SYMBOLIC COMPUTER METHODS TO AUTOMATICALLY FORMULATE VEHICLE SIMULATION CODES

by

Michael William Sayers

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Mechanical Engineering)
in The University of Michigan
1990

Doctoral Committee:

Assistant Professor Sridhar Kota, co-chairman
Adjunct Professor Robert R. Ryan, co-chairman
Professor Donald T. Greenwood
Professor Robert Howe
Professor Emeritus Leonard Segel
Research Scientist Paul S. Fancher

to my father, Robert E. Sayers

ACKNOWLEDGMENTS

The research reported in this dissertation was funded primarily by the United States Army Tank Automotive Command (TACOM). Additional support was provided through the University of Michigan Transportation Research Institute (UMTRI) fellowship program. Funding for a pilot study was provided by TACOM and the Applied Dynamics, Inc. (ADI) company.

I would like to thank the six members of the dissertation committee for their support, guidance, and editorial work on the manuscript.

I appreciate the enthusiasm and support of Ric Mousseau (Ford), Roger Wehage (TACOM), and Paul Fancher (UMTRI) during the past two years. Ric's skills at "guerrilla funding" were instrumental in getting the project off to a start, and the considerable time he spent testing the software and emphasizing practical applications have strongly influenced the AUTOSIM software that was developed. Long technical discussions with Roger provided many insights into the dynamics of multibody systems. Also, the methods described in Chapter 7 stem from Roger's observations. Paul waded through almost every draft of the manuscript prepared over the past two year, and his discussions and observations were very helpful in organizing and clarifying the material.

The first three examples in Chapter 9 were used to validate the equations generated automatically with results obtained independently by Len Segel, Don Greenwood, and Ric Mousseau. The first two examples identified modeling simplifications made by Segel and Greenwood that were later incorporated into the automated methods.

The dynamics course taught by Bob Ryan was very influential to this work. Although we have never met, I would also like to thank Professor Thomas Kane (Stanford), whose approach to dynamics proved to be a strong foundation for this work.

This dissertation is in part a consequence of early encouragement from Len Segel, Tom Gillespie, James (Red) Gallagher, and Cesar Queiroz. More recent support and encouragement were provided in copious quantities by my wife, Nancy, and daughter, Samantha.

PREFACE

Simulating the behavior of mechanical systems comprised of rigid bodies and massless elements is a well-established technology that has been available with digital computers for over twenty-five years. One of the main application areas of multibody simulation is that of vehicle dynamics. With faster, cheaper, and more versatile hardware, applications involving simulation are almost limitless. Further, it is widely acknowledged that simulation is useful for wide-ranging analytical activities, such as (1) evaluating alternative designs prior to building prototypes, (2) studying the behavior of existing systems and design configurations, (3) reconstructing accidents, (4) studying the behavior of humans or hardware components via "real-time" "man-in-the-loop" or "hardware-in-the-loop" simulation. Yet, even with these recognized advantages, computer simulation of ground vehicles is not a tool used routinely by designers or other engineers. Why is this?

Just as the potential utility of simulation is widely known, it is also well known that existing software is not sufficiently convenient to meet the needs of most engineers. Although there have been great strides in devising ever better ways of formulating the equations of motion of multibody systems, and also new methods for numerically "solving" the equations to compute the behavior numerically, there is a great deal of work involved in translating these methods into robust, easy-to-use computer codes. Most engineers who need answers that can be obtained through simulation are limited in the types of computer that can be used (desktop) and the time needed to learn to use new software (a few days, at most).

At the University of Michigan Transportation Research Institute (UMTRI), formerly known as the Highway Safety Research Institute (HSRI), there is a long tradition of using computer simulation to study the behavior of ground vehicles. Even so, we face some of the same problems as engineers in industry. The computers available to us are primarily (1) desktop computers, and (2) the mainframe computer of the university. Commercial simulation software, such as the ADAMS and DADS programs, have not been feasible alternatives for us, due to the large amounts of computation required for simulating ground

vehicles. Desktop computers are too slow, and the CPU charges on our mainframe computer are too high when such large amounts of computation are involved. Further, considerable expertise in the use of the commercial codes is required to add the semi-empirical models used to represent tires and suspensions (when such additions are even possible.)

Instead, specialized simulation codes are used which are more computationally efficient because they were developed for specific vehicle models. (These include the "Phase 4" heavy truck simulation, the "Yaw-roll heavy truck model," and others.) These are large Fortran programs which were written in the 1970's and early 1980's. Unfortunately, current simulation needs never seem to exactly match the capabilities of the existing software. Hence, with every new research project, an existing program must be modified slightly to accommodate a new vehicle configuration, or to compute a new set of output variables. Modifying these large programs, and then verifying their correctness, is a daunting undertaking that limits their usefulness even here at UMTRI where they were developed.

The task of developing equations of motion for a multibody system and the task of putting those equations into a simulation code both both require a meticulous attention to detail and a considerable amount of time. Also, a specialized knowledge of dynamics and numerical analysis methods is necessary to even get started. The research reported in this dissertation was begun shortly after noting that these tasks are ideally suited to some of the technologies and methods that have been developed in the field of Artificial Intelligence (AI).

One of the basic tools of AI is Lisp, a language well-suited for symbol manipulation and prototyping other computer languages. The basic strategy in this work was to design a language suited to developing simulation codes, and to implement that language in Lisp. (In contrast, most past work in simulating ground vehicles has involved the development of equations by a dynamicist in a form that can be coded in an existing computer language by a programmer.) The programming techniques used in this work emphasize recursion, "object oriented programming," and manipulation of symbolic data, rather than numbers and matrices. These concepts are well established in computer science, and are not even considered a part of AI any more (although they are mainly the result of AI research). However, they have not yet been applied extensively to the area of multibody simulation.

A primary reason that AI techniques are not widely used in analyzing multibody systems is that Lisp and the associated programming techniques were, until a few years

ago, only feasible on mainframe computers, or on specialized (i.e., very expensive) AI workstations. Advances in computer hardware have now made these tools available on virtually all computers used by engineers, ranging from Apple Macintosh and IBM PC desktop computers to Cray supercomputers.

The objective of the research reported in this dissertation was to look at the process of developing simulation codes, and to separate the creative engineering part from the drudgery. Ideally, once the model is conceived by an analyst, the dynamics analysis and program development can be handled automatically by the computer.

Software was developed, called AUTOSIM, which demonstrates that, indeed, much of the work formerly performed by specialists in dynamics and numerical analysis can be handled automatically. Development time for a detailed simulation code is reduced from months to hours.

One of the most significant practical result of the work is that the methods are extendable to other types of engineering applications. The AUTOSIM software is essentially an extension to the existing Lisp language. On top of Lisp, it adds computer algebra, a representation of multibody systems, and a representation of a numerical analysis computer program that will be generated as output. Although AUTOSIM also happens to include a multibody formalism and design for generating a simulation code, that constitutes a relatively small part of the overall software. Engineers interested in applications that involve multibody systems can build upon the AUTOSIM language to program almost any type of analysis with only a modest incremental effort. Essentially, any job can be automated if it can be specified as a sequence of operations involving algebra, kinematics analyses, computer programming, and well-defined mathematical analyses (similar to a multibody formalism).

Mike Sayers
February 1990

Note: This copy of the dissertation was made by converting the original Microsoft Word 4 files to MS Word 6, for conversion to the Adobe Portable Data Format (PDF). It is thought to be identical to the original, except the page breaks sometimes differ by a few lines. The differences are due to changes in MS Word, not the thesis content.

April 1999

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

LIST OF APPENDICES

Appendix

1. INTRODUCTION

This dissertation deals with the modeling and computer simulation of mechanical systems composed of rigid bodies and massless force- and torque-producing elements. Motions of the rigid bodies are predicted by numerically integrating differential equations developed from principles of mechanics. The mechanical system is called a *multibody system*, the computer program that integrates the differential equations is called a *simulation code*, and the differential equations are called the *equations of motion* for the system. Multibody systems pertaining to ground vehicles are of particular interest.

1.1. Objective

The main objective of this work was to create a means for automatically generating highly efficient simulation codes for ground vehicles, while incorporating realistically modelled components. To do this, the dissertation includes (1) a software design for representing the mechanical system in symbolic form as a set of computer data objects, (2) a *multibody formalism* (i.e., a formal strategy for deriving equations of motion for a multibody system) that is valid for systems with various types of connections between the bodies, (3) methods to manipulate symbolic expressions automatically within the multibody formalism, (4) the design of an interface to the analyst that permits the description of unconventional force- and torque-producing components, and (5) a way to accommodate external computer subroutines that may have already been developed. A software package called AUTOSIM was written in the Lisp computer language to validate and demonstrate the methods. The software also includes an interface with the analyst that permits immediate evaluation of expressions involving scalars, vectors, points, bodies, etc., and the ability to generate complete simulation codes that are correct, properly documented, and reasonably easy to use.

1.2 New Research

Multibody formalisms that have been used to automatically formulate equations, whether numerically or symbolically, have not been representative of how human analysts

approach the same job. The multibody formalisms have represented well-structured analysis strategies that can be programmed easily, whereas the human analyst usually applies modeling and engineering knowledge to simplify the representation of parts of the model, if for no other reason than to reduce the algebra involved in deriving the equations. Of course, by reducing the algebra, the numerical computations based on the equations are usually reduced as well. This dissertation develops several new analysis methods that are based on concepts previously used only by human analysts.

First, an object-oriented symbolic computer language is developed so that the methods can be programmed. Data objects represent (1) algebraic expressions, (2) physical components in the multibody system, and (3) software components of the numerical simulation code being created. The algebra portion of the system is implemented using an original design such that vector expressions can be freely developed and manipulated without concern for which coordinate systems are involved. Unit-vectors are represented not as 3x1 matrices, but as primitive data objects that define three-dimensional directions. With this representation, vector operations are performed with information contained in the unit-vector objects, rather than by matrix operations as has been done in other computer algebra languages. Consequently, vector expressions can be developed and manipulated with the full degree of flexibility that a human analyst uses.

The multibody representation is also new. Components and geometric entities such as bodies, points, forces, and moments are represented with data objects tailored to describe those elements and their relationships to each other. Past methods have not directly represented elements of the system, but have instead constructed matrices to represent information related to the elements.

Finally, the inclusion of the output simulation code as a group of objects subject to automated manipulation is a new approach.

The simplification methods developed in this work incorporate methods that have not been used before in automated multibody analysis methods. In previous work, algebraic simplifications have been based only on rules of algebra. In this work, simplifications are based not only on rules of algebra, but also on (1) engineering judgements (e.g., terms that can be proven to be numerically negligible are thrown out), (2) recognition of modeling equivalences (e.g., grouping inertial terms to form composite bodies), and (3) programming techniques used for numerical analysis (e.g., recursion, factoring out constants that can be precomputed, introducing intermediate variables, etc.).

Another product of the research is a “complete” symbolic multibody formalism that permits the analyst to include any forces and torques that can be modeled mathematically, even if the models are unorthodox. Also, output variables computed by the simulation code can be defined by the analyst in terms of arbitrary combinations of directions from all coordinate systems present in the multibody system. (Past symbolic multibody formalisms place many restrictions on the sorts of forces and variables that can be referenced.) In addition to dealing with issues of multibody dynamics, the formalism includes considerations of how the equations of motion are eventually programmed for numerical solution.

A symbolic method is presented for expressing the equations of motion in explicit form, eliminating the need for numerically solving sets of simultaneous equations. For vehicle systems, the symbolic method can be much more efficient than numerical methods commonly used.

Techniques are developed for handling constraints in a more automated manner than has been possible before with symbolic multibody analyses. Nonholonomic constraints and closed kinematic loops are described by the analyst with simple vector expressions that are processed automatically to obtain scalar constraint equations.

Kinematic “closed loops” (e.g., four-bar linkage, slider-crank mechanism, etc.) with complicated constraint equations are handled by automatically writing numerical computation code to satisfy the constraints in such a way that singularities are unlikely to occur. That is, the symbolic computation methods are used not only to derive conventional differential equations, but also Jacobian coefficients and source code to recursively compute values for variables when closed-form solutions are not feasible.

The combined effect of these new techniques is significant in at least two ways: (1) the simulation codes generated are more efficient for vehicle dynamics models than any other formulations that have been published, and (2) the input description prepared by the analyst is minimal and does not require knowledge of the formalism details.

1.3. Organization of Dissertation

Chapter 2 presents background material for the work, covering key concepts and previous work. Chapter 3 summarizes conventions in terminology and notation used throughout this dissertation. Chapter 4 describes the sort of numerical computer code that is desired as the output of the automated symbolic software.

Chapter 5 develops the computer representation of symbolic objects needed to completely describe a multibody system and a simulation code. Most of this material deals with computer algebra, and the representation and manipulation of vector/dyadic quantities.

Chapter 6 presents an overview of the dynamics theory needed to develop the formal strategy for automatically analyzing multibody systems. The method advocated by Kane for manually analyzing a system is presented, and then extended to include details for formulating the equations of motion for numerical solution.

Chapter 7 develops the symbolic solution for sets of simultaneous linear equations, such as those obtained by the method presented in Chapter 6.

Chapter 8 presents the multibody formalism. It describes how the model conceived by the analyst is described in simple terms and translated into a computer representation. Once the system is described, the equations of motion are developed automatically and a self-contained Fortran simulation code is generated.

Chapter 9 describes six example multibody systems that were analyzed using AUTOSIM and presents results of investigations into the significance of the various techniques. Conclusions are summarized in Chapter 10.

Appendices are included to provide more detail about the examples. Appendix A briefly describes the AUTOSIM commands used in the examples. Appendices B, C, D, and E contain Fortran source code generated by AUTOSIM for some of the examples.

Those readers interested in all details of the work are encouraged to read the ten chapters in sequence. Those interested mainly in the practical aspects can skip right to Chapter 9, and refer to Chapter 3 and Appendix A as necessary to understand the conventions and notation. Those interested mainly in the dynamics formalism should read Chapters 3, 6, 7, and 8. Readers interested mainly in the symbolic computation methods should read Chapters 3 and 5, and skim through 7 and 8.

2. BACKGROUND

The numerical simulation of multibody systems has been receiving an escalating amount of attention in the past twenty years. The interest has been driven in part by the ever-increasing capabilities of the digital computer, both in the areas of hardware performance and and in programming concepts. Further, increasing design challenges for complex spacecraft, robot manipulators, and high-speed mechanisms mandate simulation during the design process. Even when established mechanical systems such as ground vehicles are considered, simulation is essential for (1) designing future products in a globally competitive environment, (2) evaluating the suitability of novel vehicle configurations on public roads, (3) reconstructing accidents, and (4) investigating the behavior of humans in simulated conditions through driving simulators involving "real-time" "man-in-the-loop" simulation.

Given that most complex multibody systems that are of greatest interest can only be understood with the aid of computer simulation, modern textbooks in dynamics now emphasize analyses suited for computer solution (e.g., [35, 58]). Symposia and sessions have been held on the subject of multibody systems [16, 38, 77], and specialized textbooks are starting to appear that describe multibody dynamics from the perspective of programming the dynamics in a computer algorithm [25, 97]. In the literature of mechanical dynamics, papers dealing with analytical and computational methods pertaining to multibody systems are too numerous to cite here (for example, Ref. [97] includes 257 citations). However, several overviews are available [38, 63, 74, 110, 113, 135].

The job of simulating a multibody mechanical system involves three steps: (1) creating an idealized model of the system, (2) formulating equations of motion, and (3) solving the equations numerically. The first step is the most critical, for it requires the creative application of engineering knowledge and judgement to determine (a) what characteristics of the system are important, (b) what characteristics should be neglected, and (c) a strategy for modeling the important characteristics using rigid bodies, massless springs, and other idealized elements. The third step can be performed by numerically integrating the nonlinear ordinary differential equations of motion with a computer program called a *simulation code*. It is the second step that is of primary interest in this work.

Approaches that are taken to formulate and solve equations for a system after a model has been conceived by an analyst can be organized into three categories:

1. Equations of motion of the multibody system are derived by the analyst and translated by a programmer into a *specialized simulation code* that pertains to one particular multibody system.
2. A *generalized simulation code* is used in which the equations have been formulated and programmed once and for all in a generalized fashion.
3. *Symbolic analysis software* is used to aid the analyst and programmer in the formulation of equations and the development of a specialized simulation code.

The manual derivation of the equations of motion for even a modestly complex system is a tedious undertaking that involves considerable algebra, a nagging uncertainty regarding the correctness of the equations, and a considerable programming and debugging effort. To avoid these problems, the process of formulating equations is automated in the second and third of the above approaches, which are discussed at length in the following two sections.

2.1. Generalized Simulation Codes

Generalized (numerical) simulation codes are computer programs that employ a multibody formalism established for all systems. They first build a set of equations from a description provided by the analyst, and then proceed to numerically integrate the equations to simulate behavior of the system. Available multibody codes that are used for simulation of spacecraft, robots, mechanisms, biomechanics, and vehicles have been reviewed elsewhere [39, 63]. With respect to simulations of ground vehicles, the codes ADAMS [3, 11, 18, 36, 76, 79, 86, 87] and DADS [85, 121] are primarily used by industry in North America. These, and other generalized codes (e.g., [2, 26, 27, 80, 88, 95, 115, 116, 117]), are appealing to many engineers because they offer a “complete solution” that handles the entire simulation effort, from model description to the numerical integration of equations. Of course, there are some compromises made to achieve the generality.

One compromise is that the generalized codes often run slowly relative to specialized simulation codes. A human dynamicist usually tries to obtain equations of motion that are as simple as possible, using a number of techniques that will be detailed later. Further, good programmers can improve computational efficiency when the equations are incorporated into the simulation code. Because general-purpose simulation codes are

written to apply to *all* multibody systems, most simplification techniques cannot be used. For vehicle simulations, the eventual difference in simulation speed between a special-purpose code and a generalized code can be more than an order of magnitude (differences in run-time speeds have been observed to vary by a factor ranging from 10 to several hundred). The inefficiency of the general-purpose software makes it less than ideal for highly repetitive design studies, and unfeasible for real-time, hardware-in-the-loop operations.

Another compromise is that the generalized codes are not completely general when it comes to introducing force- and torque-producing components. This can be a problem with multibody systems that include elements characterized by semi-empirical models that are not likely to have been fully anticipated by the programmer. For example, ground vehicles include tires, nonlinear springs, complex shock absorbers, etc. that are modelled differently based on the intended use of the simulation. Assuming that an engineer is able to develop a computer representation of such an element as an *external subroutine*, the subroutine must be incorporated into the multibody simulation. If the simulation program is written by hand, it is a simple matter to incorporate external subroutines. However, for a generalized simulation code, external subroutines are limited to cases that were anticipated by the original programmer. Variables needed as inputs to the external subroutine (positions, angles, speeds, etc.) are not always readily available, and may require the analyst to develop interface software to compute the needed values from variables provided in the multibody program.

2.2. Symbolic Analysis by Computer

Symbolic computation offers the potential to combine the high reliability of a general-purpose code with the efficiency and modeling flexibility associated with the development of a new special-purpose code. In this approach, the computer generates a simulation code that is similar in structure and efficiency to one written by a human programmer.

There are three variations on this approach that have been taken for performing the symbolic computation needed for analyzing multibody systems:

1. A generic symbolic manipulation language is used by a dynamicist who performs the analysis in the same manner as would be done “by hand,” except that the computer aids in performing the algebra.

2. A symbolic manipulation language tailored for dynamicists is used by an analyst who guides the analysis, but uses the computer to perform algebraic manipulations and to apply routine kinematical and dynamical formulas.

3. A complete, self-contained multibody analysis program is used to formulate equations automatically, based on a description of how bodies in the multibody system are connected to each other.

Generic Computer Mathematics Languages

Generic symbolic mathematics software has been employed to develop equations of motion for multibody systems. Most of the work reported to date has been done with the MACSYMA language [21, 33, 34, 44, 45, 73, 81, 82, 92], possibly because it has been available on mainframe computers for over fifteen years. Other generic symbolic languages that have been used are FORMAC [69] and REDUCE [67, 94]. Newer languages with similar capabilities are MAPLE [20], MuMath [139], and Mathematica™ [138]. MuMath and Mathematica can be used by a much greater audience than MACSYMA, as they run on personal desktop computers. Further, more commercial symbolic computation languages are rapidly appearing for the new generations desktop computers.

The generic mathematical languages include capabilities far beyond the basic "high-school algebra" needed for analyzing multibody systems. For example, the language MACSYMA consists of about 3000 compiled Lisp functions, accounting for over 300,000 lines of Lisp source code [92]. In past work, powerful computers have been required for acceptable performance [82]. Also, the analyst must not only be an expert at dynamics, but also in the use of the symbolic computer language.

Published equations generated with computerized symbolic manipulation have not been particularly efficient [82]: the main advantages of this approach have been (1) that algebra errors on the part of the analyst are eliminated, and (2) that the time needed by the analyst to obtain the equations is reduced.

Part of the difficulty in using generic computer mathematical languages stems from their lack of capability to represent the vector and dyadic expressions that occur naturally when analyzing multibody systems. For example, consider a system that includes rigid bodies A, B, and C, where A is connected to ground by a hinge joint, B is connected to A by a hinge, and C is connected to B. Using unit-vectors fixed in each body, the angular velocity of C might be written by an analyst as

$$\vec{\omega}^{C} = u_1 \vec{\mathbf{n}}_1 + u_2 \vec{\mathbf{a}}_2 + u_3 \vec{\mathbf{b}}_3 \qquad (2.2.1)$$

where u_1, u_2, and u_3 are state variables with units of rotational speed, and $\vec{\mathbf{n}}_1$, $\vec{\mathbf{a}}_2$, and $\vec{\mathbf{b}}_3$ are unit-vectors fixed in bodies N (ground), A, and B, respectively. This vector expression is written without concern for the coordinates needed to represent the unit-vectors $\vec{\mathbf{n}}_1$, $\vec{\mathbf{a}}_2$, and $\vec{\mathbf{b}}_3$. As such, it cannot be represented in any of the languages cited thus far. Instead, a coordinate system must be chosen so that the vector can be represented by an array of three scalar expressions, where each expression corresponds to one coordinate in the chosen coordinate system. 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 kept track of, and (2) it is not always clear right away which is the "best" coordinate system to choose.

Computer Mathematics for Dynamics

At least one symbolic computation language has been developed specifically for interactive use by a dynamics expert [106]. With this language, called AUTOLEV (**Auto**mated **Lev**inson), the dynamicist guides the analysis by introducing state variables, defining coordinate systems, etc. Essentially, the dynamicist analyzes the system using Kane's method (described in Chapter 6), 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 complete, self-contained Fortran program that is ready to compile and run as a simulation code customized for the multibody system that was just analyzed. Although the analyst is required to be well-versed in the Kane method of dynamic analysis, the software is simpler to use than other symbolic mathematics computer languages. It is designed for use on the IBM PC, and is therefore more accessible than many symbol manipulation languages. AUTOLEV does not automatically generate nonholonomic constraints or constraints for kinematic loops, nor does it have a facility for solving linear equations in symbolic form.

Automated Symbolic Multibody Analyses

With a sufficiently detailed multibody formalism, equations of motion can be developed automatically using only rudimentary computer algebra. Self-contained symbolic multibody codes have been written to formulate equations that can be merged into a simulation program. This area of symbolic multibody analyses is the most significant for the research reported in the dissertation. In the following summaries, the dynamical

formulations used in the programs are noted, but will not be described in any detail until Chapter 6.

Rosenthal and Sherman developed the symbolic multibody program SD/FAST, known earlier as SD/EXACT [100, 101]. SD/FAST demonstrates that highly efficient equations of motion can be derived automatically by a self-contained program with its own built-in computer algebra capabilities, using Kane's equations [55, 58]. SD/FAST includes provisions for dealing with some kinds of closed kinematical loops via Lagrange multipliers [35, 97]. If a system has a linkage involving ball joints and pins, the constraints can be handled automatically.

A modified, highly recursive version of Kane's analysis method was published by Wampler in his PhD dissertation [125]. Wampler isolated portions of acceleration terms, called acceleration remainders, to build equations of motion in a form that is explicitly suited for numerical solution. Some of his techniques are also used in this dissertation. The main application of Wampler's dissertation involved robotics, so the techniques are not completely generalized. They include provisions for actuator dynamics, and lack provisions for topologies other than chains. Wampler demonstrated his formalism with several numerical multibody codes that offer efficiency better than anything available at that time (1985).

