CAMCMD)
PARAMETER (PI = 3.1415926)
IF (T .LT. 1.) THEN
  CLKCMD = 4.-PI
  CAMCMD = -.5
ELSE IF (T .LT. 11.) THEN
  CLKCMD = 4. -.025*(T-1.) - PI
  CAMCMD = -.5 + .01*(T-1.)
ELSE
  CLKCMD = 3.75 -PI
  CAMCMD = -.4
END IF
RETURN
END

FUNCTION THRUST(T, AXIS, ERROR)
INTEGER AXIS
REAL DBAND, TMIN, FIRE(3), TOFF(3)
SAVE TOFF, FIRE
DATA DBAND /.0025/
DATA TMIN /.02/
DATA FIRE, TOFF /3*0., 3*0./

IF (ERROR .LT. -DBAND) THEN
  IF (FIRE(AXIS) .LT. 1.) TOFF(AXIS) = T + TMIN
  FIRE(AXIS) = 1
ELSE IF (ERROR .GT. DBAND) THEN
  IF (FIRE(AXIS) .GT. -1.) TOFF(AXIS) = T + TMIN
  FIRE(AXIS) = -1
ELSE IF (T .GE. TOFF(AXIS)) THEN
  FIRE(AXIS) = 0
END IF
THRUST = FIRE(AXIS)
RETURN
END

```

Fig. 8 External function to simulate thrusters.

The original expression in Eq. (20) required four multiply operations, one divide, one integer power, two function evaluations, and one add. By factoring out the constant $A/\cos(3A)$, the operations needed after the constants are reduced to three multiplies, one integer power, one function, and one add. Also, further processing might involve one or more of the intermediate variables. For example, consider an expression with some of the same terms:

$$\ll 5 \cos(3A)/\sin(BX + CY) \gg -z_7 \quad (22)$$

where

$$p_2 = 5 \cos(3A) \quad z_7 = p_2/z_4 \quad (23)$$

In this case, evaluating the expression in Eq. (22) involves just a single additional divide operation, after handling the constant.

Before the equations are written as output in the target language, they are inspected for intermediate variables that are not needed. Recall (or see Table 1) that one of the slots in the *sym* object is called *hide*. The *hide* slot is used to keep count of how many times the *sym* actually appears in the equations. To count occurrences, the *hide* slots in all intermediate variables in an eqs are set to zero and then equations used to compute derivatives and output variables are processed with a function that operates recursively to validate *syms*. An important part of the design of AUTOSIM is that the two symbolic elements—the *sym* and the *uv*—are stored in memory such that there are no copies (e.g., the object printed as z_2 exists in only one place, even though it appears in more than one expression). Lisp uses *pointers* to reference such objects. When an elementary object such as a *sym* is changed, all expressions containing that element are updated since their pointers continue to point at the changed object.) The eqs object only prints equations involving *syms* whose *hide* slots are no longer set to zero. For example, if an eqs structure contains 100 equations, but only 10 involve *syms* with *hide* counts greater than zero, then only 10 equations are printed. The other 90 equations are still in memory, but are hidden.

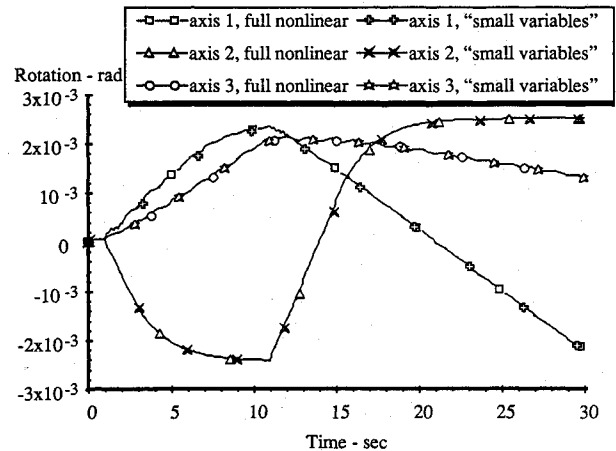


Fig. 9 Attitude of spacecraft body computed for two formulations.

There are some reasons not to introduce a new intermediate variable if that variable will only be used once. First, the equations become almost unreadable by humans. The equations of motion for multibody systems are usually complicated to begin with, and introducing intermediate variables that only appear once compounds the difficulty. Further, some compilers optimize machine instructions for large expressions, putting temporary intermediate results in machine locations that exploit the design of the specific hardware. If an intermediate variable is defined in the source code, the compiler is obliged to save its value, possibly at the expense of computational efficiency. After the *hide* values have been established for all indexed-syms that appear on the left side of an equation, a second pass is made in which all intermediate variables that are used only once (*hide* = 1) are expanded back into the original expressions.