The methods of Wampler were adopted by Nielan, who wrote a computer program called SYMBA to generate multibody equations symbolically, in a fashion similar to SD/FAST [83]. Nielan also included part of the more general Kane formulation, to handle topologies other than chains. The formulation obtained by Nielan produced equations for a robotic system called "The Stanford Arm" that were the most efficient obtained to date (1986) [45, 57, 82, 124, 125]. For systems more characteristic of spacecraft, the formulations were comparable to those obtained by SD/FAST.

There are serious practical limitations in both SD/FAST and SYMBA with respect to ground vehicle simulations. First, they only derive expressions for inertia forces and inertia torques. That is, the programs offer no way for the analyst to specify active forces and torques, as is normally done with generalized simulation codes such as ADAMS. The analyst must still develop some of the equations by hand, program them, and manually merge the code generated by SD/FAST or SYMBA with the hand-written code. The inertia forces and inertia torques dominate equations of motion in some fields, notably spacecraft

and robotics simulation. However, equations of motion for ground vehicles have fairly simple inertia terms and very complicated force and torques descriptions. For ground vehicle models, the bulk of the simulation code would still have to be developed by hand.

A second limitation is that SD/FAST or SYMBA cannot generate and apply nonholonomic constraints. Although SD/FAST will generate code with constraint coefficients, it is up to the analyst to obtain expressions for those coefficients and manually edit them into the simulation code.

A third limitation is that the above symbolic computation algorithms keep all nonlinear terms in all equations. A human analyst typically throws out terms that are known to be small, such as some Coriolis accelerations, products of small trigonometric functions, etc. Even for a simple vehicle model studied as an example, these simplifications were shown to have improved efficiency by a factor of 3 [103].

Several symbolic multibody programs have been developed in Europe specifically for handling vehicle systems. These are NEWEUL, MESA VERDE, and MEDYNA. Unlike the SD/FAST and SYMBA software, these programs properly include forces and moments from tires, suspensions, and other force- and moment-producing components in a multibody system. Also, they can be used to derive linearized equations of motion.

The program NEWEUL is based on a multibody formalism that can be applied to systems of rigid bodies constrained by both holonomic and nonholonomic constraints. The program has been used for applications involving ground vehicles, spacecraft, and robots [40, 64, 65, 66, 108, 109, 111, 112]. The nonlinear terms in the equations of motion appear in an isolated matrix, making it simple to obtain either linearized or fully nonlinear equations. (However, it is not possible to derive equations where some variables are small and others are not.) The NEWEUL program was originally written in Fortran and performs the symbolic manipulations by representing expressions by integer codes in arrays [112]. This method permits symbolic manipulation in a language not designed for that purpose, but with severe limits in comparison with other symbolic manipulators. NEWEUL cannot represent “nested” expressions. That is, products of sums are always “expanded.” (For example, the expression (A + B)*(C + D) contains two “nested” sums: (A + B) and (C + D).) Also, the automated replacement of repeated expressions is not performed by NEWEUL. Hence, equations generated by NEWEUL that have been published include many redundant expressions that would be taken out by almost any human programmer. Further, when symbolic expressions are always expanded, great demands are placed on the computer resources, and analysis of complex systems becomes

impossible. For example, for a simple three degree-of-freedom vehicle model, the computational efficiency of equations derived with nested expressions and intermediate expressions was about a factor of seven better than when those capabilities were disabled [103]. The exact form of the input to NEWEUL has not been described in the literature, other than to define matrices that must be provided by the analyst using NEWEUL.

The program MESA VERDE stands for "**ME**chanism, **SA**tellite, **VE**hicle and **R**obot **D**ynamics **E**quations" and is based on a multibody analysis strategy developed by Wittenburg and Roberson, and programmed by Wolz [136, 137]. The multibody system is described by several matrices with integer and symbolic elements, which are provided as inputs by the analyst. The analyst can define state variables in a variety of ways, and can apply arbitrary force- and moment-producing elements. An interesting method is used to specify closed kinematical loops [72]. One body in the loop is entered twice, with half the mass and inertia each time. MESA VERDE is told that the two entered bodies are the same, and constraint equations are automatically generated. The output of the program is the set of equations of motion, written in either PASCAL or FORTRAN. Example equations generated by the program have not appeared (to this author's knowledge) in the English literature, nor have example input requirements. However, the published descriptions indicate that the analyst is expected to be familiar with the multibody representation developed by Wittenburg and Roberson.

The program MEDYNA [24, 52, 62] formulates equations of motion in ACSL (Advanced Continuous Simulation Language) that are linear with respect to state variables, but which can involve nonlinear force and moment-producing elements. MEDYNA was developed specifically for ground vehicles, especially those that travel on tracks and guideways. Although its multibody dynamics analysis is limited to a linear representation, it includes a number of pre-defined component models such as tires, railcar wheels, etc. to facilitate its use with vehicles.

2.3. Research Approach

Although simulation of mechanical systems is widely acknowledged as a necessary engineering tool, the technology is not yet mature in the sense that simulation of complex nonlinear systems is not performed with the ease of other engineering analyses, such as linear system analysis, CAD, etc. Each of the three general approaches described at the start of this chapter has associated limits.

1. To develop a simulation code without computer aid requires (1) expertise in dynamics, (2) a great deal of time to perform the pencil-and-paper analysis of the system to be simulated, and (3) more time and programming expertise to put the equations of motion into a form that can be solved by computer.

2. To use the generalized simulation codes described in Section 2.1, the engineer must (1) have access to general-purpose simulation software and computer powerful enough to run the software (generally, a mainframe computer or minicomputer), (2) have extensive experience in dynamics, and (3) also have extensive knowledge and experience with the simulation software. Even with the required software, hardware, experience, and knowledge, the generalized simulation codes may require too much computer time per run to be used for some analyses.

3. The symbolic analysis software packages mentioned in the previous section serve to aid the analyst (in various degrees) in the development of simulation codes as might be done "by hand." However, much of the work must still be done by the analyst, particularly in specifying forces and torques acting on bodies in the system, and in specifying constraints. Also, the symbolic programs generate only a portion of the overall specialized simulation code.

The third approach is the most recent, and least developed. With recent developments in computer programming from artificial intelligence applications, many procedures that used to be difficult or impossible to program can now be automated. Symbolic multibody programs developed previously have been developed from the view: "Given the ways that data can be represented symbolically in existing computer languages, how can equations of motion for a mechanical multibody system be generated automatically?" Multibody formalisms have been complicated, in order to compensate for the limited representation possible in a conventional computer language. Because the symbolic manipulation capabilities have been rudimentary, some important simplification methods have not been applied. (Simplification techniques that are not included in the computer algebra can still be applied by including them in the multibody formalism, as is the case with the linearization option in NEWEUL, but there is a loss of modeling flexibility because the formalism must include specific "plans" for dealing with all algebraic combinations that can occur in the systems being modeled.) The methods are less general than approaches taken by human analysts, because all of the possible combinations of multibody systems must be anticipated in the formalism. Also, the simulation codes are generally not nearly as efficient as can be obtained with better symbol manipulation capabilities.

In contrast, this dissertation takes the view: "How can the ways that humans analyze multibody systems be automated?" To start, a symbolic mathematics language called AUTOSIM is developed in Lisp to **auto**matically generate **sim**ulation codes [104]. Although a number of mathematical symbol manipulation languages exist, none have the capabilities needed to easily support the generation of efficient simulation codes. Much of the literature in computer symbol manipulation has focused on the manipulation of polynomials and the development of integrals and derivatives of complex expressions [12, 17, 92, 94, 123, 127]. In contrast, AUTOSIM neglects these types of manipulations. Instead, the language is built upon representations of three aspects of the overall system in symbolic form as 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 the numerical simulation code being generated.

The above data have been represented with scalar expressions and matrices in all work done prior to this. The representation of vectors and dyadics in AUTOSIM is similar to the "component-free vectors" in MuMath, except that the dot-product operator generates a scalar expression (using knowledge of the multibody system), rather than a symbolic vector expression.

Another novel aspect of AUTOSIM is the way in which intermediate expressions are introduced to obtain efficient Fortran code. All of the symbol manipulation languages can print equations in FORTRAN, but AUTOLEV is the only one that identifies subexpressions that can be replaced by intermediate variables. None of the existing software packages have the capability to identify constants that can be precomputed. (To obtain maximum efficiency with existing software, the analyst is required to specify numerical values for all parameters. A new set of equations must be formulated if any of the parameter values are changed.)

Next, a multibody formalism is needed that parallels the process employed by a human analyst. The formalism must be specified in sufficient detail that it can be programmed in the new symbolic language. Ideally, the best features of existing multibody simulation methods should be included, namely:

- A “complete solution” such as is provided by many generalized simulation codes minimizes the time needed to proceed from a model concept to a working simulation code if the process is entirely automated.
- Virtually any force- or moment-producing component can be included in a simulation code developed by hand, even if the force/moment characteristics include such behavior as (1) friction, hysteresis, and other discontinuities; (2) behavior that is dependent on its past history (in addition to current states); and (3) dynamic behavior of variables not directly a part of the multibody system.
- Virtually any motion variable, no matter how unorthodox, can be defined for use in codes developed by hand. Such variables are typically needed as inputs to external subroutines, or as output variables.

  The generalized codes, both numerical and symbolic, are (hopefully) debugged once and for all. Once debugged, equations produced are always valid and correct.
- The symbolic methods (automated and manual) can result in highly efficient simulation codes, needed for use with desktop computers or for interacting with hardware in real-time applications.

The advantages can be combined by creating a generalized symbolic multibody analysis program that offers a “complete solution” and still allows the modelling freedom available when simulations are developed by hand. Accordingly, AUTOSIM formulates equations of motion and then generates a complete simulation code. When the source code for the simulation is compiled, the resulting program reads input files with parametric data, simulates the system, and writes output files in a form suitable for automated post-processing software.

To match the “ease of use” that can be obtained with specialized simulation codes, the analyst describes input parameters and output variables with algebraic expressions, such that the simulation code generated by AUTOSIM reads input and writes output that is exactly the input and output of interest to the *end user*—the engineer using the simulation code generated by AUTOSIM. To accomplish this, the analyst can define multibody parameters as arbitrary expressions involving constants familiar to the end user. (For example, the location of the mass center of a vehicle might be described with an expression involving a wheelbase and static axle loads.) Any force, torque, or motion variable can be specified as an output without referring directly to state variables. External subroutines and

functions can be freely introduced into the system to handle complicated and unusual elements.

3. CONVENTIONS

There is no standard convention for describing the elements and topologies of multibody systems. Even with established areas of mathematics such as vector algebra, different notations are used by different authors. In the hope of simplifying the discussions that follow in subsequent chapters, this chapter is included to detail the conventions used throughout this dissertation.

The first convention is that a word or phrase that represents an important technical concept is shown in italics the first time it appears, unless the name is already well-established. A definition is usually supplied in the material that follows. Subsequent appearances of the term or phrase appear in normal typeface.

3.1. Elements in a Multibody System

The multibody systems under consideration in this dissertation are mechanical systems composed of rigid bodies and massless elements that apply forces and torques to the bodies.

Rigid Bodies, Reference Frames, and Coordinate Systems

A *reference frame* is an environment in which points remain fixed with respect to each other at all times. A *rigid body* is an object in which every point is fixed in the same reference frame. Thus, each rigid body in a mechanical system has an associated reference frame. It is possible to conceive of reference frames for which there are no corresponding rigid bodies. For example, consider a rolling disk. In addition to the reference frame that rolls with the disk, it is convenient to define an auxiliary reference frame that follows the disk but which does not roll.

In the remainder of this dissertation it is not essential to distinguish between reference frames and rigid bodies, and some of the descriptions are simplified by using the two names interchangeably. That is, rather than writing "the reference frame associated with body B," the shorter phrase "body B" is used. Reference frames that do not correspond to

a physical body in the system are treated as rigid bodies with zero mass and zero moments of inertia. In the rolling disk example, the system is modelled as two rigid bodies: (1) a massless body A which (a) slides over the ground, (b) steers, and (c) leans, and (2) a body B that spins relative to A and has mass and inertia .

A *coordinate system* is a numbering convention used to assign a unique ordered trio of numbers to each point in space. All coordinate systems involved in this work are right-handed Cartesian coordinate systems, defined by three mutually orthogonal axes intersecting at an origin. Further, each coordinate system is fixed in one of the bodies of the system. In general, any number of coordinate systems can be defined for a given body. However, in this work, only one coordinate system is introduced with each body. That is, there is a one-to-one correspondence between bodies, reference frames, and coordinate systems for all multibody systems as they are described in this dissertation.

Joints and Constraints

Kinematic relationships between bodies are defined by *joints.* In the context of how a multibody system is described, a joint defines a set of zero or more *holonomic* constraints that limit the geometric relationships that are possible between the bodies. Forces, torques, and speeds are not factors in a holonomic constraint. Holonomic constraint equations are called *rheonomic* if they include explicit functions of time, and *scleronomic* if they do not. Unless stated otherwise, holonomic constraints are assumed to be scleronomic.

Joint constraints are handled in two ways in the multibody formalism developed later. Most of the joints appear in a *tree topology*, as described in Section 3.4. Additional joints, if they exist, are handled by adding constraint equations.

In addition to the holonomic constraints applied by joints, a system may also be subject to *nonholonomic constraints.* These are constraints on motion but not position or orientation. For example, a two-axled vehicle slowly navigating a turn is constrained such that the instantaneous velocity vector of each wheel center is oriented in the same direction as the wheel. That is, there is no lateral slipping. Thus, movement of the vehicle from one position to another is constrained. However, the vehicle is not limited with respect to the positions it could possibly occupy after sufficient maneuvering.

3.2 State Variables

A simulation code computes values for a number of *output variables* at discrete points in time, based on initial conditions, applied forces and moments, and the parameter values for the system. The set of variables written as output by the simulation code is completely arbitrary and can be defined as the analyst sees fit.

To compute the output variables, a set of differential equations is numerically integrated. Those differential equations are written in terms of *state variables*. The state variables are selected to mathematically describe the state of the system, such that any position or speed variable of interest can be written as an explicit function of the state variables. Selecting state variables is an essential analytical step that will be developed at length in Chapter 8. The state variables are grouped into nested categories, shown schematically in Figure 3.2.1 and listed in Table 3.2.1.

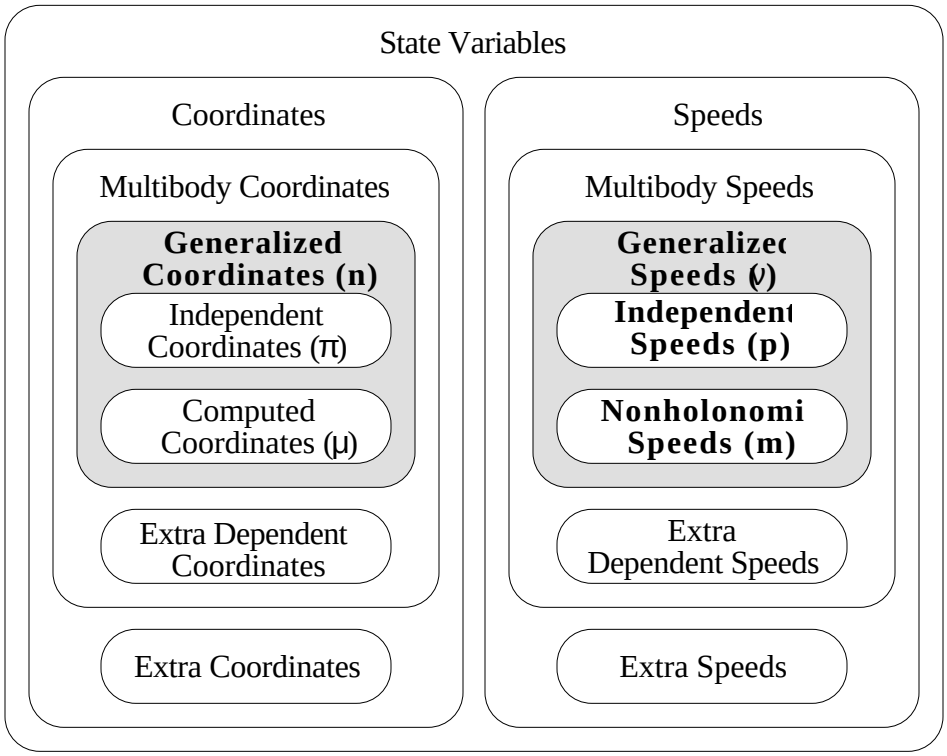

Figure 3.2.1. Categories of state variables.

Table 3.2.1. Categories of state variables.

Name	Nomenclature	Description
Independent coordinates	$q_1, \ldots q_\pi$	Generalized coordinates computed by integrating their derivatives.
Computed coordinates	$q_{\pi+1}, \ldots q_n$	Generalized coordinates computed both by integrating derivatives and with iterative numerical procedures. ($\mu = n - \pi$)
Dependent coordinates	expressions	Coordinates defined as functions of time or as explicit functions of independent and computed coordinates.
Extra coordinates	symbols	Coordinates added by the analyst that are not a part of the multibody system.
Independent speeds	$u_1, \ldots u_p$	Speeds computed by integrating their derivatives.
Nonholonomic speeds	$u_{p+1}, \ldots u_\nu$	Speeds defined as linear combinations of independent speeds. The dependencies involve forces and moments of constraint that influence the system dynamics. ($m = \nu - p$)
Extra Dependent speeds	expressions	Speeds defined as linear combinations of independent speeds. The dependencies do not involve forces and moments of constraint that influence the system dynamics.
Extra speeds	symbols	Speeds added by the analyst that are not a part of the multibody system.

The figure shows how the most specific categories are nested. The broadest set includes all state variables, and is divided into two groups: *coordinates* (scalar variables involving position) and *speeds* (scalar variables involving velocity). Coordinate variables are further divided into two sets: *multibody coordinates* and *extra coordinates*. The multibody coordinates include all variables that represent displacement in either translation or in rotation between the rigid bodies. Multibody coordinates have units of length or angular displacement (e.g., in, rad). The extra coordinates are variables added by the

analyst that are computed in the simulation code, but which do not describe kinematics of the mechanical system. For example, air pressure in an accumulator might be a variable in a hydraulic system coupled to the multibody system. Similarly, the speeds are grouped in two categories: *multibody speeds* and *extra speeds*. The multibody speeds include all variables that describe velocity between rigid bodies, and have units of length/time or angular rate (e.g., in/sec, rad/sec). Extra speeds are variables added by the analyst that are not directly related to the multibody system.

The choice of whether an extra variable is classified as a speed or a coordinate is made by the analyst and is arbitrary. These are simply variables, not directly related to the multibody system kinematics, that are computed by integrating their derivatives in the simulation code.

The multibody coordinates are divided into two sets: *generalized coordinates* and *extra dependent coordinates.* The extra dependent coordinates are variables that can be written as explicit functions of generalized coordinates. When such expressions are found, they are used in place of the coordinates wherever they appear in the equations of motion. Thus, the extra dependent coordinates are coordinates that were removed from the equations.

The multibody speeds are divided into *generalized speeds* and *extra dependent speeds.* The dependent speeds are variables that can be written as functions of generalized speeds, *and which have no influence on forces and moments of constraint*.

The extra coordinates, extra speeds, extra dependent coordinates, and extra dependent speeds are shown in the figure and table for the sake of completeness, given that the AUTOSIM software has provisions for using them. However, they do not appear in the remainder of the dissertation. It is the four categories enclosed in the shaded boxes in the figure that are used in the methods developed later, and which are described further below.

The generalized coordinates are introduced to allow the position of any point in the multibody system to be written in terms of those coordinates and system parameters. The generalized coordinates are further divided into two groups: *independent coordinates* and *computed coordinates.* The independent coordinates are the coordinates that can be computed only by integrating their time derivatives. The computed coordinates are variables that are defined implicitly as functions of independent coordinates by constraint equations. If a constraint equation is “solved” to obtain one coordinate as an explicit function of the others, it is classified as an extra dependent coordinate and removed from the equations of motion. However, when the constraint equations are too complicated,

explicit solutions are not easily found. Instead, numerical alternatives are used that will be described in section 8.3. As indicated in the figure and table, there are n generalized coordinates, π independent coordinates, and μ computed coordinates, where $n = \pi + \mu$.

The generalized speeds are defined such that the speed of any point in the multibody system can be written in terms of system parameters, the generalized coordinates, and the generalized speeds. The generalized speeds are divided into two sets: *independent speeds* and *nonholonomic speeds*. The independent speeds are the speeds that can be computed only by integrating their time derivatives. Nonholonomic speeds can be written as functions of the independent speeds. The nonholonomic speeds are mathematically not independent due to the presence of forces and moments of constraint that influence the behavior of the system. (The nonholonomic speeds are distinguished from the extra dependent speeds that are not incorporated in the equations of motion.) There are ν generalized speeds, p independent speeds, and m nonholonomic speeds, where $\nu = p + m$.

In all six of the examples in Chapter 9, the state variables are defined such that $n = \nu$, $m = \mu$, and $p = \pi$. However, the multibody formalism developed in Chapter 8 is made more general by not presuming this relationship.

Table 3.2.1 showed the convention for writing the state variables. Their time derivatives are written by putting a dot over the variable. When the state variables are printed by AUTOSIM as Fortran source code, the generalized coordinates are printed as an array named Q (e.g., Q(1), Q(2), etc.), the derivatives of the coordinates are printed as an array named QP (e.g., QP(1), QP(2), etc.), the independent speeds are printed as an array U, and the derivatives of the independent speeds are printed as an array UP. Symbols for dependent coordinates, dependent speeds, and nonholonomic speeds are not used. These variables are always replaced with expressions involving independent variables.

3.3. Notation

The analyses presented in the following sections are developed from the method for analyzing a multibody system described by Kane and Levinson in their dynamics textbook [58]. Their notational conventions are adopted, although some modifications have been made with the intent of improving clarity and simplicity within the scope of this dissertation.

Subscripts and Superscripts

Subscripts are used (1) to distinguish related coefficients and variables, (2) to identify array elements, and (3) to annotate parameters.

In many of the formulations that follow, a convention is employed in the letters used as subscripts. A subscript i indicates a number between 1 and n, where n is the number of generalized coordinates; a subscript j refers to a positive number associated with a body or joint (e.g., rotational degrees of freedom of a joint); a subscript o is an offset that relates a joint index j to an index i for the generalized coordinates (that is, i = j + o); a subscript r is a number between 1 and p, where p is the number of independent speed variables; and a subscript s designates the index of a dependent variable (either a nonholonomic speed or a computed coordinate).

Superscripts are used to identify points, bodies, and reference frames associated with a variable or parameter. (Several examples appear in following subsections.)

Bodies and Points

Bodies are identified with capital letters written in a plain typeface. In following material, N always refers to the inertial reference, B generally refers to an arbitrary body under consideration, and A refers to the parent of B.

Each body has three associated points that are written according to a standard convention. For body B, these points are: (1) B_0, the origin of the coordinate system associated with B, (2) B*, the center of mass of B, and (3) B_J, the "joint point" of B, which is a point fixed in A that coincides with B_0 when all generalized coordinates are zero.

Other points are written as capital letters, and are defined as they are introduced.

Vectors and dyadics

A *unit-vector* is a fundamental algebraic element that has a unity magnitude and defines a direction in three-dimensional space. A *vector* is a sum of one or more products of unit-vectors and scalars. Vectors and unit-vectors are designated with an overhead vector arrow, e.g., $\vec{\mathbf{r}}$. A *dyad* is a notational convenience that occurs when two vectors appear side by side in an expression,[1] and a *dyadic* is an expression that contains dyads. A dyadic

---

[1] A dyad is a simple product of two unit-vectors. For example, the expression $(\vec{a} \bullet \vec{b})\,\vec{c}$ is a vector

is designated with a dyadic arrow ($\vec{\vec{\mathbf{I}}}$). Vectors and dyadics represented by English letters are also shown in boldface.[1]

The *coordinate system* associated with each body is defined by three axes whose directions are defined by three mutually orthogonal unit-vectors, and which all pass through the *origin* (a point). The unit-vectors are named with a lower-case letter that matches the body, and subscripted with indices 1, 2, and 3. For example, the unit-vectors for the inertial reference N are $\vec{\mathbf{n}}_1$, $\vec{\mathbf{n}}_2$, and $\vec{\mathbf{n}}_3$. Similarly, the unit-vectors for body B are named $\vec{\mathbf{b}}_1$, $\vec{\mathbf{b}}_2$, and $\vec{\mathbf{b}}_3$. A dyadic called a *basis dyadic* is associated with each body and is obtained by "doubling" the unit-vectors. For body B, the basis dyadic is $\vec{\vec{\mathbf{b}}} = \vec{\mathbf{b}}_1\,\vec{\mathbf{b}}_1 + \vec{\mathbf{b}}_2\,\vec{\mathbf{b}}_2 + \vec{\mathbf{b}}_3\,\vec{\mathbf{b}}_3$. The dot product of a basis dyadic and a vector is equivalent to the original vector. That is, it is algebraically equivalent to multiplying by unity. However, the result is expressed in the basis of B, such that the only unit-vectors appearing in the expression are $\vec{\mathbf{b}}_1$, $\vec{\mathbf{b}}_2$, and $\vec{\mathbf{b}}_3$.