Discussion

Automated modeling of multibody systems has typically offered great convenience for the dynamicist who is willing to sacrifice certain capabilities. In the case of generalized numerical codes, computation inefficiency is sacrificed and some types of subcomponent models are difficult or impossible to include in the programmed system description. In the case of symbolic multibody programs, a programmer must write external functions and subroutines, which are often quite complex, and then manually edit those routines into the computer-generated code. One step in remedying these limitations is to develop a computer language that can perform the same symbolic procedures as a human analyst armed with pencil, paper, and persistence. This paper described the design of such a language. Consider once again the example spacecraft system. At this time, we will consider the treatment of moments acting on the rigid bodies, and also the inclusion of external procedures.

Suppose the object of the simulation is to simulate a slew maneuver in which the clock and camera are moved from initial values of $(4 - \pi)$ and -0.5 rad, respectively, to final values of $(3.75 - \pi)$ and -0.4 rad, over a 10-s interval. The orientation of the spacecraft body is controlled by three pairs of thrusters that fire bursts of propellant when the angle of the craft drifts beyond a dead zone tolerance of 0.0025 rad, and which remain on for at least 0.02 s. Each pair of thrusters is balanced to apply a pure couple to *B* about the axes 1, 2, and 3. Figure 8 lists a very simple Fortran subroutine that provides controller commands for this maneuver and also a Fortran function that defines the control laws for the thrusters. The Fortran code shown in Fig. 7 is the **complete** handwritten part of the simulation code. Every other line of code is written or assembled automatically by AUTOSIM, based on the inputs from Fig. 6. The macro *add-moment* is used to include the

Table 13 Parameter values for example

| | | | |
|-----------|-------------------------|---------|-----------------------|
| DB | 10 N-m-s | IF_2 | 0.2 kg-m ² |
| DC | 20 N-m-s | KB | 2000 N-m/rad |
| G | 2 s | KC | 3500 N-m/rad |
| IB_{11} | 115 kg-m ² | L_1 | 1.5 m |
| IB_{12} | -14 kg-m ² | L_2 | 0.75 m |
| IB_{13} | 14 kg-m ² | L_3 | 0.1 m |
| IB_{22} | 316 kg-m ² | L_5 | 0.22 m |
| IB_{23} | -34.6 kg-m ² | L_6 | 0.2 m |
| IB_{33} | 440 kg-m ² | L_7 | 1.2 m |
| IC | 0.35 kg-m ² | L_8 | 3.3 m |
| ID_{11} | 4.85 kg-m ² | LTT_1 | 0.23 m |
| ID_{12} | 0.41 kg-m ² | LTT_2 | 0.21 m |
| ID_{13} | -0.07 kg-m ² | LTT_3 | 0.31 m |
| ID_{22} | 2.2 kg-m ² | MB | 410 kg |
| ID_{23} | -0.54 kg-m ² | MC | 6.8 kg |
| ID_{33} | 5.5 kg-m ² | MD | 57.5 kg |
| IF_1 | 27.2 kg-m ² | MF | 10.7 kg |

moments applied by the torsional springs and the thrusters. The macros add-variables and add-subroutine cause AUTOSIM to properly include a CALL to the user-written subroutine CMD. The command mks specifies that the MKS units system should be used to generate labels for output plot files and documentation files that are optionally generated by AUTOSIM. The command dynamics results in the complete derivation of the equations of motion for the system.

The portions of the inputs in Fig. 6 that are underlined specify that certain variables are small. These include the 12 coordinates and speeds associated with B , and the four rotational coordinates and speeds associated with the Boom (bodies E and F). However, the rotational coordinates and speeds of bodies C and D are not small. Figure 9 compares time history plots computed by simulation codes generated with and without the underlined portions of the input. It shows that the small-variable assumptions are fully justified for the slew maneuver of interest. Table 13 lists numerical values used to compute these results. Each parameter was generated from the AUTOSIM input from Fig. 6, either explicitly (e.g., L_1 , K_B , etc.) or implicitly (e.g., M_B , I_{B11} , etc.). A companion paper 1) provides details as to how the equations for this system are formed, 2) includes excerpts of the equations, and 3) contains a summary of their computational complexity.¹⁹

Conclusions

Methods were presented for representing all of the components of a simulated multibody system in symbolic form on a computer, including 1) algebraic expressions for vector/dyadic analyses, 2) physical components in a multibody system, and 3) program structures needed in a simulation code. A language called AUTOSIM has been written in Lisp to implement these methods. Modeling and programming strategies employed by humans can be mimicked in computer software when all of these objects are available for computer manipulation. The main practical advantages of this approach are 1) models of dynamic systems can be developed with much less effort than alternative methods, 2) modeling options are available to tailor a simulation code to match the expectations of an intended end user (e.g., by using arbitrary coordinate systems and familiar parameter definitions), and 3) the numerical computation software generated is highly efficient, such that the code can be used for real-time simulation and other applications where computational efficiency is critical. An advantage for researchers is that alternative modeling strategies can be tested and compared with a fraction of the effort that would otherwise be needed.

Acknowledgments

The work reported in this paper was funded by the U.S. Army Tank and Automotive Command and by the University

of Michigan Transportation Research Institute Fellowship program.

References

- ¹Crespo da Silva, M. R. M., and Hodges, D. H., "Role of Computerized Symbolic Manipulation in Rotorcraft Dynamics Analysis," *Computers & Mathematics with Applications*, Vol. 12a, No. 1, 1986, pp. 161-172.
- ²Golnaraghi, M., Keith, W., and Moon, F. C., "Stability Analysis of a Robotic Mechanism Using Computer Algebra," *Applications of Computer Algebra*, edited by R. Pavelle, Kluwer, Boston, MA, 1984, pp. 281-292.
- ³Hussain, M. A., and Noble, B., "Application of Macsyma to Kinematics and Mechanical Systems," *Applications of Computer Algebra*, edited by R. Pavelle, Kluwer, Boston, MA, 1984, pp. 262-280.
- ⁴Pavelle, R., "Macsyma: Capabilities and Applications to Problems in Engineering and the Sciences," *EUROCAL '85 European Computer Algebra Conference*, Springer-Verlag, New York, 1985, pp. 19-32.
- ⁵Levinson, D., "The Derivation of Equations of Motion of Multiple-Rigid-Body Systems Using Symbolic Manipulation," AIAA Paper 76-816, 1976.
- ⁶Krishnaswami, P., and Bhatti, M. A., "Symbolic Computing in Optimal Design of Dynamic Systems," *Proceedings of the 10th Design Engineering Division Conference on Mechanical Vibration and Noise*, Cincinnati, OH, American Society of Mechanical Engineers, Sept. 1985, pp. 1-6.
- ⁷Char, B. W., Geddes, K. O., Gentleman, W. M., and Gonnet, G. H., "The Design of MAPLE: A Compact, Portable, and Powerful Computer Algebra System," *EUROCAL '83 European Computer Algebra Conference*, Springer-Verlag, New York, 1983, pp. 101-115.
- ⁸Wooff, C., and Hodgkinson, D., *muMATH: A Microcomputer Algebra System*, Academic, London, England, UK, 1987.
- ⁹Wolfram, S., *Mathematica*, Addison-Wesley, Reading, MA, 1988, p. 767.
- ¹⁰Nielan, P., and Kane, T., "Symbolic Generation of Efficient Simulation/Control Routines for Multibody Systems," *Dynamics of Multibody Systems, IUTAM/IFTOMM Symposium*, Springer-Verlag, New York, 1985, pp. 153-164.
- ¹¹Schaechter, D. B., and Levinson, D. A., "Interactive Computerized Symbolic Dynamics for the Dynamicist," *Journal of the Astronautical Sciences*, Vol. 36, No. 4, 1988, pp. 365-388.
- ¹²Kane, T. R., and Levinson, D. A., *Dynamics, Theory and Applications*, McGraw-Hill, New York, 1985, p. 395.
- ¹³Schiehlen, W. O., and Kreuzer, E. J., "Symbolic Computerized Derivation of Equations of Motion," *Dynamics of Multibody Systems*, Springer-Verlag, New York, 1977, pp. 290-305.
- ¹⁴Rosenthal, D. E., and Sherman, M. A., "High Performance Multibody Simulations via Symbolic Equation Manipulation and Kane's Method," *Journal of the Astronautical Sciences*, Vol. 34, No. 3, 1986, pp. 223-239.
- ¹⁵Wittenburg, J., and Wolz, U., "MESA VERDE: A Symbolic Program for Nonlinear Articulated-Rigid-Body Dynamics," *Proceedings of the 10th Design Engineering Division Conference on Mechanical Vibration and Noise*, Cincinnati, OH, American Society of Mechanical Engineers, Sept. 1985.
- ¹⁶Orlandea, N., Chace, M. A., and Calahan, D. A., "A Sparsity-Oriented Approach to the Dynamic Analysis and Design of Mechanical Systems, Parts I and II," *Journal of Engineering for Industry*, Vol. 99, Aug. 1977, pp. 773-784.
- ¹⁷Nikravesh, P. E., and Haug, E. J., "Generalized Coordinate Partitioning for Analysis of Mechanical Systems with Nonholonomic Constraints," *Journal of Mechanisms, Transmissions, and Automation in Design*, Vol. 105, Sept. 1983, pp. 379-384.
- ¹⁸Steele, G. L. J., *Common Lisp: The Language*, Digital, Bedford, MA, 1984.
- ¹⁹Sayers, M. W., "Symbolic Vector/Dyadic Multibody Formalism for Tree-Topology Systems," *Journal of Guidance, Control, and Dynamics*, Vol. 14, No. 6, 1991, pp. 1240-1250.
- ²⁰Sayers, M. W., "Symbolic Computer Methods to Automatically Formulate Vehicle Simulation Codes," Ph.D. Dissertation, University of Michigan, Ann Arbor, MI, Jan. 1990.
- ²¹Stribersky, A., Fancher, P. S., MacAdam, C. C., and Sayers, M. W., "On Nonlinear Oscillations in Road Trains at High Forward Speeds," *11th IAVSD Symposium of Vehicles on Roads and Tracks*, edited by R. J. Anderson, Swets and Zeitlinger, Amsterdam, 1989, pp. 552-565.