In Chapter 9, example multibody systems are analyzed and a great deal of computer input and output code is listed. The computer printouts include no formatting, such as bold typeface, subscripts, vector arrows, etc. In that chapter, unit-vectors are written with enclosing square brackets. Symbolic names are written in upper-case letters if they are "outputs" from the computer, and in lower-case letters if they are "inputs" from the analyst. For example, the unit-vectors $\vec{\mathbf{n}}_1$, $\vec{\mathbf{n}}_2$, and $\vec{\mathbf{n}}_3$ might be written [N1], [N2], and [N3], or, as [n1], [n2], and [n3]. (Upper or lower case is not significant with respect to the meanings of symbols.)

Position, Velocity, Acceleration, and Derivatives

Vectors representing position are generally written with the letter "**r**." A superscript is used to identify the two points connected by the vector. For example, $\vec{\mathbf{r}}^{A_0B_0}$ is a vector that

---

whose direction is $\vec{\vec{\mathbf{C}}}$ and whose scalar magnitude is the dot product $\vec{a} \cdot \vec{b}$. The same expression could also be written $\vec{a} \cdot (\vec{b}\,\vec{c})$, where the expression $(\vec{b}\,\vec{c})$ is a dyad. A dyad is a notational convenience that is useful for indicating quantities that are eventually projected (by the vector dot product operator) onto arbitrary directions.

[1] Vectors represented with Greek letters are not shown in boldface, due to a limitation of the Apple printer used to create this document.

goes from point A_0 to point B_0. When only one point is shown, the implicit first point is the origin of the inertial reference. For example, the absolute position of B_0 is given by the vector $\vec{\mathbf{r}}^{N_0B_0}$, written more simply as $\vec{\mathbf{r}}^{B_0}$.

Vectors representing direction are also written with the letter "**r**." The body associated with the direction is shown with a superscript, and a descriptive subscript defines the type of direction. For example, a rotation axis for body B is written $\vec{\mathbf{r}}^{B}_{rot}$.

Reference frames for derivatives are indicated with preceding superscripts. For example, the derivative of the vector $\vec{\mathbf{r}}^{B_0}$ with respect to the reference frame A is written $^{A}\frac{d\,\vec{\mathbf{r}}^{B_0}}{dt}$. When no preceding superscript is shown, the derivative is with respect to the inertial reference. Derivatives taken with respect to the inertial reference are also shown with an overhead dot for brevity. That is,

$$\dot{\vec{\mathbf{r}}}^{B_0} = \frac{d\,\vec{\mathbf{r}}^{B_0}}{dt} = {}^{N}\frac{d\,\vec{\mathbf{r}}^{B_0}}{dt} \tag{3.3.1}$$

Vectors representing velocity are generally written with the letter "**v**." A superscript is used to indicate the point whose velocity is represented. A leading superscript is used to indicate a reference frame other than the inertial one. For example, the velocity of B_0 in the reference frame of A is written $^{A}\vec{\mathbf{v}}^{B_0}$. If the velocity is relative to the inertial reference, the leading superscript is usually omitted. That is, the absolute velocity of point B_0 is written $\vec{\mathbf{v}}^{B_0}$, rather than $^{N}\vec{\mathbf{v}}^{B_0}$.

Acceleration is written with the letter "**a**" in a fashion similar to velocity. For example, $^{A}\vec{\mathbf{a}}^{B_0}$ is the acceleration of B_0 with respect to the reference frame of A, and $\vec{\mathbf{a}}^{B_0}$ is the absolute acceleration of the same point.

Incremental velocities and accelerations are defined as the difference in velocity or acceleration between two points. For example,

$$\vec{\mathbf{v}}^{A_0B_0} = \vec{\mathbf{v}}^{B_0} - \vec{\mathbf{v}}^{A_0} \tag{3.3.2}$$

and

$$\vec{\mathbf{a}}^{A_0B_0} = \vec{\mathbf{a}}^{B_0} - \vec{\mathbf{a}}^{A_0} \tag{3.3.3}$$

The name “incremental” is used instead of “relative,” because the terms “relative velocity” and “relative acceleration” are sometimes defined as velocities and accelerations relative to a specified reference frame. (The incremental velocity is the time derivative of the position vector connecting the two points, with respect to the inertial reference. Similarly, the incremental acceleration is the second derivative of the same position vector, with respect to the inertial referene.)

Angular velocity is written with the letter “ω.” Here, superscripts refer to bodies. For example, the angular velocity of B relative to A is written ${}^{A\vec{\to}B}\omega$, and the absolute angular velocity of B is written $\vec{\omega}^{B}$.

Angular acceleration is written with the letter “α.” Thus, the angular acceleration of body B is written $\vec{\alpha}^{B}$.

Incremental angular acceleration is written by putting the symbols for both bodies in the superscript on the right-hand side of the symbol, e.g., the incremental acceleration from A to B is

$$\vec{\alpha}^{AB} = \vec{\alpha}^{B} - \vec{\alpha}^{A} \tag{3.3.4}$$

The incremental angular velocity from A to B is also the relative velocity of B with respect to A. That is,

$$\vec{\omega}^{AB} = \vec{\omega}^{B} - \vec{\omega}^{A} = {}^{A\vec{\to}B}\omega \tag{3.3.5}$$

Because they are equivalent, incremental angular velocities are always written as relative velocities (that is, the incremental angular velocity between A and B is equal to the angular velocity of B relative to the reference frame of A). The same equivalence does not hold for angular acceleration, and therefore the notation of eq. 3.3.4 is used. (The incremental angular acceleration is the time derivative of ${}^{A\vec{\to}B}\omega$ with respect to the inertial reverence, not the reference frame of A.)

Table 3.3.1 provides a summary of the vector and dyadic notation that will be used extensively in Chapters 6 and 8.

Table 3.3.1. Notational conventions for vectors and dyadics.

Notation	Description
$\vec{\mathbf{a}}_i$, $\vec{\mathbf{b}}_i$, etc., i=1,2,3	unit-vectors for body (body A for $\vec{\mathbf{a}}_1$, $\vec{\mathbf{a}}_2$, $\vec{\mathbf{a}}_3$; body B for $\vec{\mathbf{b}}_1$, ..)
$\overleftrightarrow{\mathbf{a}}$, $\overleftrightarrow{\mathbf{b}}$, etc.	basis dyadic for associated body
$\vec{\mathbf{r}}^B_{dir}$	direction associated with joint between body and its parent. Superscript specifies body, subscript describes type of direction.
$\vec{\mathbf{r}}^P$	position vector from fixed origin to point in superscript.
$\vec{\mathbf{v}}^P$, $\vec{\mathbf{a}}^P$	absolute velocity or acceleration of point in superscript.
$\vec{\omega}^B$, $\vec{\alpha}^B$	absolute rotational velocity or acceleration of body in superscript.
$\overleftrightarrow{\mathbf{I}}^{B*}$	inertia dyadic of body about its mass center. (Superscript specifies center of mass.)
$\vec{\omega}^B_i$, $\vec{\mathbf{v}}^{B*}_i$	holonomic partial angular and central velocities for body B associated with speed u_i. (Body/mass center is shown in superscript, speed index is subscript.)
$\tilde{\vec{\omega}}^B_r$, $\tilde{\vec{\mathbf{v}}}^{B*}_r$	nonholonomic partial angular and central velocities for body B associated with speed u_r. (Body/mass center is shown in superscript, speed index is subscript; tilde indicates that partial velocity is nonholonomic.)
$\vec{\mathbf{a}}^{B*}_{rem}$, $\vec{\alpha}^B_{rem}$, $\tilde{\vec{\mathbf{a}}}^{B*}_{rem}$, $\tilde{\vec{\alpha}}^B_{rem}$	holonomic and nonholonomic acceleration remainders. Without a tilde, the symbol represents the portion of acceleration comprised of quadratic terms involving generalized speeds. With a tilde, it's the portion of acceleration not accounted for by derivatives of independent speeds.
$\vec{\mathbf{v}}^{A*B*}$, $\vec{\mathbf{v}}^{A*B*}_i$, $\tilde{\vec{\mathbf{v}}}^{A*B*}_r$, $\vec{\mathbf{a}}^{A*B*}_{rem}$, $\vec{\omega}^{AB}$, $\vec{\alpha}^{AB}_{rem}$, etc	incremental terms appearing in recursive relationships, e.g., $\tilde{\vec{\mathbf{v}}}^{B*}_r = \tilde{\vec{\mathbf{v}}}^{A*}_r + \tilde{\vec{\mathbf{v}}}^{A*B*}_r$; $\vec{\alpha}^B_{rem} = \vec{\alpha}^A_{rem} + \vec{\alpha}^{AB}_{rem}$.

Matrices and Arrays

Matrices and arrays are represented by underlined letters in plain typeface. Lower case letters are used to represent one-dimensional arrays, e.g., $\underline{f}$. Capital letters are used to represent two-dimensional matrices, e.g., $\underline{M}$. An overhead dot indicates that the derivative is taken of every element of the array. For example, the array of independent speeds is $\underline{u}$, and the array of the derivatives of the independent speeds is $\dot{\underline{u}}$.

A one-dimensional array is called a vector by many authors. To avoid confusion with the concept of a vector described earlier, the word "vector" is reserved here for unit-vectors and expressions involving products of scalars and unit-vectors.

As a matter of style in this dissertation, two-dimensional matrices are called matrices and one-dimensional matrices are called arrays. Also, it so happens that all two-dimensional matrices that appear in this dissertation are square (that is, the number of rows equals the number of columns).

In past work involving computer representations of vectors and dyadics, vectors are often represented as 3-element arrays and dyadics are represented as 3x3 matrices. As will be seen in Chapter 5, that representation is not used in this work. Matrices and arrays are used sparingly, particularly in comparison to other multibody formalisms.

Computer Data Objects

The multibody system is eventually represented as a set of symbolic computer data *objects* that are manipulated by the computer. An "object" is a set of data that can be handled as a single entity by the computer. The set might be a number, an alphanumeric character, a *string* (a string is a sequence of alphanumeric characters usually written in quotes, e.g., "this is a string"), an array, a *list* (a list is a sequence of objects, usually enclosed in parentheses), a subroutine, a *symbol* (a symbol is an object that associates a name and value), and others. In Chapter 5, many new types of objects are defined to represent algebraic expressions and elements of a multibody system. Names of computer objects are written in the Courier typeface, e.g., `symbol`. All symbols are represented internally in upper-case letters, but are often written in lower-case. For example, the same symbol can appear as `rotor`, `ROTOR`, `Rotor`, or `RoToR`.

Computer procedures are called subroutines in some languages; in Lisp, they are usually *functions* or *macros*. In the context of computer methods[1], the word "function" refers to a computer object that performs a sequence of operations, possibly involving data provided as *arguments*. For example, the Lisp object

[1] In a different context, the word "function" is used to indicate an expression that includes certain variables, e.g., "Y is a function of X" indicates that X appears in the expression that represents Y.

```
(add 2 3)
```

invokes the function `add` and provides values for two arguments (2 and 3). The object is *evaluated* by applying the function `add` to the values of the arguments, yielding the result of 5.

The names of formal arguments to computer functions are written in italics. For example, the function `add` works by adding the two arguments *arg1* and *arg2*. When the function is evaluated, the names shown in italics are replaced by the actual arguments.

Data objects introduced in Chapter 5 have *slots*, where each slot has a name and value. The slot names are always written in italics, e.g., the *terms* slot of a `sum` contains the terms of a summation.

Some of the functions have arguments that are are used to override default values. They are optional, and, if used, must be identified with *keywords*. All keywords are shown in the Courier font, and begin with the character ":" (without the quotes). Example keywords are `:body`, `:name`, and `:coordinate-system`.

Appendix A presents a short summary of Lisp syntax and provides a few more details about how the computer data objects are written. Table 3.3.2 summarizes the conventions just described.

Table 3.3.2. Conventions for computer data objects.

Convention	Description
`courier`	typeface used for (1) function and macro names, (2) types of data objects, and (3) names of Lisp symbols.
italics	typeface used for (1) formal arguments to Lisp functions, and (2) names of slots in Lisp structures.
[N1]	short names enclosed in square brackets and ending in the number 1, 2, or 3 are unit-vectors for the body associated with the short name (e.g., [N1] indicates the unit-vector $\vec{\mathbf{n}}_1$, associated with N).
`:keyword`	symbols shown in Courier typeface that begin with a colon are keywords used to specify optional arguments.

Parentheses, Braces, and Brackets

Parentheses and square brackets are used conventionally in equations and in text. However, additional meanings apply when computer objects are described or printed in Chapters 5, 8, and 9.

Parentheses are used to indicate Lisp forms and other `lists`. Expressions printed by AUTOSIM use parentheses according to Fortran conventions for (1) nesting expressions and (2) showing arguments of subroutines and functions.

Square brackets [] are used in expressions printed by AUTOSIM to indicate unit-vectors, e.g., [N2].

Curly brackets {} are used in descriptions of AUTOSIM functions and macros to indicate optional arguments.

Continental brackets « » are used to indicate that the enclosed expression is replaced with an intermediate variable and that the intermediate variable is used for subsequent appearances of the expression.

3.4 Topology

The *topology* of the multibody system is the description of how bodies are connected to each other. The connections can be thought of as introducing degrees of freedom for bodies that would otherwise be completely constrained. (Alternatively, they can be thought of as constraining bodies that are otherwise free to move in any manner.)

Degrees of Freedom

The number of degrees of freedom for a multibody system is the number of independent generalized speeds, p.

In this dissertation, it is useful to also consider the number of generalized coordinates associated with each rigid body in the system. For lack of a better name, this number is called the *number of degrees of freedom of the joint* connecting the body to another body. There can be up to three rotational degrees of freedom, and up to three translational degrees of freedom.

The total number of joint degrees of freedom for all of the bodies in a system equals the number of multibody coordinates. If a system has kinematical loops or nonholonomic constraints, the number of degrees of freedom for the entire system is less than the sum of the degrees of freedom of the body/joint pairs. Constraint equations account for the difference.

Trees

A tree is a type of graph constructed from entities called *nodes*. One node is the "root node" that starts the tree, and which has no "parent node." Every other node in the tree is defined as a "child" of a previously defined node. An example tree is shown in Figure 3.4.1, for 8 nodes labeled by capital letters. Parent-child relations are shown by lines, with the parent node above the child node(s). The root node is N; nodes A and B have N as their "parent." Thus, A and B are the "children" of N. B has three children. Nodes G, C, D, and E all have no children, and are called "leaves" of the tree.

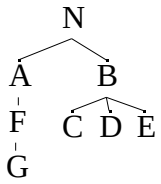

Figure 3.4.1. Example tree.

Many multibody systems are well suited for description by trees. The nodes of the tree are rigid bodies and the connecting lines are joints between the bodies. If the body has no physical connection to its parent (e.g., a free body whose parent is the inertial reference N), the joint simply imposes zero constraints. In the other extreme, a body rigidly attached to the parent involves a joint that imposes six constraints. For example, the tree in 3.4.1 could be used to represent the multibody system shown in Figure 3.4.2, with ovals used to designate rigid bodies. The root in the multibody tree is a fixed inertial reference, N.

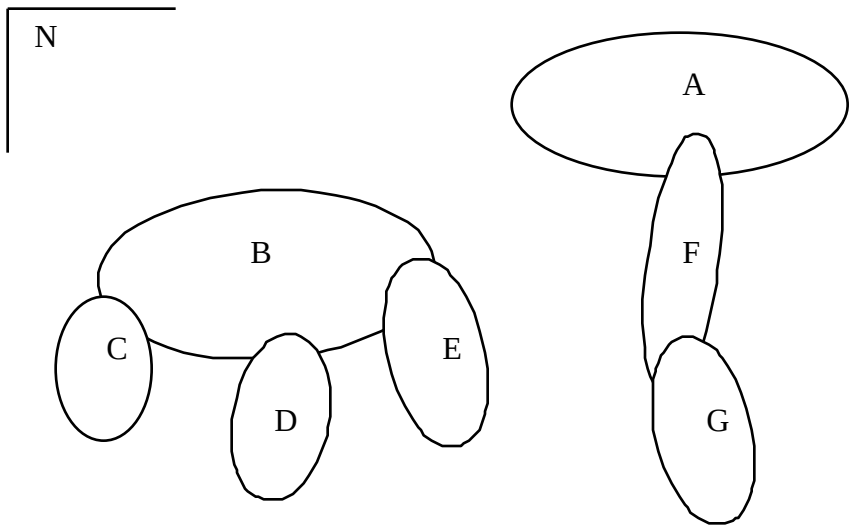

Figure 3.4.2. Rigid bodies in a tree topology.

A *tree-type multibody system* is one in which all holonomic constraints are accounted for in the tree. That is, as each body is added to the tree, a joint relating the body to its parent is also introduced. The number of degrees of freedom added with the new body is a number between 0 and 6, depending on the constraints imposed by the joint.

A tree-type multibody system is shown in Figure 3.4.3. There are two bodies, A and B, and ground, N. Body A has N as its parent, and body B has A as its parent. The motions of A and B are restricted due to the holonomic constraints imposed by the pin joints. Forces and moments are generated by the two pins as needed to constrain the motions, but the constraint forces and moments do no work and cannot actively move the bodies.

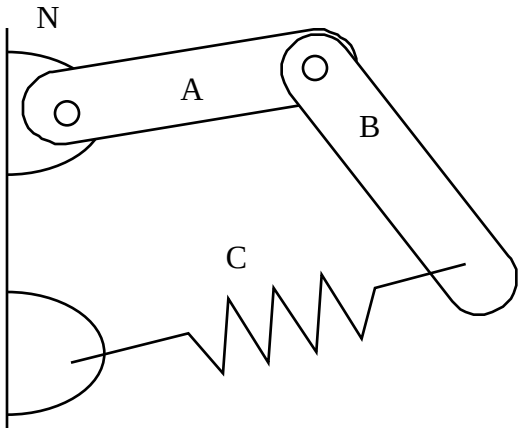

Figure 3.4.3. Two-link system.

A third element is also shown, namely, a massless spring C that connects B to N. The spring does not apply a kinematic constraint. Instead, it applies a force based on the force-deflection properties of the spring. The spring is a *force-producing* component. Under the convention developed here, the spring does not involve a joint and is not a member of the tree. With or without the spring, the system still has two degrees of freedom.

Additional Constraints

The tree representation does not directly accommodate systems with *kinematical loops* of the sort shown in Figure 3.4.4. In this system, the spring has been replaced with a rigid link, creating a four-bar-linkage with just one degree of freedom. It cannot be represented directly with a tree because there is a loop, in which the number of bodies equals the number of joints. That is, if we start with N and build a tree with N the parent of A, A the parent of B, and B the parent of C, we must stop without including the joint between C and N. C can have only one parent, and C cannot be the parent of N because N (as the root node of the tree) by definition has no parent.

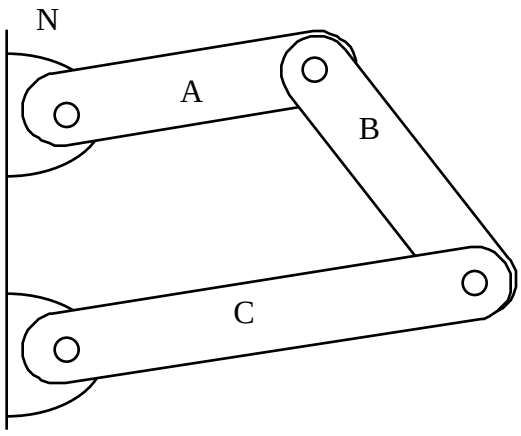

Figure 3.4.4. Four-bar linkage.

A multibody system with closed kinematical loops can be represented with a tree if it is augmented with additional information to account for the extra joint(s). The loop is closed by adding the additional joint(s) in the form of constraint equations. For example, the four-bar linkage is described by the tree shown in Figure 3.4.5, where the dotted arc indicates the joint between B and C. Without the arc, the system represented in the tree has three degrees of freedom, rather than one. The constraint relationship represented by the dotted arc must include two scalar equations that can be applied to reduce the degrees of freedom from three to one.

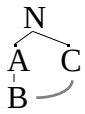

Figure 3.4.5. Tree for closed loop.

Many of the analysis methods that follow involve “traversing the tree.” With the tree drawn as shown (with the root at the top), traversing “up” the tree implies considering the parent of a body, then the parent of the parent, and so on until the root is reached. Traversing “down” the tree involves considering the children of a body, then the children of the children, and so on until all bodies that have the initial body “up” the tree are considered.

4. SPECIALIZED SIMULATION CODES

This chapter describes the general method used to simulate a multibody system when the simulation code is specific to a particular multibody system.

4.1. Overview

Figure 4.1.1 shows the flow chart for the simulation code. There are three basic tasks that the software must perform:

1. The data that distinguish this run from other runs are read as input. Inputs of this

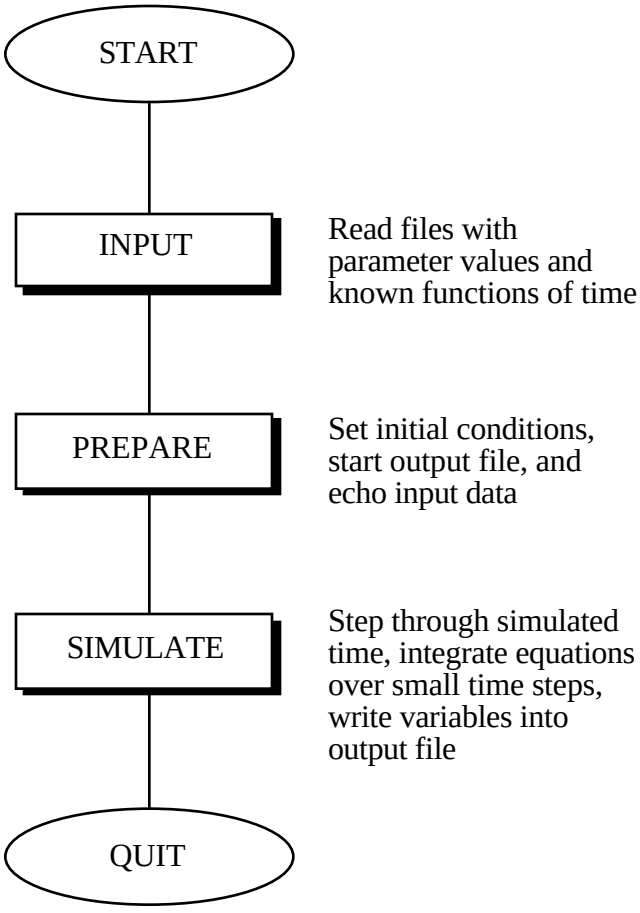

Figure 4.1.1. Overview of a simulation program.

sort include parameter values, initial conditions, and specifications for predetermined functions of time (i.e., forcing functions from controllers and disturbances). This activity is shown by the block labelled INPUT in the figure.

2. Computations are made to prepare the simulation by setting initial conditions and constants computed from parameter values. An output file is started, in which values of the output variables will be written. Also, the input values might be "echoed" by writing them into a file. These activities are identified by the block labeled PREPARE.

3. The simulation is performed. The equations of motion for the multibody system are used to compute values of state variables at discrete points in time. These values may also be written into one or more output files. This task involves operations that are repeated for each point in time, using a program loop. Thus, they are referred to as "in-the-loop" computations. These tasks are represented in the figure by the block labelled SIMULATE.

The validity and efficiency of the simulation is primarily determined by the computer code that performs the third task, SIMULATE, because it is executed many times as the simulation "steps" through time over small intervals.

4.2. Simulation Start-up Operations

If the simulation software is to be used productively, the input requirements should closely match the form of the data employed by the user to describe the system. The input files should be easily understood, and allow the user to make minor modifications easily for sensitivity studies. Also, the design of the input files should facilitate the building of libraries, in which subsystems are described in separate files that are combined to define the entire system. And finally, the simulation output should be written such that viewing of results requires minimal post-processing. To attain these goals, the portion of the code in the simulation code that performs the two tasks labelled INPUT and PREPARE is often complex, and can account for more of the computer code than the portion that performs the simulation.

Input

All parameters of the multibody system that are used in formulating kinematical and dynamical equations are generally programmed in the program as variables, so that they can

be changed with each simulation run. New values for the parameters are read from one or more input files, and then "echoed" in one or more output files to allow the user to confirm that parameters were interpreted properly by the simulation code. The design of the interface between the simulation code and the user has a significant practical affect on how easy the code is to use. This topic is not a part of this dissertation, and therefore details of how a simulation program reads and verifies new parameter values will not be explored. However, complete simulation codes are included in Appendices B and C, and the interested reader can review the methods used to provide a reasonably user-friendly file format.

It is desirable that the input parameters for the simulation codes be familiar to typical users. This has an implication when developing methods for automatically formulating simulation codes. Generating simulation codes with the "correct" parameter definitions can only be accommodated if the analyst is free to describe the system using symbols that are already familiar. If the parameters commonly used do not correspond to constants appearing in the equations of motion, the simulation code should compute the required constants from the parameters defined by the analyst.

Prepare

Before starting the iterative "in-the-loop" simulation computations, a number of computations are performed involving the data read from the input files. Terms that involve constants are "precomputed" to reduce the number of arithmetic operations required in the loop. The output file is created, and the labels associated with the output variables are written. For a constrained system, the values of dependent variables may need to be computed to start the simulation with a realizable state. For some kinds of user-supplied subroutines, variables used by those subroutines must be given starting values. Scale factors are applied as necessary to convert parameter and initial condition variables from "user-convenient units" (lbm, deg, etc) to "equation-convenient units" (in-lb/sec^2, rad, etc.). (See the subroutine INPUT in Appendices B and C for example code that performs such conversions.)

4.3. "In-The-Loop" Computations

The actual simulation is performed by numerically solving the equations of motion of the mechanical system over and over, in a loop. Figure 4.3.1 breaks down this part of the simulation software into three operations. These are performed many times, as multiples of

a basic “time step,” selected on the basis of the frequency response of the system and the use made of the simulation. Figure 4.3.2 shows approximately the frequencies in which these operations are performed. Typically, the evaluation of the derivatives via DIFEQN is performed the most frequently, while the OUTPUT operation is performed the least.

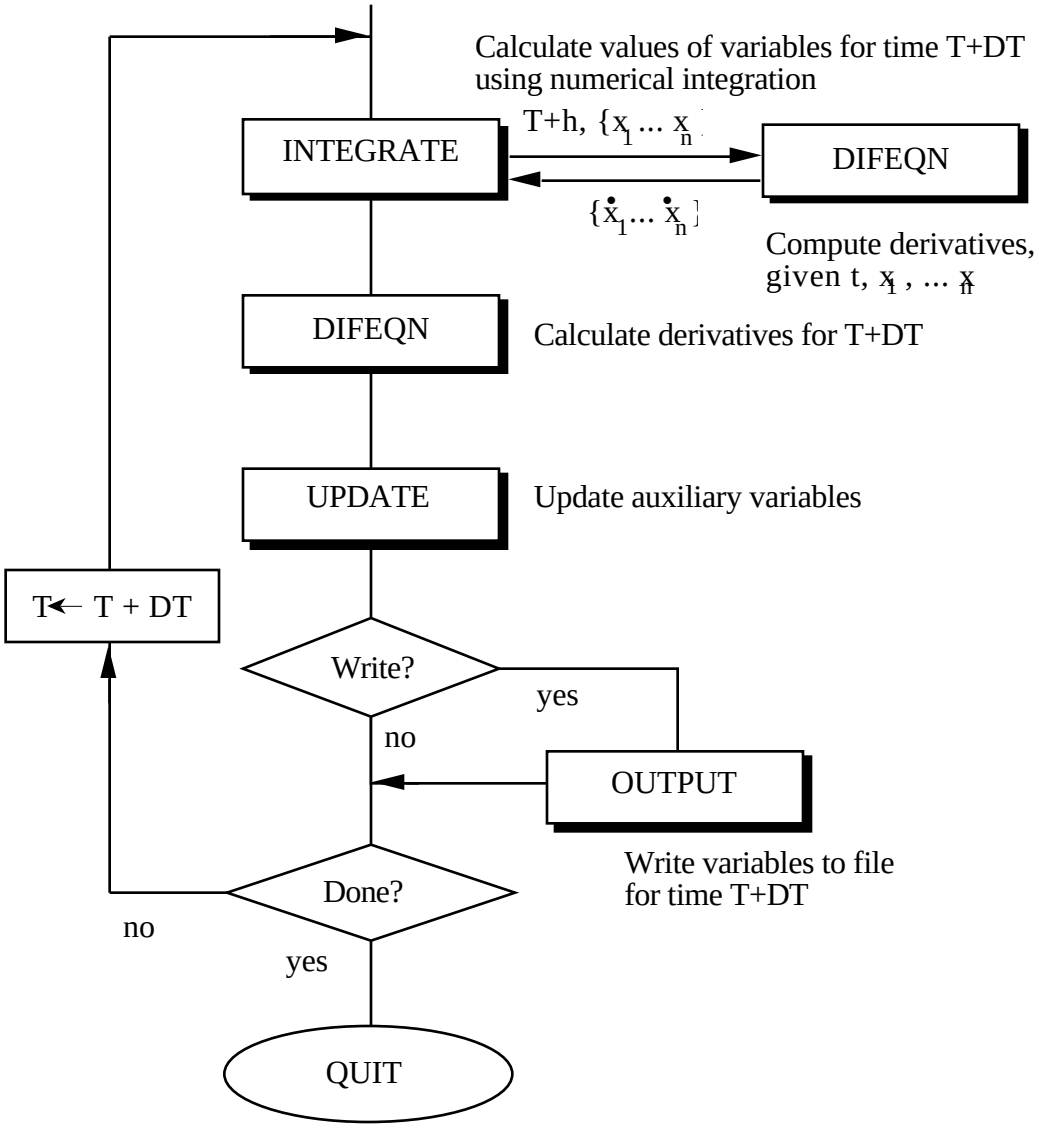

Figure 4.3.1. Block diagram for “In-the-loop” computations.

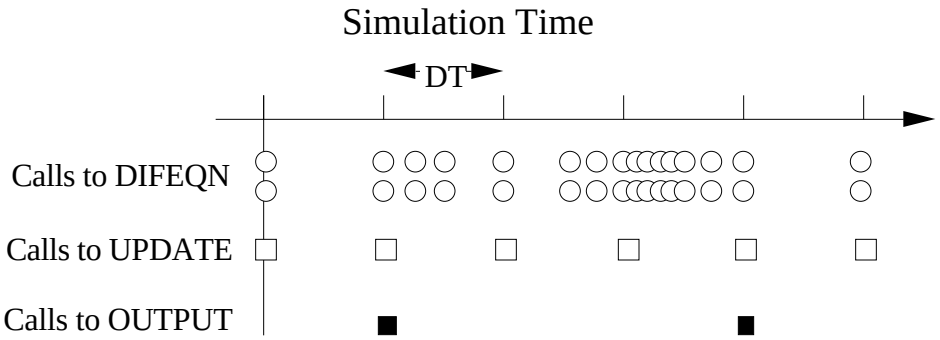

Figure 4.3.2. Example frequency of "in-the-loop" tasks.

Integrate

The equations of motion for a multibody system are ordinary single-order differential equations that are linear with respect to the derivatives but nonlinear with respect to other variables. A subroutine DIFEQN contains the equations of motion for the multibody system in a form suitable for computer solution. That is, it computes as output the derivatives of the state variables for time T, given as input the values of state variables and the value of T.

The derivatives provided by DIFEQN are used by a numerical integration algorithm (shown in the figure as the subroutine INTEGRATE) to compute values of the state variables at different times T+h, where h usually ranges between 0 and DT, and DT is a time step that is "small" with respect to the highest frequency associated with the response of the multibody system.

The subroutine DIFEQN is called after INTEGRATE, so that the accelerations for time T+DT are available when the UPDATE and OUTPUT operations are performed. For maximum efficiency, the INTEGRATE procedure can use these values also, and thus should not call DIFEQN to obtain derivatives for the new time T.

Hundreds of numerical integration algorithms exist (Euler, Runge-Kutta, Predictor-Corrector, Gear, etc.). The choice of which to use depends on the characteristics of the system being simulated, the uses to be made of the results, and personal preferences of the engineers developing the software [29, 37, 41, 93, 119]. Overall, the simulation algorithm is but a small part of the simulation code which is easily changed as necessary. The numerical integration algorithm can be an essential part of the computer representation of the multibody system, by forming the equations of motion to tie in directly with the integration equations [42, 81, 90, 122, 131].

The automated selection of an integrator algorithm is a worthy research topic, but it is not covered in this dissertation other than to emphasize how important the code in the subroutine DIFEQN is with respect to the computational efficiency of the simulation code as a whole. (There will be a mix of analytical and numerical methods used in the equations of motion as developed in Chapter 8, but the numerical methods are not directly related to the integration algorithm,)

Nearly all numerical integration methods work by invoking a function such as DIFEQN one or more times per time step. The plot of frequency shown in Figure 4.3.2 is representative when a variable-step integrator is used. It shows the frequency of calls to DIFEQN varying between time steps as needed to obtain a required numerical accuracy. Also, it shows the routine being called at least twice for each value of T at which it is invoked.

Update

There may be external subroutines which read values from files, or which "remember" histories of system variables to compute forces and moments. Once per time step these activities are performed.

Periodic updating is also necessary for real-time simulation, with hardware in the loop. Once per time step, a subroutine is invoked that communicates with hardware, passing computed values to the digital/analog converter (D/A), and grabbing new values of input variables from the analog/digital converter (A/D).

The UPDATE operation is used only when such subroutines are incorporated into the simulation code.

Output

Values of selected variables are written into one or more files for viewing. These can include motion variables, forces and moments, and variables derived from state variables. The sample frequency for the output file is a multiple of the time step. Depending on which variables are of interest, and whether the post-processing software has its own sampling requirements, the multiple can be as small as 1, or as large as several hundred. Typically, it is between 2 and 20. Because the OUTPUT routine is invoked less frequently than DIFEQN, output variables that are derived quantities should be computed in this routine, rather than in DIFEQN.

5. SYMBOLIC COMPUTATION METHODS

This chapter develops a symbolic mathematics language tailored specifically for analyzing multibody systems and generating numerical simulation codes. The language directly represents three aspects of the overall system in symbolic form:

1. vector and dyadic algebra expressions,
2. components of the multibody system (bodies, forces, etc.), and
3. pieces of computer code that go into the numerical simulation code being generated.

In the remainder of this chapter, techniques are presented for representing and manipulating these components as computer data objects, with emphasis on eventually generating numerically efficient source code in a target language (e.g., Fortran).

The symbol manipulation language described in this chapter (called AUTOSIM) is written in Lisp, or more specifically, the language "Common Lisp" [4, 118] (called simply "Lisp" throughout this dissertation). The Lisp language has long been associated with symbolic manipulation languages, and with prototyping other languages. The programs MACSYMA, REDUCE, and MuMath were developed in various versions of Lisp [92, 94, 139]. Rudimentary computer algebra systems even appear in some introductory Lisp and computer science textbooks as case studies, e.g., [7, 60].

There is another aspect of Lisp that is convenient for the intended work. Lisp systems typically are provided with an interactive environment that allows the Lisp programmer to interact easily with the computer. Basic functions such as opening, printing, editing, and creating files are supported. Further, Lisp "forms" (a Lisp form is similar to a command in other languages) can be entered and evaluated interactively. This environment is ideally suited to the needs of a dynamicist analyzing a multibody system. (The advantages of Lisp systems for general engineering analysis (i.e., not computer science) have also been noted [8, 34].) As will be seen in the examples of Chapter 9, the analyst is free to view intermediate results as the analysis proceeds, and to inspect various expressions in great detail. Also, the symbol manipulation capabilities can be used interactively to derive expressions of interest to the analysts which may or may not eventually appear in the

equations of motion for the system. Because Lisp is a well documentated language, it is unecessary to invent a new syntax for AUTOSIM. That is, AUTOSIM is implemented simply as an extension to the existing language.

Although Lisp has existed for almost as long as Fortran, it has mainly been used on mainframe computers and specialized (i.e., very expensive) workstations until the past several years. The full Common Lisp language is now available from a variety of companies for machines ranging from IBM and Apple desktop computers up to Cray supercomputers. (The computer work described in this dissertation was all performed on Apple Macintosh computers using the Allegro Common Lisp compiler [6].)

5.1. Considerations of Numerical Efficiency

Choices made by an analyst deriving equations of motion for a multibody system have a direct impact on the complexity of the resulting equations. Some of the techniques that are typically employed to simplify equations are the following:

1. State variables are introduced that are "natural" to the system being analyzed (joint displacements, speeds oriented in body-based directions, Euler angles, etc.), avoiding transformations to a predefined choice (e.g., Cartesian global coordinates) [35].

2. Terms which are known to be zero for the specific system (but which could be non-zero for a more general formulation) are omitted from the equations.

3. Forces and moments that cancel due to symmetry or because they involve no work are eliminated when possible [9, 46, 54, 55, 57, 58, 126].[1]

4. Equations are written in "factored form," involving products and ratios of sums of terms. For example, computing a value for the expression $(A + B + C)^2$ requires two additions and one integer power. In contrast, the expanded form ($A^2 + 2AB +$

[1] The effectiveness of this technique is controversial, as a trade-off is made between a small number of complicated equations and a large number of simple equations. The question of whether large sets of simple equations are better or worse than small sets of complicated equations has not been resolved, and is a topic of current research. When the objective is solely to simulate motions due to forces and moments, forces and moments of constraint are of no interest and additional computations made to determine them slow down the simulation. However, when the objective is to obtain the forces and moments of constraint as a means to evaluate alternate designs, formulations that compute the constraint forces and moments in addition to the motions are necessary.

$B^2 + 2AC + 2BC + C^2$) requires five additions, six multiplications, and three integer powers.

5. Terms involving products or powers of quantities known to be "small" are dropped if they are of order 2 or higher. In many mechanical systems, some of the motions are limited such that variables associated with those motions are much smaller than other expressions arising in the equations of motion.

6. Trigonometric functions of small quantities are replaced with truncated Taylor expansions.

Technique no. 2 (removing zero terms) can only be partially implemented for generalized numerical multibody simulation methods (via the use of sparse matrix operations). However, virtually all symbolic multibody programs employ it. Techniques 1 through 4 have been used by some programs, and techniques 5 and 6 have not been used in a generalized sense until the implementation described in this dissertation. (In past work, "small" variables, when used, are built into the multibody formalism. The analyst could not utilize knowledge that some variables and parameters were small and that others were not.)

A given set of equations can be programmed into a simulation code so as to minimize computation by using the following techniques:

7. 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. Some of the redundancy is eliminated by using a recursive dynamics analysis method. Even so, inspection of the equations of motion usually reveals that some subexpressions appear more than once. A human programmer, concerned with numerical efficiency, will avoid performing the same computation more than once by saving the results the first time and then using the result when the same computation is called for again.

8. Computations that do not have to be performed in the DIFEQN part of the program are performed elsewhere. Constant expressions are "precomputed" in the PREPARE portion of the simulation code to avoid performing identical computations more than once. Computations involving output variables (units conversions, direction transformations, etc.) are performed in the OUTPUT part of the program, which is executed less frequently than the DIFEQN part.

9. A human programmer will (hopefully) not introduce code that serves no purpose. This obvious technique can be difficult to implement in an automated analysis method. For example, details of the dynamics analysis are often recursive. Hence, it is convenient at times to introduce expressions knowing that they will be referenced in a later stage of the recursion. However, if the recursion stops, they may not be needed. Or, an expression might be developed which is later multiplied by zero. Determining whether a particular expression will be needed later can be very difficult at the time the expression is formulated, although it is trivial to do after all equations are formulated.

   Once a simulation code is working correctly, a programmer concerned with computational efficiency can look over the code for sections that can be eliminated.

10. Large matrices are partitioned into smaller matrices, based on the topology of the system, before general numeric matrix solution methods are invoked. For example, it is much less work to solve three sets of six simultaneous equations than to solve one set of 18 simultaneous equations, because as n (the number of equations) increases, the "cost" increases approximately in proportion to n^3.

11. There is a certain amount of overhead in computer codes that do not explicitly perform arithmetic, due to code generated by the compiler to support "higher level" concepts in a language such as Fortran. Some examples:

    - counters must be created and updated to perform DO LOOPS.
    - computations must be made to determine the locations in memory of array elements with variable indices.
    - saving intermediate results in variables requires moving data from high-speed working registers and CPU caches into predefined memory locations.

    Thus, codes can run faster if DO LOOPS, variable indices in arrays, and intermediate variables are used sparingly.

Techniques 7 and 11 have been applied by some, but not all, symbolic analysis programs in the past [83, 101]. Techniques 8 and 9 have not been automated before as part of a multibody analysis program. Technique 10 will be employed in the most efficient manner possible, by obtaining a recursive symbolic solution for matrix equations in which no "wasted" arithmetic operations are included.

5.2. 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. In past work, expressions have generally been represented with a list structure, where the first element of the list provides the type of data (sum, product, function, etc.) and the remaining items represent other expressions. In the NEWEUL and SYMBA codes, such lists are used to represent most expressions implicitly as sums [83, 107]. In the more general MAPLE language, a larger number of data types are accomodated with a basic list structure [20]. An alternative to using existing computer data structures such as lists or arrays is to define new types of data objects for the computer that correspond exactly to the entities that they represent. This approach is a part of the "object oriented programming" style, and is used in AUTOSIM.

Overview of Data Objects

Lisp includes over 40 types of data objects. In addition, new types are included by the use of *structures*. Internally, the structure contains a number of *slots* which are essentially variables defined locally within the structure. Each slot has a name and can be assigned a value. In AUTOSIM, structures are used as *objects* to support object-oriented programming.[1] Objects facilitate data abstraction by allowing programs to manipulate the objects, without requiring the programmer to know about details of their internal representation. Further, "generic functions" work by obtaining procedures for manipulating objects based on the types of the objects. For example, the generic function `dxdt` (used to take the absolute derivative of an expression) works by looking at the type of the argument, and looking up that type in a dispatch table of "installed" specialized functions. The specialized function from the table is then invoked. To define the derivative of a new type of expression (e.g., a user-defined function), a new specialized function is written and "installed" in the system. (The installation is simply an updating of the function dispatch table.) However, the original `dxdt` function is not modified. Thus, the object-oriented style of programming allows new types of objects and new operations to be incorporated into the system without modifying existing software.

---

[1] Extensive object-oriented versions of Lisp are readily available, but are not standardized. To ensure portability, AUTOSIM is written completely in standard Common Lisp. The object-oriented extensions are a part of AUTOSIM.

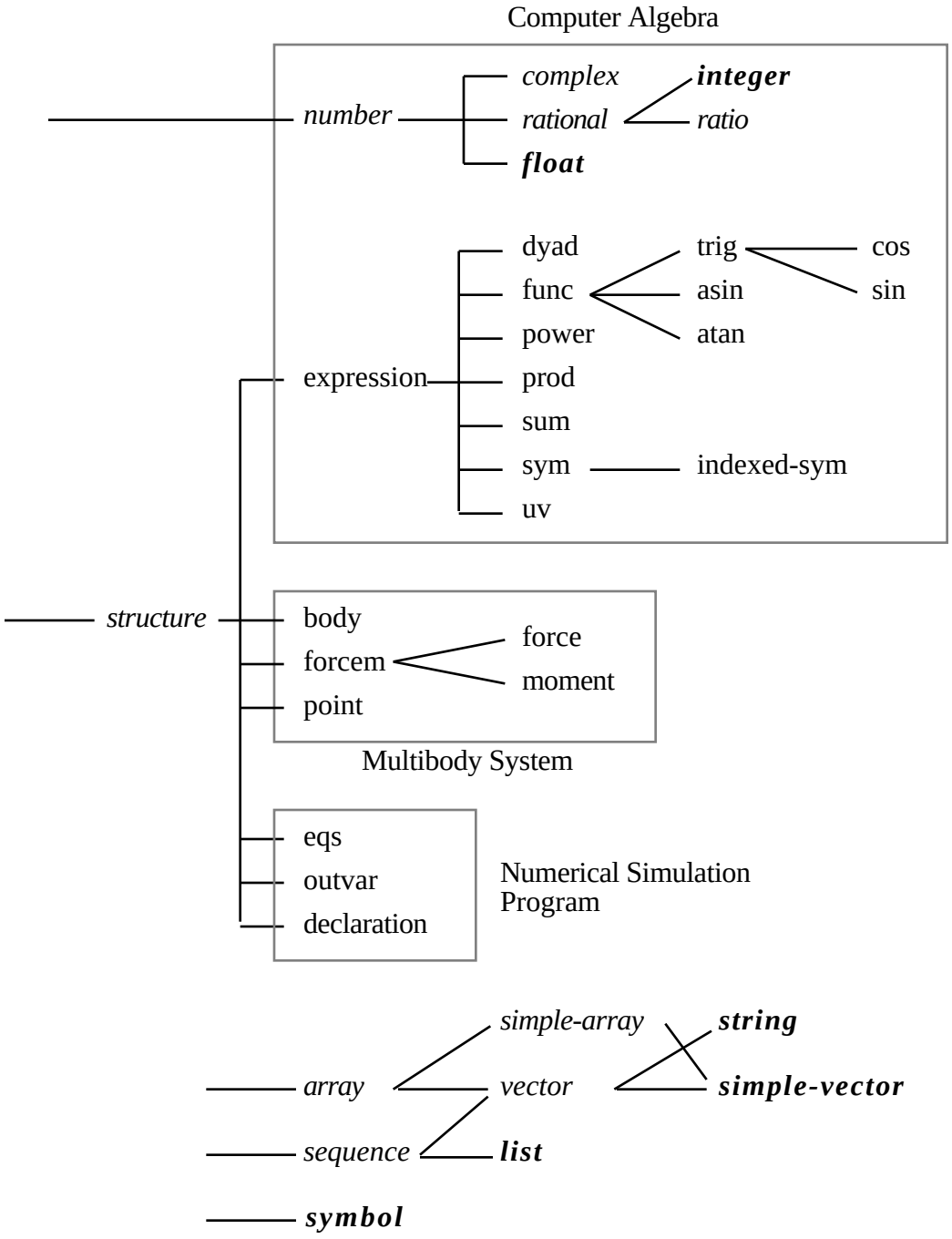

Figure 5.2.1. Hierarchy of AUTOSIM and Lisp data objects.

Figure 5.2.1 shows a hierarchy of data types used in AUTOSIM, as they relate to data types already in Lisp. Each type of 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 many functions that work with objects of type `trig`, `func`, and `expression` also work 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. All native Lisp forms are shown in italics, and those used extensively in AUTOSIM are shown in bold italics. The multibody analyses and simplification techniques are applied by manipulating these objects.

Computer Algebra

Expressions in AUTOSIM can represent scalars, vectors, or dyadics. They are composed of `number`s and `expression`s, whose characteristics are listed in Table 5.2.1. Of the expressions defined in the table, four are *elementary* types from which the other *compound* types are built. The elementary types are the `number`, the `sym`, the `indexed-sym`, and the `uv`. When printed as Fortran source code, the `sym` designates a variable and an `indexed-sym` usually designates an array element. Unit-vectors are never written in the final Fortran output, but can be entered and read by the analyst. (They are printed with enclosing square brackets.)

In the next chapter, we will see that most of the quantities appearing in the dynamics equations are vectors and dyadics. Virtually all previously developed automated multibody analysis methods formally define directions ahead of time, so that vectors can be described using three-element arrays of scalar quantities with predefined directions. This approach works fine for many kinds of rigid-body analyses, because expressions can be formulated in terms of unit-vectors fixed in the body with which they are associated. However, simpler equations are sometimes obtained by writing expressions for velocity or acceleration vectors using unit-vectors fixed in a different body. Also, an inflexible approach becomes cumbersome when dealing with forces and moments between bodies, because forces and moments are often defined in orientations that defy conventions of any single multibody formalism. Introducing arbitrary forces has not been been possible with symbolic analysis programs in the past, limiting the level of automation that is possible in the modeling.

Table 5.2.1. Summary of AUTOSIM expression types.

Type	Primary Slots	Definition	Examples
`number`		number	2, 1/3, –.3333

`expression`	*type, small-order, sort-code, dxdt, sym-value, const-or-var, units, name*	meta-type for all expression objects	
`sym`	*symbol, default, hide, exp*	symbol for a scalar parameter or variable	M
`indexed-sym`	*i, category*	indexed symbol for a scalar parameter or variable	Q(2)
`uv`	*symbol, body, dot-products, cross-products*	unit-vector	[A1]
`dyad`	*uv1, uv2*	dyad	([A1] . [A2])
`power`	*base, exponent*	base expression raised to power	U(1)**2
`prod`	*coef, factors*	product of numerical coefficient and list of expressions	2.0*M*SIN(Q(1))
`sum`	*terms*	sum of expressions	I + M*L**2
`func`	*function, args*	function that will be written into numerical program	TIRE(FZ, SLIP)
`trig`	*symbol*	sin or cos	
`cos`		cos	COS(Q(2))
`sin`		sin	SIN(Q(2))
`asin`		arc-sine	ASIN(X)
`atan`		arc-tangent	ATAN2(X, Y)

Limits involving the choice of unit-vectors used in vector expressions are averted by including unit-vectors as a primitive entity in the computer algebra representation. Vector and dyadic expressions can be introduced using simple mathematics notation, and then manipulated automatically. Also, vector velocities and accelerations can be projected in any

direction (via the dot-product operation) to define scalar output variables or scalar constraint equations.

Nested expressions (simplification technique no. 4 from Section 5.1) are supported in the designs of the compound expression types. For example, the expressions in the list of factors of a `prod` can be `sum`s, `power`s, `func`s, etc. There are no limits to the level of nesting allowed (other than computer memory).

The meta-type `expression` defines a repertoire of qualities associated with all expression types. For example, the units of any expression (if known) are kept in the *units* slot; the name of the expression (if there is one) is kept in the *name* slot; the derivative with respect to time, if known, is kept in the slot *dxdt*.

Expressions are classified in several ways besides their object type. The *type* slot tells whether an expression is a `scalar`, `vector`, or `dyadic`. `Power`s, `sym`s, and `indexed-sym`s always have their *type* slot set to the value `scalar`. Also, all numbers are by definition `scalar`. A `uv` has its slot set to `vector`, and a `dyad` is set to `dyadic`. The `prod` and `sum` objects can be any one of the three types, depending on the types of their components.

The *const-or-var* slot tells whether an expression is a constant or a variable. It is mainly used for scalar expressions, to identify expressions that can be precomputed. The value of this slot is set for a `sym` or an `indexed-sym` when it is created. When compound expressions are examined, the *const-or-var* slot is set to `const` if all expressions contained in the compound object are constants; otherwise it is set to `var`.

Some of the other slots are described later, in the context of the algebraic operations used on expressions.

Multibody System

A multibody system is composed of bodies influenced by forces and moments and connected to each other by joints. Points are fixed geometric locations in bodies used to define joint attachments, force attachments, and points of interest needed to define output variables or constraint equations.

`body` — A data structure called a `body` is used to represent each body in the system. A `body` contains about 30 slots that are used to access information about (1) the kinematics of a joint associated with the body, (2) properties of the rigid body, (3) the position and

orientation of the coordinate system fixed in the body, and (4) expressions that arise in the dynamics analysis applied to the multibody system. Because the vector dot-product and cross-product operations involve transformations between coordinate systems, some of the information in a `body` will be used to perform those operations. A few slots in a `body` that support algebra functions are shown in Table 5.2.2. Many more slots exist and are described in Chapter 8.

Table 5.2.2. Some of the slots in a `body` that support algebra functions.

Slot Name	Type	Definition
symbol	`symbol`	`symbol` for user to reference the body.
name	`string`	descriptive name written into output files produced by a simulation code.
parent	`body`	parent body in tree topology.
children	`list`	list of bodies that have this body as their parent.
uvs	`array`	3 unit-vectors that define 1-2-3 axis directions of coordinate system.
cos-matrix	`array`	direction cosine matrix relating the unit-vectors of this body to those of its parent.
level	`number`	level of the body in tree.
0-point	`point`	origin of coordinate system of this body.
joint-point	`point`	point in parent body that coincides with the *0-point* when all generalized coordinates are zero.
abs-w	`expression`	absolute rotational velocity of this body.
abs-v0	`expression`	absolute velocity of the *0-point.*

Massless bodies can be used to introduce compound joints or intermediate reference frames. Also, bodies with zero degrees of freedom can be used to add (or subtract) mass or inertia to an existing body.

Because each body (except the body used as the inertial reference) is explicitly a child of another body in the system, this design for the body organizes the multibody system into a tree topology. (The tree topology is described by the *parent* and *children* slots.)

Methods used previously to represent multibody systems have involved arrays that indicate relationships between bodies. As a minimum, a *body-connection matrix* is needed to indicate which bodies are connected by joints [46, 53]. Other matrices are needed to

indicate parent-child relationships and applications of constraint equations [32, 51, 75, 126]. The representation presented here is much simpler and permits reconstruction of the entire tree starting from any body in the tree, using only `body` objects. It also facilitates analyses that require that the bodies be processed in a certain sequence. For example, lisp code is shown below to apply a function `func` to each body in an order such that the parent is always processed before the child.

```
;;; apply function func to each body from the root down

(defun apply-func-to-tree-top-down (func body)
  (funcall func body)
  (dolist (b (body-children body))
     (apply-func-to-tree-top-down func b)))
```

The order of processing occurs from parent to child because the function is first applied to the body, and then the `apply-func-to-tree-top-down` function is recursively applied to the children of the body. By reversing two operations in the above function, so that the recursion occurs before the body is processed, the children are always processed first:

```
;;; apply function func to each body from the leaves up

(defun apply-func-to-tree-bottom-up (func body)
  (dolist (b (body-children body))
     (apply-func-to-tree-bottom-up func b))
  (funcall func body))
```

When the motion of a body relative to its parent is constrained due to the connecting joint, the vector expressions developed for the body motions can be defined recursively, based on the motions of the parent and the relative motion between the body and its parent. The example function `apply-func-to-tree-top-down` is representative of the functions employed in AUTOSIM to apply recursive formulations developed in Chapter 8.

Additional information is needed to fully describe multibody systems with nonholonomic constraints, or systems with holonomic constraints that define closed kinematical loops. The additional constraint information associates two bodies that are not already linked by a parent-child relationship. This information is not kept with either body. As will be seen later, constraint equations are included by modifying the `indexed-sym` objects used to represent state variables.

`point` — Points are used to define locations of interest in bodies, such as origins of the coordinate systems, mass centers, attachment points, etc. Each `body` contains at least four `point`s. (In addition to the two points listed in Table 5.2.2, a third point is introduced at

the mass center of the body as defined by the analyst, and a fourth point is introduced at the mass center as defined in the multibody formalism.) Additional `points` can be defined as needed to identify attachment points for forces or as points of interest for output variables and constraint equations. Table 5.2.3 shows how a `point` is defined in the system.

Table 5.2.3. Some of the slots in a `point`.

Slot Name	Definition
symbol	Symbolic name (`symbol`) for user to identify the `point`.
name	descriptive name (`string`) of the `point`.
body	`body` that contains the `point`.
coordinates	`array` of 3 coordinates of the `point` in the coordinate system of the `body`.

`forcem` — 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 5.2.4.

Table 5.2.4. Some of the slots in a `forcem`.

Slot Name	Definition
symbol	Symbolic name (`symbol`) for user to identify `forcem`.
name	descriptive name (`string`) of `forcem`.
direction	vector expression that gives direction of `forcem`.
magnitude	scalar expression that gives magnitude of `forcem`.
body1	first `body` on which `forcem` acts.
body2	second `body` from which `forcem` acts.
point1	`point` on line of action of force on `body` 1 (`force` only).
point2	`point` on line of action of force on `body` 2 (`force` only).

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

$$\vec{\mathbf{T}} = \vec{\mathbf{r}}^{B*P} \times \vec{\mathbf{f}} \qquad (5.2.1)$$

where $\vec{\mathbf{r}}^{B^*P}$ is the position vector going from the center of mass, B*, to the point P on the body through which the force passes, and $\vec{\mathbf{f}}$ 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. Three that are the most significant are the types `eqs`, `outvar`, and `declaration`.

`eqs` — 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.

`outvar` — 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 `outvar`s is processed to ensure that statements are generated to compute all dependent variables defined by the analyst. The labeling information is written by the simulation in such a way that output files can be handled automatically by post-processing software for graphics and analysis.

`declaration` — 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.

In its present form, all output source code is written in the Fortran language. However, the representation of the simulation program in `eqs`, `outvar`, and `declaration` objects is not dependent on the language. Generating simulation code in a different language (e.g., C) is mainly a matter of augmenting the print functions for each type of object, so that they are printed according to the syntax of the target language.

5.3. Computer Algebra Operations

The mathematical operations needed to derive equations of motion for a multibody system and generate source code for a numerical simulation program can organized into five categories for the purpose of implementing the operations in software: (1) operations

are implicitly performed when a compound expression object is created (e.g., a `power` object represents an expression raised to a power, a `prod` object represents the multiplication of expressions, etc.), (2) several primitive algebra operations are defined that use information obtained from slots in the expression objects to create a new expression object and assign values to some of its slots, (3) operations are defined to easily obtain variables associated with rigid bodies and their coordinate systems (e.g., angular rotation of a body), (4) higher-level algebra operations are defined in terms of primitive operations, and (5) some operations are performed on computer code that has already been generated. This last category of operations is analogous to a human programmer "looking over" the code he or she has written, to possibly make improvements.

Making Expression Objects

Each definition of a compound expression object implies an operation. The functions that make objects check their arguments and create simpler objects when possible. In fact, significant algebraic simplifications are performed in these operations. Table 5.3.1 summarizes simplifications that are performed by creator functions.

Most of the "small" quantity simplifications occur in the `make-sum` operation. 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 modified 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 [120].

Table 5.3.1. Simplifications performed by creator functions.

Function	Simplifications

`make-asin`
`make-cos`
`make-sin`
	• if argument is the inverse function, return argument of argument (e.g. $\sin(\sin^{-1}x) \rightarrow x$).
	• if argument is a number, evaluate.
	• if argument is small, return truncated Taylor expansion.

`make-atan`	• same simplifications as for `make-asin.` • if there are two arguments, divide both by GCF. [e.g., $\tan^{-1}(A*X, A*Y) \rightarrow$ ATAN2(X,Y)]

`make-power`	• if base is a `power`, change exponent. • if base is number, evaluate. • if base includes small terms, drop if possible.
`make-prod`	• 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 `prod`s, multiply coefficients and combine lists of factors (i.e., expand nested `prod`s). •• if any factors can be combined into a `power`, make the substitution. • else, sort factors and create `prod` object.
`make-sum`	•• compare "small-order" values of terms and remove those which are negligible. •• check for trig identities: $\sin^2 x + \cos^2 x \rightarrow 1$; $1 - \sin^2 x \rightarrow \cos^2 x$; $1 - \cos^2 x \rightarrow \sin^2 x$. •• if any terms are `sum`s, remove them and append terms from nested `sum`s to existing list (i.e., expand nested `sum`s). •• 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.
Note:	simplifications marked with •• mean that after the simplification is performed, the `make-` operation is called again recursively using updated arguments.

The other places that "small" simplifications occur are in the trigonometric functions. Truncated Taylor series are used to create expressions for these functions when the arguments are small. Otherwise, the appropriate `trig` or `func` object is made and returned.

Care has been taken to ensure that equivalent occurrences of a compound expression always are created the same way. `Sum`s nested within `sum`s and `prod`s within `prod`s are removed. For example, the sum (A + B) + C yields (A + B + C), rather than ((A + B) + C). Terms and factors are sorted in the `make-prod` and `make-sum` functions. For example, the product of B and A*C is A*B*C rather than B*A*C. A sign convention for `sum`s is used that results in a repeatable formulation for a given sum, regardless of how it

is obtained. For example, the expression (–A – B – C) would never be generated: instead, that result is always represented as –(A + B + C).

Primitive Algebra Operations

Table 5.3.2 summarizes the primitive mathematical operations. These operations involve one or two arguments. In the object-oriented environment, each operator has an associated dispatch table which is used to find a function for dealing with a specific type of expression (for unary functions) or combination of types (for binary operations). For example, to add a `sum` and a `prod`, the appropriate table is searched for the combination `(sum prc` new types of expression objects and new functions are "installed" in the system without modifying any of the existing software.

Most of the operators in the table work as might be expected. Exceptions and special notes are provided below.

`mul` — When developing expressions through multiplication, further simplifications are attempted. That is, numbers are multiplied on the spot, multiple appearances of an expression are combined into a `power`, multiple `power`s with the same base expression are combined, etc. Products are usually not expanded, in order to keep factored forms. However, there are times that expanded forms are preferred. For example, when solving for a symbol in an expression, it is necessary to subtract two potentially complex expressions such that the result contains no reference to the symbol being solved for. The expressions are expanded to ensure that the symbol is not buried in a subexpression, such that complete cancellation takes place.

Table 5.3.2. Summary of primitive AUTOSIM mathematics operations.

Operation	Argument(s)	Description
`add`	*x, y*	$x + y$
`const-or-var`	*x*	is *x* constant or variable?
`cross`	*v1, v2*	$\vec{\mathbf{v}}_1 \times \vec{\mathbf{v}}_2$
`dot`	*v1, v2*	$\vec{\mathbf{v}}_1 \bullet \vec{\mathbf{v}}_2$
`dxdt`	*x*	$\dot{x}$
`gcf`	*x, y*	find symbolic greatest common factor.
`mul`	*x, y*	$x\,y$ (either *x* or *y* must be a scalar)
`neg`	*x*	$-x$

`partial`	y, x	$\partial y / \partial x$ (x is scalar)

`gcf` — The symbolic "greatest common factor" (GCF) between X and Y is determined. (If X and Y have no factors in common, or one of them is a number, then the GCF is 1.)

`add` — The general method for adding two expressions X and Y is with the formula

X + Y = GCF(X, Y) * (X / (GCF(X, Y) + Y / GCF(X, Y))

After the GCF is factored out, the results are combined with `make-sum`. For example, when the expressions A*X and B*X**2 are added, the result is X*(A + B*X).

`dot` — The dot product operation is valid for two vectors, a vector and a dyad, or two dyads. The method used for applying the operation is to recursively expand expressions into multiplications and additions of subexpressions, and dot products of `uv`/`dyad` pairs. This approach eventually expands the original dot product to an expression involving operations defined for scalar algebra, together with dot products between unit-vectors. Thus, the only new primitive operation needed is the dot product between two `uv`s.

Recall that the `uv` contains a slot called *dot-products*. This contains a table with all pairs of `uv`s whose dot product is known. Initially, each table contains three entries for the three `uv`s 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 `uv`s of the trio.) If the table contains the answer, it is used. Otherwise, the dot product is between two `uv`s associated with different bodies that have not yet been analyzed. In that case, an analysis is performed as described below.

Each `body` has a slot with a direction cosine matrix relating the `uv`s for that body with the `uv`s of the parent. The `uv` whose body is furthest "down" the topology tree is transformed into an expression involving the three `uv`s of its parent body. 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 `uv`s are closer in the tree. Eventually, the process is guaranteed to stop when both arguments are `uv`s associated with the same `body`.

The results of the process are stored in the table of dot-products for one of the `uv`s, so that the "tree-climbing" and transformations (via the direction cosine matrices) are not required the next time the dot product is needed.

The method of "tree climbing" ensures that the minimum number of direction transformations is performed for each dot product operation. Thus, trigonometric simplifications are not required for this operation.

Note that the dot-product operator makes use of information from both the `uv` object from the computer algebra part of the system, and also the `body` object from the multibody part of the system.

`cross` — The cross product operation is performed using the same recursive approach as described above 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 multiplication by –1 if the table of the second `uv` is used). Otherwise, the cross-product is formulated using the expansion:

$$\vec{\mathbf{a}} \times \vec{\mathbf{b}} \rightarrow \left[\left(\vec{\mathbf{a}} \bullet \vec{\mathbf{b}}_1\right)\vec{\mathbf{b}}_1 + \left(\vec{\mathbf{a}} \bullet \vec{\mathbf{b}}_2\right)\vec{\mathbf{b}}_2 + \left(\vec{\mathbf{a}} \bullet \vec{\mathbf{b}}_3\right)\vec{\mathbf{b}}_3\right] \times \vec{\mathbf{b}} \tag{5.3.1}$$

where $\vec{\mathbf{a}}$ is the first `uv`, $\vec{\mathbf{b}}$ is the second, and $\vec{\mathbf{b}}_1$, $\vec{\mathbf{b}}_2$, and $\vec{\mathbf{b}}_3$ are the unit-vectors for the body containing $\vec{\mathbf{b}}$. As was the case for the dot product, some of the information needed to perform the operation is obtained from the `body` object from the *body* slot of the `uv` object.

`dxdt` — 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`, `indexed-sym`, `number`, or `uv` is reached. The time derivative of a `sym` or `indexed-sym` is zero if the expression is a constant, otherwise it is obtained from the *dxdt* slot.

The time derivative of a `uv` ($\vec{\mathbf{u}}$) is defined as

$$\dot{\vec{\mathbf{u}}} \rightarrow \vec{\omega}^{\mathrm{B}} \times \vec{\mathbf{u}} \tag{5.3.2}$$

where $\vec{\omega}^{\mathrm{B}}$ is the absolute rotational velocity of the `body` containing $\vec{\mathbf{u}}$, obtained from the *abs-w* slot of the `body` found from the *body* slot of the `uv`.

There are other ways in which the time derivative might be defined. For example, one could project the `uv` into the coordinate system of the fixed inertial reference and then take derivatives of the scalar components. However, eq. 5.3.2 has two strong advantages:

1. it leads to simple expressions, matching the conventional definition of the derivative of a vector fixed in a rotating reference frame.

2. the cross-product operation remains valid after small terms have been dropped and trigonometric functions have been replaced with truncated Taylor series. Thus, simplifications from small angles and small speeds can be made as soon as the small quantities appear in the analysis without causing errors in derivatives of unit-vectors taken later.

After the absolute time derivative of an expression is derived, the result is put into the *dxdt* slot for further reference.

`partial` — Partial derivatives are obtained using the same basic process as used for `dxdt`, except that (1) results are not saved, (2) the partial derivative of a `sym` or `indexed-sym` is zero unless it is equal to the argument *symbol*, in which case the partial is 1, and (3) partial derivatives of `uv`s are zero.

Multibody Operations

A few operations for dealing with points and bodies are useful for specifying forces, moments, and dependent variables of interest. These are summarized in Table 5.3.3.

The effect of `vel` can be obtained using the `pos` operator together with `dxdt`. However, the result usually involves derivatives of generalized coordinates, whereas the `vel` function provides the result as an expression involving generalized speeds.

Table 5.3.3. Summary of AUTOSIM operations for bodies and points.

Operation	Argument(s)	Description
`pos`	*point*	position vector from origin of inertial reference to *point*
`pos`	*point1, point2*	position vector from *point2* to *point1*
`rel-vel`	*point, body*	relative velocity of *point* in reference frame *body*
`rot`	*body*	rotational velocity of *body*
`vel`	*point*	velocity vector from origin of inertial reference to *point*
`vel`	*point1, point2*	absolute velocity of *point1* minus the absolute velocity of *point2*

Accelerations are obtained by combining the `dxdt` function with `rot` and/or `vel`.

Higher Level Operations

Table 5.3.4 lists mathematics operations that are derived from the above primitive functions. Some of the operators in the table have standard meanings and are implemented according to their definitions. Others are not standard, and are defined below.

`angle` — The angle between two vectors $\vec{\mathbf{v}}_1$ and $\vec{\mathbf{v}}_2$ is determined by defining three unit-vectors and projecting one onto the other two to obtain an expression for the arctangent of the angle. The steps are described below and illustrated in Figure 5.3.1:

1. The directions of the two vectors are obtained:

$$\vec{\mathbf{u}}_1 = \frac{\vec{\mathbf{v}}_1}{|\vec{\mathbf{v}}_1|} \qquad \vec{\mathbf{u}}_2 = \frac{\vec{\mathbf{v}}_2}{|\vec{\mathbf{v}}_2|} \tag{5.3.3}$$

2. A third direction is defined that lies in the plane defined by $\vec{\mathbf{v}}_1$ and $\vec{\mathbf{v}}_2$, and is orthogonal to $\vec{\mathbf{v}}_1$:

$$\vec{\mathbf{u}}_3 = \left(\vec{\mathbf{u}}_1 \times \vec{\mathbf{u}}_2\right) \times \vec{\mathbf{u}}_1 \tag{5.3.4}$$

3. The angle, θ, is defined as

$$\theta = \tan^{-1}\left(\frac{\vec{\mathbf{u}}_3 \bullet \vec{\mathbf{u}}_2}{\vec{\mathbf{u}}_1 \bullet \vec{\mathbf{u}}_2}\right) \text{sign}\left(\vec{\mathbf{v}}_3 \bullet \left[\vec{\mathbf{u}}_1 \times \vec{\mathbf{u}}_2\right]\right) \tag{5.3.5}$$

Table 5.3.4. Summary of higher-level mathematics operations.

Operation	Argument(s)	Description
`angle`	*v1*, *v2*, {*v3*}	angle between *v1* and *v2*, with sign determined by optional *v3*
`constant-part`	*exp*	constant part of expression
`convert-coordinates`	*coordinates*, *oldbody*, *newbody*, {*offset-p*}	convert *coordinates* from coordinate system of *oldbody* to the coordinates system of *newbody*.
`dir`	*v*	direction of vector, i.e., $\vec{\mathbf{v}}/\vec{\mathbf{v}}$.
`div`	*exp1*, *exp2*	invert *exp2*, then multiply with *exp1*.
`dot-plane`	*v1*, *v2*	project *vexp1* onto plane normal to *vexp2*.
`inv`	*exp*	make-power with exponent of –1
`mag`	*v*	scalar magnitude of vector, $\|\vec{\mathbf{v}}\| \rightarrow \sqrt{\vec{\mathbf{v}} \bullet \vec{\mathbf{v}}}$.
`nominal`	*exp*	find expression when all generalized coordinates are zero.

`solve-for`	*x, L, R,* {*num*}	solve for *x*, given relationship of form: *L(x) = R*.
`square`	*exp*	multiply *exp* with itself.
`sub`	*exp1, exp2*	negate *exp2* and add to *exp1*.

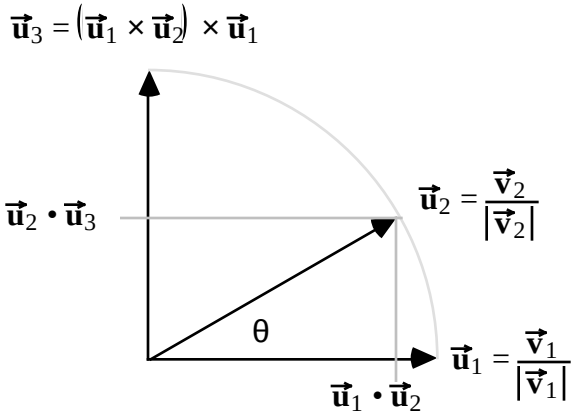

Figure 5.3.1. Angle calculation.

This method is valid for angles of any size. Results are expressed using the Fortran ATAN2 function, which accepts two arguments and is valid for the range of $-180° \le \theta \le +180°$. The `make-atan` function is used to create the resulting expression, with the possible simplifications noted earlier in Table 5.3.1. Note that an optional third vector, $\vec{\mathbf{v}}_3$, is used to establish the sign of the angle. (The sign function in eq. 5.3.5 has a value of ±1, with a sign that matches that of its argument.)

`constant-part` — This function returns zero unless (1) the expression is a constant, or (2) it is a sum (or a negative sum) with at least one constant term. It is used to obtain the part of an expression that is constant, and is useful for selecting potential divisors (for constraint equations) that are unlikely to have zero values, regardless of the values of the state variables.

`convert-coordinates` — This function returns an array of three coordinates based in *newbody*, when provided an array of three coordinates based in *oldbody*. It is used to permit the analyst to define points and directions using a specified coordinate system, rather than the coordinate system of the body containing the new point or direction. To perform the conversion, the coordinates are multiplied by the unit-vectors of *oldbody* and added to define a vector $\vec{\mathbf{r}}$. If the optional argument *offset-p* is omitted or given a value of NIL, the coordinates are converted without considering the possible offset between the origins of the coordinate systems of *oldbody* and *newbody*. That is,

$$\vec{\mathbf{r}} = a_1\vec{\mathbf{a}}_1 + a_2\vec{\mathbf{a}}_2 + a_3\vec{\mathbf{a}}_3 \{- \mathrm{pos}(B_0, A_0)\} \qquad (5.3.6)$$

where a_1, a_2, and a_3 are the three coordinates in the input array, and $\vec{\mathbf{a}}_1$, $\vec{\mathbf{a}}_2$, and $\vec{\mathbf{a}}_3$ are the three unit-vectors fixed in the *oldbody*, A_0 is the origin of *oldbody*, B_0 is the origin of *newbody*, and the curly braces indicate that the offset is optional, depending on whether the *offset-p* argument was given a non-NIL value. The output coordinates b_1, b_2, and b_3 are then defined as

$$\begin{aligned} b_1 &= \texttt{nominal}\left(\vec{\mathbf{r}} \bullet \vec{\mathbf{b}}_1\right) \\ b_2 &= \texttt{nominal}\left(\vec{\mathbf{r}} \bullet \vec{\mathbf{b}}_2\right) \\ b_3 &= \texttt{nominal}\left(\vec{\mathbf{r}} \bullet \vec{\mathbf{b}}_3\right) \end{aligned} \tag{5.3.7}$$

(The `nominal` function is defined below.) When converting the coordinates of a direction, it is appropriate to omit the *offset-p* argument. On the other hand, when converting the coordinates of a point, the *offset-p* argument should be provided with the value T.

`dot-plane` — This function describes a procedure in which a vector $\vec{\mathbf{v}}_1$ is projected onto a plane perpendicular to a second vector, $\vec{\mathbf{v}}_2$. This is done by defining the plane as a dyadic, and then taking the dot product of the vector with that dyadic. The new vector is defined as $\vec{\mathbf{v}}_1 \bullet \left(\vec{\mathbf{v}}_3\,\vec{\mathbf{v}}_3 + \vec{\mathbf{v}}_4\,\vec{\mathbf{v}}_4\right)$ where

$$\vec{\mathbf{v}}_3 = \frac{\vec{\mathbf{v}}_1 \times \vec{\mathbf{v}}_2}{\left|\vec{\mathbf{v}}_1 \times \vec{\mathbf{v}}_2\right|} \qquad\qquad \vec{\mathbf{v}}_4 = \frac{\vec{\mathbf{v}}_3 \times \vec{\mathbf{v}}_2}{\left|\vec{\mathbf{v}}_3 \times \vec{\mathbf{v}}_2\right|} \tag{5.3.8}$$

`nominal` — This function simplifies an expression by setting all state variables to zero. As noted above, it is applied in the `convert-coordinates` function. Also, it is useful for obtaining a nominal spring length.

`solve-for` — This function is used to "solve" a constraint equation. Given an equation of the form

$$L(x) = R \tag{5.3.9}$$

where x is the symbol to eliminate, this returns an expression that can be used to replace x. The expression R is assumed to be independent of x. First, a linear solution is sought. If $L(x)$ is a sum, the terms not containing x are subtracted from both sides of the equation and the function recursively calls itself with new arguments. Otherwise, the candidate solution is

$$x = R \Big/ \frac{\partial L}{\partial x} \qquad \left(\text{if } \frac{\partial^2 L}{\partial x^2} = 0 \right) \tag{5.3.10}$$

If L is not linear with respect to x, as is the case for many constraint equations involving position, then a numerical "solution" is obtained if the optional argument *num* is T. In this case, the "solution" is based on the assumption that the current value of x is close to the correct value, such that $(L - R)$ is close to zero. (This assumption is valid in the one place it is used in AUTOSIM, as will be seen in Section 8.3.) Call the current value x_0 and the corrected value x_c, and consider an expression F that is identically zero when x has the correct value. That is,

$$F(x_c) = L(x_c) - R \tag{5.3.11}$$

Expanding in a Taylor series gives the following:

$$\begin{aligned} 0 &= F(x_c) \\ &= F(x_0) + \left(\frac{\partial F(x_0)}{\partial x} \right) (x_c - x_0) + O[(x_c - x_0)^2] \end{aligned} \tag{5.3.12}$$

Ignoring the higher order terms, and solving for x_c in terms of x_0 yields the following:

$$x_c \approx x_0 - \frac{F(x_0)}{\partial F(x_0) \big/ \partial x} \tag{5.3.13}$$

The solution generated by the function `solve-for` is a recursive computational formulation that replaces the old value of x in a Fortran program with a new value. That is,

$$x \leftarrow x - \frac{F(x)}{\partial F(x) \big/ \partial x} \tag{5.3.14}$$

where the symbol "$\leftarrow$" indicates replacement. The example system analyzed in section 9.3 illustrates code of this nature.

Operations on Program Code

The equation simplifications noted earlier (simplification techniques 8, 9, and 10 in Section 5.1) are easy 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.

Introduction of Intermediate Variables and Constants

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 below, 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. The method for doing this involves a table of all expressions that have been replaced by intermediate variables. The replacements are `indexed-sym` objects, which print as elements of a Fortran array PC (for precomputed constants) or Z (for variables). A simplified version of the algorithm in `intro-var-if-new` is as follows:

- If the expression is an `indexed-sym`, a `sym`, or a `number`, it is returned.
- Else, if the expression is a vector or dyadic, terms are collected so that each unit-vector or dyad appears only once. (For example, the terms in the expression "$L_1\,\vec{\mathbf{a}}_1 + L_2\,\vec{\mathbf{b}}_1 - L_3\,\vec{\mathbf{a}}_1$" would be collected, to yield the expression "$(L_1 - L_3)\,\vec{\mathbf{a}}_1 + L_2\,\vec{\mathbf{b}}_1$.") Then, `intro-var-if-new` is applied to every scalar in the expression.
- Else, if the expression is in the table of existing intermediate variables, the corresponding `indexed-sym` is returned.
- Else, if the expression is a constant, define a new `indexed-sym`, put it at the end of the list in the `eqs` object for intermediate constants, put the expression and symbol into the table of intermediate variables, and return the new `indexed-sym`.
- Else, if any constant expressions can be factored out, do so. Apply `intro-var-if-new` to the constant part and the variable part, then apply `intro-var-if-new` to the product.
- Else, apply `intro-var-if-new` to all components of the compound expression (arguments in a `func`, factors in a `prod`, etc.), then continue.
  - If the expression is a `prod`, process the scalar factors two at a time. If the `prod` included a factor that is a `uv` or `dyad`, skip over it. Multiply the first two scalar factors and apply `intro-var-if-new` to the result. Multiply the result with the next scalar factor and apply `intro-var-if-new` to that result. Proceed until all scalar factors have been processed. The

definitions of the new `indexed-sym`s are variables, and are placed at the end of an `eqs` object used for the intermediate variables.

– Else, introduce a new `indexed-sym`, put its definition at the end of the appropriate `eqs` object, update the table, and return the new `indexed-sym`.

This algorithm is recursive, and results in a number of intermediate expressions being introduced for a single compound expression. For example, consider the expression A*(B*X + C*Y), where A, B, and C are constants and X and Y are variables. Processing this expression with the `intro-var-if-new` function leads to the following `eqs` object for intermediate constants,

```
PC(1) = A*B
PC(2) = A*C
```

and the following object for intermediate variables:

```
Z(1) = PC(1)*X
Z(2) = PC(2)*Y
Z(3) = Z(1) + Z(2)
```

The number of multiplications needed to compute the full expression has been increased from 3 in the original, to 4 with the intermediate variables. However, two of the new multiplications involve constants, leaving only two multiplications that must be performed at each time step during a numerical simulation run.

For the above algorithm to be effective, it is essential that expressions are uniquely identified in the table. For example, if the product A*(1 + COS(Q(1))) is in the table of previously replaced expressions, a search for (–COS(Q(1)) –1)*A would fail, even though the two expressions are algebraically equivalent. This is why the `make-prod` and `make-sum` functions described earlier ensure that a given product or sum always has the same structure.

The above algorithm always introduces a new intermediate variable whenever an arithmetic operation or function evaluation occurs. For simple multibody systems, this can sometimes degrade computational efficiency by eliminating possible simplifications that occur by factoring. For example, consider an expression A*U(1) which is later added to A*U(2). If both expressions are replaced by intermediate variables, say Z(5) and Z(15), the sum is (Z(5) + Z(15). It requires 2 multiplications, which occur when Z(5) and Z(15) are computed. If the intermediate variables were not introduced, the result of the addition would be A*(U(1) + U(2))—an expression with only one multiplication.

There are some reasons not to introduce a new intermediate variable if that variable will only be used once. First, some potential simplifications are not made, such as the one just described. Second, the equations become almost unreadable by humans. The equations are usually complicated to begin with, and introducing intermediate variables that only appear once compounds the difficulty. Third, some Fortran compilers optimize machine instructions for large expressions, putting temporary intermediate results directly into working registers. For machines with vector processing or other parallel computing capabilities, the compiler may further improve efficiency by breaking down complex expressions to take full advantage of the hardware. If an intermediate variable is defined in the source code, the compiler is obliged to save its value by moving it into a RAM location. For these reasons, the method described below for removing unused code is extended to also eliminate any intermediate variables that would only be used once.

Removal of Unused Code

Before the equations are written as output into a Fortran program, they are inspected for intermediate variables that are never used, or used only once. Only equations that contribute to the computation of the derivatives of the state variables or to the computation of output variables are actually written into the simulation code that is generated by AUTOSIM.

An important part of the design of AUTOSIM is that the three symbolic elements—the `sym`, the `indexed-sym`, and the `uv`—are stored in memory such that there are no copies (e.g., the object printed as "Q(2)" exists in only one place, even though it appears in more than one expression).[1] Recall 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. The `eqs` object only prints equations involving `sym`s whose *hide* slots are not set to 0. For example, if an `eqs` contains 100 equations, but only 10 involve `sym`s with *hide* counts greater than 0, then only 10 equations are printed. The other 90 equations are still in memory, but are hidden.

To count occurrences, the *hide* slots in all intermediate variables in an `eqs` are set to 0, and then equations used to compute derivatives and output variables are processed with a

---

[1] Lisp uses *pointers* to reference such objects when they are "contained" in other objects. Thus, when an elementary object is changed, all expressions "containing" that element are updated since their pointers continue to point at the changed object.

function called `validate-exp`. The `validate-exp` function operates recursively to "validate" `sym`s. If its argument is a `sym` or `indexed-sym`, it increments the count in the *hide* slot, and then applies itself recursively to the expression on the right-hand side of the equation (available from the *exp* slot). If the argument is a compound expression, `validate-exp` applies itself to all of the parts of the expression (arguments in a `func`, factors in a `prod`, etc.)

After the *hide* values have been established for all `indexed-sym`s that appear on the left-hand 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.

6. MULTIBODY DYNAMICS THEORY

As noted in Chapter 2, there is a large body of literature covering techniques for analyzing multibody systems. Traditionally, dynamics analysis methods in textbooks have started with the equations of a particle, then a system of a few particles, then a rigid body, and then a few rigid bodies (e.g., [35]). Emphasis is placed on gathering physical insight into the system, so as to introduce meaningful variables, coordinate systems, etc. By understanding the system and all of the details of the formulation of the system equations, the analyst may choose to alter the model or eliminate terms to achieve simpler equations. However, little is said about dealing with complex systems with numerous rigid bodies subject to constraints. In contrast, "multibody formalisms" have been developed and published which offer a systematic analysis method based on matrix representations [25, 26, 85, 87, 97, 110, 134, 137]. In these methods, the analysis consists of setting up matrices which are subsequently manipulated to yield the equations of motion. Because all of the details of the analysis are handled as matrix manipulations, it is more difficult for the analyst to apply simplifications.

An analysis method developed by Kane [58] is used in this work for several reasons: (1) as is the case for the multibody formalisms, it is presented as a "cookbook" methodology that can be used to systematically analyze the most complex of multibody systems, (2) the method is presented for a human analyst to follow, and it permits all of the traditional simplifications to be made by the analyst while deriving equations, (3) it requires relatively little symbolic computation, compared to other popular approaches (e.g., the Lagrangian or the direct Newton-Euler type of analysis), and (4) it has been reported to lead to highly efficient equations of motion [15, 55, 56, 57, 59, 70, 98, 125].

The purpose of this chapter is to review the basic existing method and to extend it to the form needed for computer solution. This lays the groundwork for the detailed multibody formalism developed in Chapter 8. The chapter begins with a quick summary of the first principles underlying the equations of motion for a mechanical multibody system. Next, Kane's approach is described, and extended to a form suitable for numerical integration algorithms. Terms useful for formulating the equations in a matrix structure (for computer

solution) are introduced. The equations and quantities developed in this chapter form the foundation of a multibody formalism.

Steps involving judgements, traditionally made by the analyst, are added to the formalism in Chapter 8. Also, a formal strategy for including kinematical loops and other constraints is developed in Chapter 8. (The strategy involves the derivation of coefficients introduced in this chapter.)

6.1. Fundamental Concepts

The state of the multibody system is described by state variables, which are divided into two groups: generalized coordinates and generalized speeds (see Section 3.2 for definitions). For constrained systems, some of the degrees of freedom might be eliminated by (1) holonomic constraints, imposed by geometry, and/or (2) nonholonomic constraints, imposed by motion limits. Starting with an unconstrained system of bodies, each holonomic constraint removes an independent coordinate and an independent speed, whereas each nonholonomic constraint removes only an independent speed. Further, coordinates and speeds known by the analyst to be of no interest might be omitted. Overall, the system has *n* generalized coordinates (designated q_1, q_2, … q_n) and *p* independent speeds (designated u_1, u_2, … u_p). The system is said to have *p* degrees of freedom.

The objective in analyzing a multibody system in this dissertation is to be able to compute time histories of variables of interest, in response to known inputs. To achieve this objective, it is necessary to (1) define a valid set of state variables that describe the system and can be used to compute the output variables of interest, and (2) derive equations of motion for computing the state variables as functions of time. The equations of motion are ordinary differential equations involving the state variables, their derivatives, and known functions of time. These differential equations are commonly classified into two groups: *kinematical* and *dynamical*. The kinematical equations are used to compute derivatives of the generalized coordinates, and are developed from the definitions of the state variables. The dynamical equations are used to compute derivatives of the independent speeds (accelerations), and are derived from first principles of the dynamics of rigid bodies.

Kinematical Equations

Kinematical equations define derivatives of the generalized coordinates as linear combinations of the generalized speeds. They never include influences of masses, forces, or moments. In many cases, generalized speeds are defined as the derivatives of the generalized coordinates. If so, the kinematical equations are simply

$$\begin{Bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dots \\ \dot{q}_n \end{Bmatrix} = \begin{Bmatrix} u_1 \\ u_2 \\ \dots \\ u_n \end{Bmatrix} \tag{6.1.1}$$

Sometimes, however, there are reasons for defining speeds that are not derivatives of the coordinates.

The fundamental tactic for obtaining a kinematical equation for a translational generalized coordinate derivative is to obtain two vector expressions for a point on a body: (1) as the time derivative of the position of that point, and (2) using kinematical rules involving moving reference frames and the definitions of the generalized speeds. The first vector expression involves derivatives of the generalized coordinates, whereas the second does not. Scalar equations are obtained by equating the vector expressions and dot-multiplying both by an appropriate unit vector.

For example, consider a simple vehicle model involving one rigid body constrained to planar motions. It has two translational degrees of freedom, q_1 and q_2, that define the coordinates of its center of mass in the directions $\vec{\mathbf{n}}_1$ and $\vec{\mathbf{n}}_2$, and a yaw rotational degree of freedom q_3 about the axis $\vec{\mathbf{n}}_3$. Three generalized speeds are defined:

$$u_1 = \vec{\mathbf{v}}^{B^*} \bullet \vec{\mathbf{b}}_1 \qquad u_2 = \vec{\mathbf{v}}^{B^*} \bullet \vec{\mathbf{b}}_2 \qquad u_3 = \vec{\omega}^{B} \bullet \vec{\mathbf{n}}_3 \tag{6.1.2}$$

The velocity of the mass center B* is defined implicitly in eq. 6.1.2, as

$$\vec{\mathbf{v}}^{B^*} = u_1 \vec{\mathbf{b}}_1 + u_2 \vec{\mathbf{b}}_2 \tag{6.1.3}$$

Alternatively, an expression for the velocity is obtained by taking the derivative of the position vector that goes from the fixed origin to B*:

$$\begin{aligned} \vec{\mathbf{v}}^{B^*} &= \frac{d}{dt}\left(q_1 \vec{\mathbf{n}}_1 + q_2 \vec{\mathbf{n}}_2\right) \\ &= \dot{q}_1 \vec{\mathbf{n}}_1 + \dot{q}_2 \vec{\mathbf{n}}_2 \end{aligned} \tag{6.1.4}$$

By equating eqs. 6.1.3 and 6.1.4 and dot multiplying with $\vec{\mathbf{n}}_1$ and $\vec{\mathbf{n}}_2$, two kinematical equations are obtained:

$$\dot{q}_1 \vec{\mathbf{n}}_1 \cdot \vec{\mathbf{n}}_1 + \dot{q}_2 \vec{\mathbf{n}}_2 \cdot \vec{\mathbf{n}}_1 = u_1 \vec{\mathbf{b}}_1 \cdot \vec{\mathbf{n}}_1 + u_2 \vec{\mathbf{b}}_2 \cdot \vec{\mathbf{n}}_1$$

$$\dot{q}_1 = u_1 \cos q_3 - u_2 \sin q_3 \qquad (6.1.5)$$

$$\dot{q}_1 \vec{\mathbf{n}}_1 \cdot \vec{\mathbf{n}}_2 + \dot{q}_2 \vec{\mathbf{n}}_2 \cdot \vec{\mathbf{n}}_2 = u_1 \vec{\mathbf{b}}_1 \cdot \vec{\mathbf{n}}_2 + u_2 \vec{\mathbf{b}}_2 \cdot \vec{\mathbf{n}}_2$$

$$\dot{q}_2 = u_1 \sin q_3 + u_2 \cos q_3 \qquad (6.1.6)$$

Note that these equations are linear with respect to the generalized speeds, although the coefficients are nonlinear functions of the generalized coordinates.

A similar tactic is used for rotational velocity of the body: two expressions are obtained for rotational velocity, equated, and dot-multiplied with an appropriate unit-vector. For this example, the third kinematical equation is

$$\dot{q}_3 = u_3 \qquad (6.1.7)$$

It is not always possible to obtain equations with a single unknown variable on the left-hand side. In the most general terms, the kinematical equations are written in matrix form:

$$\underline{S}\, \underline{\dot{q}} = \underline{v} \qquad (6.1.8)$$

Where $\underline{S}$ is an $n \times n$ matrix, $\underline{\dot{q}}$ is a column array of length n containing the derivatives of the generalized coordinates, and $\underline{v}$ is a column array of length n.

To develop this strategy in more detail requires knowledge of how the generalized coordinates and speeds are defined. Later, when rules are established for introducing the state variables, the formulation of the kinematical equations can be specified in complete detail.

Newton-Euler Equations

Dynamical equations are derived from first principles of the dynamics of rigid bodies, namely, the Newton and Euler equations. For a rigid body, whose principle axes are labelled 1, 2, and 3, Euler's equations are

$$I_1 \alpha_1 - (I_2 - I_3)\, \omega_2\, \omega_3 = T_1$$

$$I_2 \alpha_2 - (I_3 - I_1)\, \omega_3\, \omega_1 = T_2$$

$$I_3 \alpha_3 - (I_1 - I_2)\, \omega_1 \omega_2 = T_3 \tag{6.1.9}$$

where T_i is a component about axis i of a couple applied to the body, relative to its center of mass or a point fixed in space, I_i is a moment of inertia for the rigid body about its mass center or a point fixed in space, taken in the direction of axis i, α_i is the component of angular acceleration of the body about axis i, and ω_i is the component of angular velocity about axis i. Newton's equation is simply

$$m\, a_i = F_i \tag{6.1.10}$$

where i can be an axis oriented in any direction, F_i is the component along axis i of the resultant force applied to the body, and a_i is the component along axis i of the acceleration of the mass center.

The Newton-Euler equations are written more simply for a rigid body, independent of direction, using vector and dyadic quantities. Momentum terms for body B can be written as vectors:

$$\vec{\mathbf{P}}^B = m^B \vec{\mathbf{v}}^{B^*} \tag{6.1.11}$$

$$\vec{\mathbf{H}}^{B^*} = \overleftrightarrow{\mathbf{I}}^{B^*} \bullet \vec{\omega}^B \tag{6.1.12}$$

where $\vec{\mathbf{P}}^B$ is the translational momentum of body B, m^B is the mass, $\vec{\mathbf{H}}^{B^*}$ is the angular momentum of B about its mass center, and $\overleftrightarrow{\mathbf{I}}^{B^*}$ is the inertia dyadic of the body about the mass center. From these expressions, the Newton-Euler equations for B are

$$\sum \vec{\mathbf{F}} = \frac{d\vec{\mathbf{P}}^B}{dt} = m^B \vec{\mathbf{a}}^{B^*} \tag{6.1.13}$$

$$\sum \vec{\mathbf{T}} = \frac{d\vec{\mathbf{H}}^{B^*}}{dt} = \vec{\omega}^B \times \overleftrightarrow{\mathbf{I}}^{B^*} \bullet \vec{\omega}^B + \overleftrightarrow{\mathbf{I}}^{B^*} \bullet \vec{\alpha}^B \tag{6.1.14}$$

In the above equations, $\sum \vec{\mathbf{F}}$ is the sum of all forces applied to the body and $\sum \vec{\mathbf{T}}$ is the sum of all moments (torques of couples) applied to the body about its mass center.

Scalar expressions are obtained from eqs. 6.1.13 and 6.1.14 by projecting the vectors onto a direction of interest via the dot-product operation. For example, consider the direction defined by a unit-vector $\vec{\mathbf{u}}$. The scalar equation obtained by taking the dot-product of $\vec{\mathbf{u}}$ with eq. 6.1.13 yields a force balance similar to eq. 6.1.10 in the $\vec{\mathbf{u}}$ direction. The corresponding dot product obtained with eq. 6.1.14 results in a moment balance about the center of mass of the body, for an axis parallel with $\vec{\mathbf{u}}$. If $\vec{\mathbf{u}}$ is oriented along a principle axis of B, this moment balance is an Euler equation. (When $\vec{\mathbf{u}}$ is not oriented along a principle axis, a more complicated scalar equation is obtained that includes the

products of inertia of B.) For an unconstrained body, six independent scalar dynamical equations are obtained by taking dot products of eqs. 6.1.13 and 6.1.14 with any three orthogonal unit vectors.

The Newton-Euler equations define a linear relationship between the derivatives of velocity (translational and rotational) and the sum of the forces (for translation) or moments (for rotation) applied to the body. For a system with *p* dynamical degrees of freedom, a set of scalar equations can be obtained that has the form:

$$\underline{M}\,\underline{\dot{u}} = \underline{f} \tag{6.1.15}$$

where $\underline{M}$ is a *p* x *p* matrix called the mass matrix, $\underline{\dot{u}}$ is a column array containing the *p* derivatives of independent speeds, and $\underline{f}$ is a column array of length p, called the force array[1]. Many detailed approaches have been developed for obtaining the equations, and one such method is presented later. Here, we consider only the general concept of how the Newton-Euler equations are extended for multiple bodies.

Constrained Systems

For constrained systems, the simple approach of dotting the vector force and moment equilibrium equations with three orthogonal unit vectors is by itself insufficient because it produces too many equations for the system.

One method for dealing with a constrained system is to first obtain a set of equations for each rigid body as if the body were unconstrained, and to then add additional equations for each force and moment of constraint. The total set of equations then includes both differential equations (from the Newton-Euler relationships) and algebraic equations (from the constraints). Special numerical solution methods have been developed for solving sets of ordinary differential and algebraic equations, which are numerically similar to ordinary differential equations for "stiff" systems [29, 30]. Another approach is to search for the "most independent" state variables and integrate only those [31, 32, 43, 51, 54, 61, 68, 71, 78, 85, 129, 132]. Because holonomic constraint equations involving position are often highly nonlinear, in some formulations only the derivatives of the constraints are

---

[1] The matrix $\underline{M}$ consists of the coefficients of the derivatives of the independent speeds as they appear in the equations of motion. These coefficients are sometimes masses, sometimes moments of inertia, and sometimes expressions with units of mass or moments of inertia. The name "mass matrix" is not perfectly descriptive, but it is widely used in the literature. The array $\underline{f}$ simply includes all terms that appear on the right-hand side of each equation of motion. The elements of $\underline{f}$ have units of forces and moments, which is why the name "force array" is used.

included because they are linear. To prevent error from accumulating during the integration, "constraint stabilization" methods are used [14, 19, 84, 89, 96, 126]. (A new version of this approach is used in Chapter 8 for dealing with closed kinematical loops.) Because there are many more equations than there are degrees of freedom, formulations of this sort are sometimes called "redundant equations."

Another method for dealing with a constrained system is to introduce only one generalized speed for each nonholonomic degree of freedom, and one coordinate for each joint degree of freedom. Then there is a one-to-one correspondence between degrees of freedom and equations of motion. Formulations of this sort are sometimes called "minimal equations." In the above example involving a rigid body moving on a plane, the restriction to planar motions is the result of holonomic constraints that prevent vertical deviations or rotation about a roll axis or pitch axis. A minimal set of generalized coordinates and speeds was introduced (n=p=3). Suppose a nonholonomic constraint is also applied, by defining the forward speed as a constant. Then, the system would have three generalized coordinates and two nonholonomic degrees of freedom. To obtain minimal equations, the generalized speed u_1 would be removed and replaced with a constant.

If a given multibody system is analyzed by different methods, to obtain both minimal and redundant equations, it is usually the case that the many redundant equations are individually very simple, whereas the few minimal equations are individually more complicated [39]. There is no consensus in the literature that one approach is inherently superior to the other for general numerical solution. However, for symbolic formulations, less manipulation is needed to obtain equations in explicit form if the implicit equations are already minimal.

A minimal equation formulation strategy is developed in this dissertation that is based on Kane's approach. Forces and moments of constraint are generally not included in the equations of motion. However, methods for including the constraint forces and moments exist for cases in which they are of interest [9, 10, 58].

6.2. Kane's Approach

Kane has developed a methodology in which sums of forces and moments are projected against vector quantities called partial velocities, which will be defined shortly. The partial velocities are defined such that they account for constraints, and a minimal set of

differential equations is obtained. The procedure, as developed in a textbook [58], is summarized below for nonholonomic systems. (Note: because the following summarizes about 100 pages of text, some liberties have been taken to very briefly review tasks performed by the analyst. Also, some changes in notation have been made to accommodate methods developed later.)

First, the analyst develops a conceptual model of the system. He or she decides how many bodies are used to represent the system, and how they are kinematically related to each other. For each body, a trio of unit-vectors is established to define directions and positions relative to that body. All of the force and moment-producing components are identified. The attachments of these components to the bodies are described. Also, external forces (gravity, vehicle tire forces, aerodynamic effects, etc.) are identified.

Next, a position analysis is performed to introduce generalized coordinates and develop expressions needed to write expressions relating points of interest in the system. Generalized coordinates are introduced for each joint degree of freedom that is of interest, or which contributes to a force or moment, or which is needed to write expressions for the velocities of mass centers of bodies in the system.

A velocity analysis is performed to develop variables and expressions needed to write the velocity of any particle in the system. Generalized speeds are introduced for each joint degree of freedom such that it is possible to write an expression for the instantaneous velocity of any point on any body in the system, using only dimensional parameters, generalized coordinates, and generalized speeds. When these are introduced, the analyst should develop kinematical equations to define derivatives of the generalized coordinates in terms of generalized speeds.

The system may be subject to nonholonomic constraints, which prevent all of the generalized speeds from being mathematically independent. If there are ν generalized speeds and m nonholonomic constraints, then are are p independent speed variables, where

$$p = \nu - m \tag{6.2.1}$$

If the system includes nonholonomic constraints (i.e., $m \neq 0$) then the p independent speeds should be numbered such that $u_1 \ldots u_p$ are independent and $u_{p+1} \ldots u_\nu$ are nonholonomic. It is necessary to develop explicit expressions for the m nonholonomic speeds, using the following form:

$$u_s = \sum_{r=1}^{p} A_{sr} u_r + b_s \quad (s = p+1, \ldots \nu) \tag{6.2.2}$$

where the coefficients A_{sr} and b_s may be constants or functions of the generalized coordinates and of time (they are defined by the analyst). The equations defined in eq. 6.2.2 are nonholonomic constraint equations.

The analyst develops an expression for the angular velocity vector of each body $\vec{\omega}^B$. From each angular velocity, ν holonomic partial angular velocities are defined:

$$\vec{\omega}_i^B = \frac{\partial \vec{\omega}^B}{\partial u_i} \quad (i = 1, \ldots \nu) \tag{6.2.3}$$

A partial angular velocity is simply a coefficient appearing in an expression for angular velocity. Because angular velocities are always vectors, and speeds are always scalars, it follows that a partial angular velocity is always a vector. The total number of partial angular velocities that exists for the multibody system is the product $\nu \; N_{Bodies}$, where N_{Bodies} is the number of rigid bodies in the system.

Next, expressions are developed for the velocity vectors of the the mass centers of each body, $\vec{\mathbf{v}}^{B*}$. From these expressions, ν holonomic partial velocities are defined:

$$\vec{\mathbf{v}}_i^{B*} = \frac{\partial \vec{\mathbf{v}}^{B*}}{\partial u_i} \quad (i = 1, \ldots \nu) \tag{6.2.4}$$

If the system is nonholonomic, nonholonomic partial angular velocities and partial angular velocities are defined, using the coefficients from the constraint equations:

$$\tilde{\vec{\omega}}_r^B = \vec{\omega}_r^B + \sum_{s=p+1}^{\nu} A_{sr} \vec{\omega}_s^B \quad (r = 1, \ldots p) \tag{6.2.5}$$

Also,

$$\tilde{\vec{\mathbf{v}}}_r^{B*} = \vec{\mathbf{v}}_r^{B*} + \sum_{s=p+1}^{\nu} A_{sr} \vec{\mathbf{v}}_s^{B*} \quad (r = 1, \ldots p) \tag{6.2.6}$$

The nonholonomic partial velocities are written with a tilde over the vector arrow to distinguish them from the holonomic partial velocities.

For each body, all force vectors acting on the body are added to obtain a resultant force. Forces acting between two bodies should appear in the resultant force vectors for both

affected bodies, with opposite directions. (For example, if a spring is attached to points in bodies A and B, the direction of the force applied to B is the opposite of the direction used for A.)

For each body, the torques of all couples acting on the body are also added. These include pure torque couples (torsional springs, rotary motors, etc.) and moments of applied forces. (The moment associated with an applied force is $\vec{\mathbf{r}} \times \vec{\mathbf{f}}$, where $\vec{\mathbf{f}}$ is the force vector and $\vec{\mathbf{r}}$ is a position vector that goes from the center of mass to a point through which the force acts.) Torques acting between two bodies should appear in the resultant vectors for both affected bodies, with opposite directions.

The contributions of all active forces and torques in the system are summarized in p nonholonomic generalized active forces, defined as

$$\widetilde{F}_r = \sum_{B}^{\text{all bodies}} \left\{ \left(\sum_{t=1}^{N_{B,T}} \vec{\mathbf{T}}_{\mathbf{t}}^{B} \right) \bullet \vec{\tilde{\omega}}_r^{B} + \left(\sum_{f=1}^{N_{B,F}} \vec{\mathbf{F}}_f^{B} \right) \bullet \vec{\tilde{\mathbf{v}}}_r^{B^*} \right\} \qquad (6.2.7)$$

In the above equation, the number of torques and moments acting on body B is designated $N_{B,T}$, and the individual torques and moments are designated $\vec{\mathbf{T}}_{\mathbf{t}}^{B}$. Similarly, the number of forces acting on body B is designated $N_{B,F}$, and the individual forces are designated $\vec{\mathbf{F}}_f^{B}$. The outer summation, with index B, is meant to imply summing over all bodies in the system.

Note that eq. 6.2.7 resembles the left-hand side of the Newton-Euler equations (eqs. 6.1.13 and 6.1.14), with the vector force and torque quantities for each body in the system being projected against the partial velocity vectors and the partial angular velocity vectors associated with the body.

Expressions are developed for the angular acceleration of each body ($\vec{\alpha}^{B}$) and for the acceleration of the mass center of each body ($\vec{\mathbf{a}}^{B^*}$). With those expressions, p nonholonomic generalized inertia forces are defined as:

$$\widetilde{F}_r^* = - \sum_{B}^{\text{all bodies}} \left\{ \left(\vec{\alpha}^{B} \bullet \vec{\bar{\mathbf{I}}}^{B^*} + \vec{\omega}^{B} \times \vec{\bar{\mathbf{I}}}^{B^*} \bullet \vec{\omega}^{B} \right) \bullet \vec{\tilde{\omega}}_r^{B} + m^B \vec{\mathbf{a}}^{B^*} \bullet \vec{\tilde{\mathbf{v}}}_r^{B^*} \right\} \qquad (6.2.8)$$

where $\vec{\bar{\mathbf{I}}}^{B^*}$ is the inertia dyadic for body B about its mass center and m^B is the mass of body B. The above equation resembles the right-hand side of the Newton-Euler equations (eqs. 6.1.13 and 6.1.14). Again, vector quantities for each body are projected against the partial

velocities and partial angular velocities associated with that body. (However, the sign is reversed to accommodate the convention used by Kane.)

The final step in the analysis is the application of Kane's equation:

$$\widetilde{F}_r + \widetilde{F}_r^* = 0 \qquad (r = 1, \ldots p) \tag{6.2.9}$$

This results in *p* scalar equations involving (1) system parameters, (2) the *n* generalized coordinates ($q_1 \ldots q_n$), (3) the *p* independent speeds ($u_1 \ldots u_p$), and (4) the *p* derivatives of the independent speeds, $\dot{u}_1 \ldots \dot{u}_p$. Expanding the generalized active and inertial forces yields the following form of Kane's equation:

$$0 = \sum_{B}^{\text{all bodies}} \left(\left(\sum_{t=1}^{N_{B,T}} \vec{T}_t^B - \vec{\alpha}^B \bullet \overleftrightarrow{I}^{B*} - \vec{\omega}^B \times \overleftrightarrow{I}^{B*} \bullet \vec{\omega}^B \right) \bullet \vec{\widetilde{\omega}}_r^B + \left(\sum_{f=1}^{N_{B,F}} \vec{F}_f^B - m^B \vec{a}^{B*} \right) \bullet \vec{\widetilde{v}}_r^{B*} \right) \tag{6.2.10}$$

This use of partial velocities derives from Lagrange's form of D'Alembert's principle (i.e., the virtual work associated with constraint forces and torques must vanish), and reflects the facts that (1) forces and moments can do work only if there is movement (i.e., a speed), and (2) the partial velocities are the directions in which those movements take place. A very similar method is used in the NEWEUL formulation, based on Jourdain's principle (i.e., the virtual power associated with constraint forces and torques must vanish) [107, 112]. Also, a similar formulation was developed by Passerello and Huston [47, 48, 49, 50, 91]. Further, Kane's formulation is similar to, but simpler than, the Gibbs-Appell equations [15, 22, 23, 59, 70, 98].

The above analysis method immediately applies several of the simplification methods described in Section 5.1. First, it permits the introduction of "natural" state variables, including generalized speeds that are not derivatives of the generalized coordinates (technique no. 1). If there is reason to think that a certain set of variables is in fact optimal, the analyst is free to use that set. (Rules will be developed to Chapter 8 to define state variables that are, if not optimal, at least "very good.")

A second potential simplification occurs because non-working forces and moments are never introduced (technique no. 3).

6.3. Overview of Dynamics Analysis Method

Once the model is conceived by a human analyst as a system of idealized elements (rigid bodies, force and moment elements, constraint relationships, etc.), the creative part of the analysis effort has largely ended. The above method provides a clear path towards the subsequent formulation of dynamical equations. However, it includes many instructions to "introduce..." or "formulate..." or "obtain..." expressions. These instructions are satisfactory for human analysts but lack the detail needed for computer implementation. Also, the form of the final equations is not directly suited for incorporation into a simulation code. That is, equation 6.2.10 does not explicitly define derivatives of state variables, nor does it implicitly define the derivatives in terms of a mass matrix and force array. A human analyst typically obtains equations in the desired form by inspection and further manipulation as needed to obtain the form of eqs. 6.1.8 and 6.1.15. Therefore, the Kane method will now be extended so that the analytical efforts applied after the model is conceived can be fully automated using the symbolic manipulation tools developed in Chapter 5.

Additional Definitions

An analysis procedure can be easier to understand when it is defined in terms of familiar quantities, such as velocities and accelerations. However, when understanding is not at issue (because the procedure is being programmed), there is little point in building expressions that will later be decomposed, if the components are known from the start. For example, the central[1] translational velocities of the bodies do not actually appear in the equations of motion. (They are used only to define the concept of partial velocities.) Also, it will be seen that the accelerations often do not appear in one place in the final form of the equations of motion.

In the multibody formalisms described by Wampler and Nielan [83, 125], Kane's equations were converted to matrix form, to facilitate the automated construction of equations of motion as is done with other multibody formalisms. In this work, the original vector/dyadic notation is retained. However, terms called "remainders" that were defined

[1] "Central" velocities and accelerations refer to motions of a mass center.

by Wampler (and Rosenthal [99]) are used here as well, and are extended for nonholonomic systems.

Partial velocities and partial angular velocities will be be introduced directly and used to define other quantities, such as velocities and accelerations. The definition of partial angular velocities from eq. 6.2.3 is converted to the following:

$$\vec{\omega}^B = \vec{\omega}_t^B + \sum_{i=1}^{\nu} u_i \vec{\omega}_i^B \tag{6.3.1}$$

where $\vec{\omega}_t^B$ is a function of time. Kane keeps this term throughout the presentation, to accommodate velocity inputs. However, inputs of this sort can just as well be accommodated by introducing a nonholonomic constraint with a nonzero coefficient b (see eq. 6.2.2). All systems will be considered to be potentially nonholonomic in this dissertation. Thus, the term $\vec{\omega}_t^B$ is defined as zero so that eq. 6.3.1 can be replaced with the simpler form:

$$\vec{\omega}^B = \sum_{i=1}^{\nu} u_i \vec{\omega}_i^B \tag{6.3.2}$$

Angular velocity can also be written as a sum involving nonholonomic partial angular velocities. Combining eqs. 6.3.2 and 6.2.2 yields

$$\vec{\omega}^B = \sum_{r=1}^{p} u_r \vec{\omega}_r^B + \sum_{s=p+1}^{\nu} \left(\sum_{r=1}^{p} A_{sr} u_r + b_s \right) \vec{\omega}_s^B \tag{6.3.3}$$

This can be simplified by writing the right-hand side in terms of the nonholonomic partial angular velocities, as defined in eq. 6.2.5:

$$\vec{\omega}^B = \sum_{r=1}^{p} u_r \tilde{\vec{\omega}}_r^B + \sum_{s=p+1}^{\nu} b_s \vec{\omega}_s^B \tag{6.3.4}$$

An expression for angular acceleration can also be developed in terms of partial angular velocities:

$$\vec{\alpha}^B = \frac{d\vec{\omega}^B}{dt} \tag{6.3.5}$$

Substituting eq. 6.3.4 into 6.3.5 yields

$$\vec{\alpha}^B = \sum_{r=1}^{p} \left(\frac{du_r}{dt} \tilde{\vec{\omega}}_r^B + u_r \frac{d\tilde{\vec{\omega}}_r^B}{dt} \right) + \sum_{s=p+1}^{\nu} \frac{d\left(b_s \vec{\omega}_s^B\right)}{dt}$$

$$= \sum_{r=1}^{p} \left(\dot{u}_r \tilde{\vec{\omega}}_r^B + u_r \frac{d\tilde{\vec{\omega}}_r^B}{dt} \right) + \sum_{s=p+1}^{\nu} \frac{d\left(b_s \vec{\omega}_s^B\right)}{dt}$$

$$= \tilde{\vec{\alpha}}_{rem}^B + \sum_{r=1}^{p} \dot{u}_r \tilde{\vec{\omega}}_r^B \qquad (6.3.6)$$

where the nonholonomic angular acceleration remainder, $\tilde{\vec{\alpha}}_{rem}^B$, is defined to simplify later notation:

$$\tilde{\vec{\alpha}}_{rem}^B = \vec{\alpha}^B - \sum_{r=1}^{p} \dot{u}_r \tilde{\vec{\omega}}_r^B$$

$$= \sum_{r=1}^{p} u_r \frac{d\tilde{\vec{\omega}}_r^B}{dt} + \sum_{s=p+1}^{\nu} \frac{d\left(b_s \vec{\omega}_s^B\right)}{dt} \qquad (6.3.7)$$

The nonholonomic angular acceleration remainder is the part of the angular acceleration that is put on the right-hand side of the equal sign in the equations of motion, in the force array. It contains quadratic speed terms, and is sometimes identified as the "nonlinear" or "quadratic" component of angular acceleration. By substituting eq. 6.2.2 into eq. 6.3.7, it is written as follows:

$$\tilde{\vec{\alpha}}_{rem}^B = \sum_{r=1}^{p} u_r \left(\frac{d\vec{\omega}_r^B}{dt} + \sum_{s=p+1}^{\nu} \frac{d\left(A_{sr} \vec{\omega}_s^B\right)}{dt} \right) + \sum_{s=p+1}^{\nu} \frac{d\left(b_s \vec{\omega}_s^B\right)}{dt}$$

$$= \sum_{r=1}^{p} u_r \frac{d\vec{\omega}_r^B}{dt} + \sum_{s=p+1}^{\nu} \left\{ \left[\sum_{r=1}^{p} u_r \frac{d\left(A_{sr} \vec{\omega}_s^B\right)}{dt} \right] + \frac{d\left(b_s \vec{\omega}_s^B\right)}{dt} \right\}$$

$$= \sum_{r=1}^{p} u_r \frac{d\vec{\omega}_r^B}{dt} + \sum_{s=p+1}^{\nu} \left\{ \left(\sum_{r=1}^{p} u_r \left[A_{sr} \frac{d\left(\vec{\omega}_s^B\right)}{dt} + \dot{A}_{sr} \vec{\omega}_s^B \right] \right) + b_s \frac{d\left(\vec{\omega}_s^B\right)}{dt} + \dot{b}_s \vec{\omega}_s^B \right\}$$

$$= \sum_{r=1}^{p} u_r \frac{d\vec{\omega}_r^B}{dt} + \sum_{s=p+1}^{\nu} \left\{ \vec{\omega}_s^B \left(\dot{b}_s + \sum_{r=1}^{p} u_r \dot{A}_{sr} \right) + \frac{d\left(\vec{\omega}_s^B\right)}{dt} \left(b_s + \sum_{r=1}^{p} u_r A_{sr} \right) \right\}$$

$$= \sum_{i=1}^{\nu} u_i \frac{d\vec{\omega}_i^B}{dt} + \sum_{s=p+1}^{\nu} \vec{\omega}_s^B \left(\dot{b}_s + \sum_{r=1}^{p} u_r \dot{A}_{sr} \right) \qquad (6.3.8)$$

Two new terms are now introduced, to allow a simpler expression of eq. 6.3.8.

$$\vec{\vec{\alpha}}^{B}_{rem} = \vec{\alpha}^{B}_{rem} + \sum_{s=p+1}^{\nu} \vec{\omega}^{B}_{s}\, c_s \tag{6.3.9}$$

where the holonomic angular acceleration remainder, $\vec{\alpha}^{B}_{rem}$, is defined as:

$$\vec{\alpha}^{B}_{rem} = \sum_{i=1}^{\nu} u_i \frac{d\vec{\omega}^{B}_{i}}{dt} \tag{6.3.10}$$

and *m* constraint acceleration coefficients are defined as:

$$c_s = \dot{b}_s + \sum_{r=1}^{p} u_r \dot{A}_{sr} \tag{6.3.11}$$

To summarize, we have taken the angular acceleration of B and broken it up into several terms: (1) the part containing coefficients for the derivatives of the independent speeds, which contributes to the mass matrix, (2) a holonomic part containing products of all generalized speeds and holonomic partial angular velocities, and (3) a nonholonomic part containing products of the derivatives of the constraint coefficients and the holonomic partial angular velocities of the nonholonomic speeds. The second and third parts define the nonholonomic remainder that appears in the force array, and contains the quadratic terms in the angular acceleration.

A similar convention is used to develop an expression for the velocity of the mass center using holonomic partial velocities,

$$\vec{\mathbf{v}}^{B^*} = \vec{\mathbf{v}}^{B^*}_{t} + \sum_{i=1}^{\nu} u_i\, \vec{\mathbf{v}}^{B^*}_{i} \tag{6.3.12}$$

As with the angular velocity, the component, $\vec{\mathbf{v}}^{B^*}_{t}$ is defined as zero. (A predetermined function of time can be accommodated in the velocity as a nonholonomic constraint.) Thus, eq. 6.3.12 simplifies to the form

$$\vec{\mathbf{v}}^{B^*} = \sum_{i=1}^{\nu} u_i\, \vec{\mathbf{v}}^{B^*}_{i} \tag{6.3.13}$$

The central velocity can just as well be written in terms of nonholonomic partial velocities:

$$\vec{\mathbf{v}}^{B^*} = \sum_{r=1}^{p} u_r \tilde{\vec{\mathbf{v}}}_r^{B^*} + \sum_{s=p+1}^{\nu} b_s \vec{\mathbf{v}}_s^{B^*} \tag{6.3.14}$$

The acceleration of the mass center can be written using the nonholonomic partial velocities:

$$\begin{aligned}\vec{\mathbf{a}}^{B^*} &= \frac{d\vec{\mathbf{v}}^{B^*}}{dt} \\ &= \frac{d\,\tilde{\vec{\mathbf{v}}}_t^{B^*}}{dt} + \sum_{r=1}^{p}\left(\dot{u}_r\tilde{\vec{\mathbf{v}}}_r^{B^*} + u_r\frac{d\tilde{\vec{\mathbf{v}}}_r^{B^*}}{dt}\right) \\ &= \tilde{\vec{\mathbf{a}}}_{rem}^{B^*} + \sum_{r=1}^{p}\dot{u}_r\tilde{\vec{\mathbf{v}}}_r^{B^*}\end{aligned} \tag{6.3.15}$$

where $\tilde{\vec{\mathbf{a}}}_{rem}^{B^*}$ is called the nonholonomic central acceleration remainder. It contains the quadratic speed terms in the acceleration, and appear on the right-hand side of the equations, in the force array. It is defined similarly to the nonholonomic angular acceleration remainder:

$$\begin{aligned}\tilde{\vec{\mathbf{a}}}_{rem}^{B^*} &= \vec{\mathbf{a}}^{B^*} - \sum_{r=1}^{p}\dot{u}_r\tilde{\vec{\mathbf{v}}}_r^{B^*} \\ &= \sum_{r=1}^{n} u_r\frac{d\vec{\mathbf{v}}_r^{B^*}}{dt} + \sum_{s=p+1}^{\nu}\vec{\mathbf{v}}_s^{B^*}\left(\dot{b}_s + \sum_{r=1}^{p} u_r\dot{A}_{sr}\right) \\ &= \vec{\mathbf{a}}_{rem}^{B^*} + \sum_{s=p+1}^{\nu}\vec{\mathbf{v}}_s^{B^*}\,c_s\end{aligned} \tag{6.3.16}$$

$\vec{\mathbf{a}}_{rem}^{B^*}$ is the holonomic central acceleration remainder, defined as

$$\vec{\mathbf{a}}_{rem}^{B^*} = \sum_{i=1}^{\nu} u_i\frac{d\vec{\mathbf{v}}_i^{B^*}}{dt} \tag{6.3.17}$$

Implicit Dynamical Equations

At this point, all of the motion terms in the dynamical equations have been defined as explicit functions of (1) the partial velocities and partial angular velocities, (2) the independent speeds and their derivatives, and (3) the constraint coefficients and their derivatives.

These equations are linear with respect to the accelerations, and therefore they can be put into the form desired for the dynamical equations:

$$\underline{M}\,\underline{\dot{u}} = \underline{f} \tag{6.3.18}$$

By substituting eqs. 6.3.6 and 6.3.15 into 6.2.10 and comparing with 6.3.18, the coefficient in the mass matrix for a particular row i and column j is obtained

$$M_{ij} = \sum_{B}^{\text{all bodies}} \left(\vec{\vec{\omega}}_j^{B} \bullet \vec{\vec{I}}^{B*} \bullet \vec{\vec{\omega}}_i^{B} + m^B \vec{\vec{v}}_j^{B*} \bullet \vec{\vec{v}}_i^{B*} \right) \tag{6.3.19}$$

Subtracting eq. 6.3.19 from eq. 6.2.10 yields the coefficient for element i in the force array (corresponding to row i in the mass matrix):

$$f_i = \sum_{B}^{\text{all bodies}} \left\{ \left(\sum_{t=1}^{N_{B,T}} \vec{T}_t^{B} - \vec{\alpha}_{rem}^{B} \bullet \vec{\vec{I}}^{B*} - \vec{\omega}^{B} \times \vec{\vec{I}}^{B*} \bullet \vec{\omega}^{B} \right) \bullet \vec{\vec{\omega}}_i^{B} + \left(\sum_{f=1}^{N_{B,F}} \vec{F}_f^{B} - m^B \vec{a}_{rem}^{B*} \right) \bullet \vec{\vec{v}}_i^{B*} \right\} \tag{6.3.20}$$

Equations 6.3.18 through 6.3.20 implicitly define the *p* accelerations in terms of known quantities. The accelerations can be computed using equation solving algorithms for linear algebra, which are implemented symbolically as described in the next chapter.

7. UNCOUPLING ALGEBRAIC EQUATIONS

Both the kinematical equations and the dynamical equations occur naturally in implicit form, defining sets of simultaneous algebraic equations that must be solved to obtain the derivatives. This formulation is often described as a set of *coupled* equations. Finding the solution for the independent variables is called *uncoupling the equations*. The matrix form of the simultaneous linear equations is

$$\underline{A}\,\underline{x} = \underline{y} \tag{7.1}$$

where $\underline{A}$ is an $n \times n$ square matrix, $\underline{x}$ is a column array of n unknown variables, and $\underline{y}$ is a column array of n known values.

The solution of simultaneous linear equations is a well developed area in the field of numerical analysis. A variety of specialized algorithms have been developed for different classes of problems, as characterized by the structure of $\underline{A}$. However, even the best generalized solution method can result in extraneous computations involving elements of $\underline{A}$ that are zero for a particular multibody system, but which are not, in general, zero for all systems.

7.1 Lower-Upper Triangular Decomposition (LUD)

When the only known structure properties of the $\underline{A}$ matrix are that it is non-singular, then lower-upper (LU) triangular decomposition is the appropriate solution method [93][1]. A set of equations involving a positive definite matrix can be solved by defining two triangular matrices,

---

[1] For numerical analysis, a more efficient method exists when the matrix is known to be symmetric (as is the case for the mass matrix). The method is to decompose the matrix into a product of a triangular matrix and its transpose, e.g., $M = G\,G^T$. Cholesky's method provides a solution with half of the computation required for the LU method presented here. However, when the analyses are performed symbolically, the Cholesky method requires more manipulation to simplify the equations. This is because many of the elements in the G triangular matrix are square roots of expressions, and a certain amount of manipulation is required to eliminate the square roots. Given sufficient symbolic manipulation, both methods yield the same set of explicit equations.

$$\underline{A} = \underline{L}\,\underline{U} \tag{7.1.1}$$

where the matrices $\underline{L}$ and $\underline{U}$ have the following structures:

$$\underline{L} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \lambda_{21} & 1 & 0 & \dots & 0 \\ \lambda_{31} & \lambda_{32} & 1 & \dots & \dots \\ \dots & \dots & \dots & \dots & 0 \\ \lambda_{n1} & \dots & \dots & \lambda_{nn-1} & 1 \end{bmatrix} \tag{7.1.2}$$

$$\underline{U} = \begin{bmatrix} \upsilon_{11} & \upsilon_{12} & \upsilon_{13} & \dots & \upsilon_{1n} \\ 0 & \upsilon_{22} & \upsilon_{23} & \dots & \upsilon_{2n} \\ 0 & 0 & \upsilon_{33} & \dots & \dots \\ \dots & \dots & \dots & \dots & \upsilon_{n-1n} \\ 0 & 0 & \dots & 0 & \upsilon_{nn} \end{bmatrix} \tag{7.1.3}$$

If the L and U matrices can be found, then the original set of *n* equations are replaced by two sets of *n* equations.

$$\underline{L}\,\underline{z} = \underline{y} \qquad\qquad \underline{U}\,\underline{x} = \underline{z} \tag{7.1.4}$$

The first of these equation sets is solved to obtain the *n* elements of the $\underline{z}$ array, using forward substitution:

$$z_j = y_j - \sum_{k=1}^{j-1} \lambda_{jk}\, z_k \qquad (j = 1, 2, \dots n) \tag{7.1.5}$$

Eq. 7.1.5 is recursive, because for all j greater than 1, the evaluation of y_j involves all of the previously determined values. Thus, the index j must be incremented as shown.

The second set of equations, which provides the desired values of the $\underline{x}$ array, is solved using backward substitution:

$$x_j = \frac{1}{\upsilon_{jj}} \left(z_j - \sum_{k=j+1}^{n} \upsilon_{jk}\, x_k \right) \qquad (j = n, n-1, \dots 1) \tag{7.1.6}$$

The backward substitution is also recursive, and requires that j be decremented from n to 1.

Because half of the coefficients in $\underline{L}$ and $\underline{U}$ are known by their definitions to be 0 or 1, it is possible to store the other elements of both the $\underline{L}$ and $\underline{U}$ arrays in a single square array that will be designated $\underline{LU}$.

$$\underline{LU} = \begin{bmatrix} \upsilon_{11} & \upsilon_{12} & \upsilon_{13} & \dots & \upsilon_{1n} \\ \lambda_{21} & \upsilon_{22} & \upsilon_{23} & \dots & \upsilon_{2n} \\ \lambda_{31} & \lambda_{32} & \upsilon_{33} & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \lambda_{n1} & \dots & \dots & \dots & \upsilon_{nn} \end{bmatrix} \tag{7.1.7}$$

Crout's algorithm provides a relatively simple procedure for obtaining these elements. The procedure is recursive, and is performed for each column j in the matrix where j is incremented from 1 to n.

$$\upsilon_{ij} = \text{« } A_{ij} - \sum_{k=1}^{i-1} \lambda_{ik}\, \upsilon_{kj} \text{ »} \qquad (i = 1, \dots j) \tag{7.1.8}$$

$$\lambda_{ij} = \text{« } \frac{1}{\upsilon_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} \lambda_{ik}\, \upsilon_{kj} \right) \text{ »} \qquad (i = j + 1, \dots n) \tag{7.1.9}$$

Eqs. 7.1.5 through 7.1.9 provide a means to compute each unknown x_j using eq. 7.1.6. Because the equations are recursive, the x variables must be computed in a particular order, from x_n to x_1.

To exploit the sparsity of a particular $\underline{A}$ matrix, the solution is developed explicitly in symbolic form. Because the above equations are highly recursive, the computer code developed symbolically by applying these equations is also highly recursive (see examples in Appendices B through E).

When the solution developed here is written into a Fortran program, it is intended that each arithmetic operation is performed only once. An expression that appears more than once is replaced with an intermediate variable, so that the intermediate variable is used subsequently. The replacement of an expression with an intermediate variable is made by using the AUTOSIM function `intro-var-if-new` (described in section 5.3). In the above two equations, the invocation of this function is indicated by enclosing an expression with the symbols "«" and "»." The result is that each element in $\underline{LU}$ is represented by a symbol. That is, the expression on the right-hand side of either of the above equations is replaced with a symbol that is used in subsequent occurrences of the expression in the recursion. This guarantees that in the worst case (see Section 7.2), the symbolic LUD

solution requires exactly the same number of arithmetic operation as when the procedure is performed numerically. In all other cases, the explicit symbolic solution is more efficient because terms that are symbolically zero do not appear in the solution.

The only division operations required in the above solution method occur in eqs. 7.1.6 and 7.1.9. In both cases, the divisor is υ_{jj}, a term that includes A_{jj}. Thus, this method should only be used if the expressions in the diagonal of the $\underline{A}$ matrix are nonzero at all times. This condition is in fact satisfied for the dynamical and kinematical equations. The definition provided in the previous chapter for elements of the mass matrix guarantees that all diagonal elements of the mass matrix are nonzero for well-posed models of mechanical systems. Also, the method developed in the next chapter to form the kinematical equations ensures that the diagonal elements in the array $\underline{S}$ are all unity.

7.2 Ordering of State Variables

Upon inspecting Crout's algorithm, as defined in eqs. 7.1.8 and 7.1.9, it can be seen that the number of multiplications needed to obtain the matrix $\underline{LU}$ is of the order n^3 if all of the indicated multiplications are performed. This is because there are three nested loops:

1. The algorithm proceeds through columns j=1, n.
2. For each column j, the upper elements υ_{ij} are computed for rows i=1, j. Also, the lower elements λ_{ij} are computed for rows i=j+1, n.
3. For each element in $\underline{LU}$, a summation is needed that involves the index k, where k goes from 1 to either j or i (see eqs. 7.1.8 and 7.1.9).

However, when the algorithm is performed symbolically, the full number of operations is needed only when all multiplications in eqs. 7.1.8 and 7.1.9 yield non-zero results. For example, in eq. 7.1.8, if either λ_{ik} or υ_{kj} is zero, the symbolic multiplication yields zero, and a product is not written into the numerical analysis code.

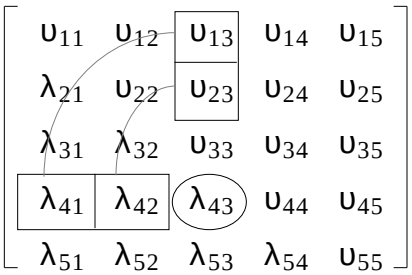

Figure 7.2.1. View of the computation of an element in the LU matrix.

Consider the number of operations needed to compute one element in a 5x5 array, e.g., λ_{43}. In Figure 7.2.1, the elements that are used in eq. 7.1.9 are shown in boxes. There are two multiplications of elements that have already been obtained (λ_{41} υ_{13} and λ_{42} υ_{23}), indicated in the figure by arcs. In order for the element λ_{43} to be zero, it is necessary that (1) the corresponding element A_{43} in the original matrix is zero, (2) either λ_{41} or υ_{13} is zero, and (3) either λ_{42} or υ_{23} is zero.[1] When A_{ij} is zero, but the corresponding element of the $\underline{LU}$ matrix is not zero because one of the conditions is not satisfied, the LU decomposition is said to have caused *matrix fill.*

If all elements in $\underline{A}$ are non-zero, there is no possibility of matrix fill, because it is already full. (Computationally, this is the worst case.) Also, if $\underline{A}$ is diagonal, there is no possibility of fill. (Computationally, this is the best case). However, if $\underline{A}$ is sparse, but not diagonal, the possibility exists. It turns out the matrix $\underline{S}$ (from the kinematical equations) is very nearly diagonal, and fill is not a problem. However, the mass matrix $\underline{M}$ is usually somewhere between the two extreme cases. Hence, the structure of $\underline{M}$ can influence the computational effort needed to uncouple the equations of motion.

Recall that the mass matrix is symmetrical. Several possible structures for a symmetric matrix $\underline{A}$ are shown in Table 7.2.1, along with the structures of the corresponding $\underline{LU}$ matrices. (Zero elements are shown in the table with zeros, non-zero elements are shown with dots.)

There is clearly less fill when the matrix is structured such the most zeros are found towards the upper-left region of the matrix. The specific locations of the zeros is determined by the ordering of the variables. For example, cases 3 and 4 could represent the same set of equations, differing only in the ordering of the variables in the $\underline{x}$ array.

---

[1] These conditions do not consider the possibility that non-zero terms will cancel. Such occurrences are not considered because the are extremely rare in the mass matrices obtained for multibody systems.

Table 7.2.1. Matrix-fill for several structures of the A matrix.

Case	Structure of A	Structure of LU
1. (Full)	$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}$	$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}$
2. (Diagonal)	$\begin{bmatrix} \bullet & 0 & 0 & 0 & 0 & 0 \\ 0 & \bullet & 0 & 0 & 0 & 0 \\ 0 & 0 & \bullet & 0 & 0 & 0 \\ 0 & 0 & 0 & \bullet & 0 & 0 \\ 0 & 0 & 0 & 0 & \bullet & 0 \\ 0 & 0 & 0 & 0 & 0 & \bullet \end{bmatrix}$	$\begin{bmatrix} \bullet & 0 & 0 & 0 & 0 & 0 \\ 0 & \bullet & 0 & 0 & 0 & 0 \\ 0 & 0 & \bullet & 0 & 0 & 0 \\ 0 & 0 & 0 & \bullet & 0 & 0 \\ 0 & 0 & 0 & 0 & \bullet & 0 \\ 0 & 0 & 0 & 0 & 0 & \bullet \end{bmatrix}$
3. (maximum fill)	$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & 0 & 0 & 0 & 0 \\ \bullet & 0 & \bullet & 0 & 0 & 0 \\ \bullet & 0 & 0 & \bullet & 0 & 0 \\ \bullet & 0 & 0 & 0 & \bullet & 0 \\ \bullet & 0 & 0 & 0 & 0 & \bullet \end{bmatrix}$	$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}$
4. (no fill)	$\begin{bmatrix} \bullet & 0 & 0 & 0 & 0 & \bullet \\ 0 & \bullet & 0 & 0 & 0 & \bullet \\ 0 & 0 & \bullet & 0 & 0 & \bullet \\ 0 & 0 & 0 & \bullet & 0 & \bullet \\ 0 & 0 & 0 & 0 & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}$	$\begin{bmatrix} \bullet & 0 & 0 & 0 & 0 & \bullet \\ 0 & \bullet & 0 & 0 & 0 & \bullet \\ 0 & 0 & \bullet & 0 & 0 & \bullet \\ 0 & 0 & 0 & \bullet & 0 & \bullet \\ 0 & 0 & 0 & 0 & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}$
5. (no fill)	$\begin{bmatrix} \bullet & 0 & 0 & 0 & 0 & \bullet \\ 0 & \bullet & 0 & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}$	$\begin{bmatrix} \bullet & 0 & 0 & 0 & 0 & \bullet \\ 0 & \bullet & 0 & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}$

Recognizing the significance of the ordering, the solution method used for the dynamical equations includes an additional step of permuting the mass matrix. Initially, the

mass matrix is formulated using an order that is convenient, based on the way in which the generalized speeds are stored in memory. Then, each row in the matrix is inspected to count the number of zero elements. The generalized speeds are then ordered such that the variable with the highest number of zeros is first and the variable with the least number is last. The mass matrix and force arrays are permuted accordingly, such the the equations defined by the matrices remain valid. Then, Crout's algorithm is used to symbolically uncouple the equations.

To some extent we are applying one of the simplification techniques used by some programmers to improve efficiency. When intermediate variables are introduced appropriately, the symbolic solution of the acceleration equations results in an efficiency at least as good as can be obtained from a carefully partitioned formulation. However, it should be noted that a potential drawback of this approach is that the structure of the system is "lost" in the building of a mass matrix which is later decomposed. Recently, a number of recursive "Order-n" formulations have been published that offer greater efficiency for systems with a "chain" topology when the length of the chain exceeds a certain number, generally around n=10 [13, 25, 128, 129, 130]. For models of ground vehicles, the formulation presented here is usually better. (Also, for the six-link Stanford Arm" robot analysed in Section 9.6, the formulation developed using the methods of Chapters 5 through 8 was about 60% more efficient than a recursive $O(n^2)$ formulation [99]). On the other hand, a recursive O(n) or $O(n^2)$ formulation should be considered for systems with "long" chain topologies.

8. A MULTIBODY FORMALISM

In this chapter, a formal procedure is developed that can be applied automatically after an analyst has conceived a model to represent the multibody system. This kind of procedure is called a *multibody formalism*. The objective is to create a complete, valid, specialized simulation code of the sort described in Chapter 4, from a description of (1) how rigid bodies in a specific system are related kinematically, and (2) how force- and moment-producing components act on those bodies. The formalism combines concepts and general methods introduced in Chapters 5, 6, and 7.

The full process has been organized into the five steps summarized below:

1. *Describe System.* The analyst describes the objects comprising the multibody system using a small set of AUTOSIM macros. As each body is added, a `body` object is created and several analyses are immediately performed to assign values to slots in order to support the computer algebra functions.

   Points of interest on rigid bodies are identified by the analyst, and corresponding `point` objects are created by the computer.

   Active forces and moments are described, and corresponding `force` and `moment` objects are created.

   Additional equations are generated for nonholonomic constraints and closed kinematical loops.

2. *Kinematical Analysis*. Kinematical equations are formed that define derivatives of generalized coordinates as functions of the independent speeds.

3. *Constraint Analysis.* The constraint equations obtained in step 1 are processed to obtain coefficients required in the dynamics analysis.

4. *Dynamics Analysis.* Terms needed for Kane's equations are obtained using a variety of formulations. The generic dynamical equations presented in Chapter 6 are then applied to obtain a mass matrix and force array. The implicit equations are

solved symbolically, as described in Chapter 7, to obtain explicit expressions for the derivatives of the independent speeds.

5. *Write Fortran Program.* A complete simulation code is written in Fortran that (1) reads input parameters, (2) simulates the multibody system, and (3) generates an output file with predicted time histories of output variables.

These five steps are described in more detail in the following sections.

8.1. Describing the System

All of the parts of the multibody system can be represented using the computer data objects presented in Chapter 5. Lisp macros and functions in AUTOSIM used by the analyst to build the description of the multibody system on the computer are summarized in Table 8.1.1. (Reference material for these macros is provided in Appendix A, and many examples of their use appear in the next chapter.) Basically, each macro creates an appropriate object and assigns data to slots in the object.

Table 8.1.1. AUTOSIM macros for describing a multibody system.

Lisp form	Purpose
`add-body`	describe one body completely, including its position in the system topology, the kinematics of its joint, and the mass and inertial properties of its rigid body.
`add-constraint`	introduce a constraint equation that will be used to eliminate one state variable.
`add-gravity`	apply a gravitational force to each body with mass.
`add-line-force`	describe force-producing component (direction of force is known).
`add-moment`	describe moment-producing component.
`add-point`	identify point of interest on a body.
`add-strut`	describe force-producing component (end points are known).
`large`	declare parameters to be “large” with small-order of –1.
`no-movement`	apply holonomic constraint for closed kinematical loop.
`small`	declare `sym`s to be “small” with a small-order of 1.

The `add-body` macro creates a `body` object to represent one of the rigid bodies in the system. It also performs several analyses to put information into slots of the new object,

that are required in order for the vector algebra operations to work. The methods used to determine values that are put into the slots of the `body` object are detailed later in this section.

The macros `add-line-force`, `add-strut`, and `add-moment` create data objects corresponding the force- and moment-producing components in the multibody system, and then assign the slots using data provided by the analyst. The objects are not manipulated until the entire system has been entered. Of course, the expressions provided for force and moment magnitudes may involve considerable algebra, but this is handled by the basic algebra routines and does not involve the multibody objects, except for functions that obtain information from the `body` objects.

In a similar vein, the macro `add-point` simply creates a data object to represent a point fixed on a body. The point may or may not be used in subsequent analyses.

The `add-gravity` macro adds the effect of a uniform gravitational field. The effect is as if a force is applied to every mass center with the direction of the gravitational field and an amplitude $g\ m^B$, where g is the gravitational constant. However, upon inspecting eq. 6.3.20, it can be seen that the same effect is obtained if the gravitational constant is added to the acceleration remainder for B. Due to the recursive formulation developed later for the acceleration remainder, it is much better to take the latter approach. Thus, the `add-gravity` macro does not actually apply forces to the bodies. Instead, it sets a global constant called `*acceleration-due-to-gravity*` to a vector obtained by multiplying the gravitational constant (nominally the symbol `gees`) by the direction of the field (nominally `[n3])`. When AUTOSIM is initialized, the value of `*acceleration-due-to-gravity*` is set to zero. Thus, gravity is not included in the analysis unless the `add-gravity` macro is invoked.

The `small` macro is used to declare that symbols are small (the *small-order* slot in each argument is set to a value of 1) and the `large` macro declares that symbols are large (the *small-order* slot in each argument is set to a value of –1). Ordinarily, these slots have a default value of 0.

The `add-constraint` macro is used to introduce a constraint equation. The constraints can apply to either coordinates or speeds. The `no-movement` macro applies `add-constraint` twice: once to define a speed constraint, and once to define a coordinate constraint. (A detailed discussion of the `add-constraint` macro is deferred until section 8.3.)

Joint Description for New Bodies

The analyses performed when a body is added deal mainly with the coordinate system of the new body, as determined by the kinematics of the joint connecting it to its parent. In order to automate this process, it is necessary to define the kinematics relating a body to its parent in a way that is meaningful to the analyst.

A *building-block joint* model is used to define the kinematical relation between a new body and its parent. The joint includes between zero and six kinematical degrees of freedom. Three of these are consecutive translations, and the other three are consecutive *simple rotations*. (A simple rotation is one in which two reference frames have one line which is fixed in both frames throughout the rotation. That line is the rotation axis.) This model is not completely generalized, because it requires that the translations occur before the rotations. However, by defining massless intermediate bodies (each with its own building-block joint), almost any joint geometry can be built with this model. The parameters that describe the building-block joint are summarized in Table 8.1.2.

Table 8.1.2. Parameters and degrees of freedom of a body/joint.

Parameter	Description
$\vec{\mathbf{r}}^{A_0B_J}$	position of joint point of B relative to origin of parent.
$(\vec{\mathbf{r}}^B_{t1}, \vec{\mathbf{r}}^B_{t2}, \vec{\mathbf{r}}^B_{t3})$	list of 0, 1, 2, or 3 directions for translational degrees of freedom of B, fixed in the coordinate system of the parent. (In Figure 8.1.1, the single direction is designated $\vec{\mathbf{r}}^B_T$.)
(i_1, i_2, i_3)	list of 0, 1, or 3 axis indices in B for sequential rotations.
$\vec{\mathbf{r}}^B_{rot}$	orientation of first rotation axis of B (fixed in the coordinate system of the parent).
$\vec{\mathbf{r}}^B_{ref}$	reference direction for first rotation of B (fixed in the coordinate system of the parent).
$(\vec{\mathbf{r}}^B_{r1}, \vec{\mathbf{r}}^B_{r2}, \vec{\mathbf{r}}^B_{r3})$	list of 0, 1, or 3 directions of rotations for B. This list is derived from the above parameters.

The geometry is illustrated in Figure 8.1.1 for an example involving one degree of freedom for rotation and one for translation.

The three directions of the coordinate system of B are the unit-vectors $\vec{\mathbf{b}}_1$, $\vec{\mathbf{b}}_2$, and $\vec{\mathbf{b}}_3$. For the parent A, the three directions are $\vec{\mathbf{a}}_1$, $\vec{\mathbf{a}}_2$, and $\vec{\mathbf{a}}_3$. The origin for B is the point

designated B_0. In this figure, the magnitude of the translation is the generalized coordinate q_i and the magnitude of the rotation is the generalized coordinate q_{i+1}.

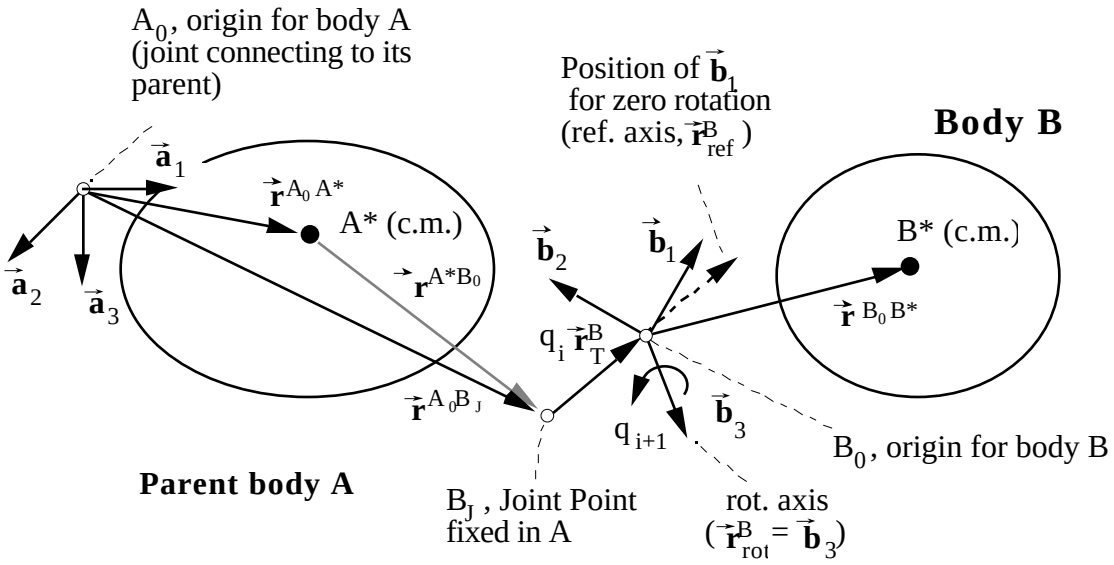

Figure 8.1.1. Geometry of body relative to its parent.

The relationship between the coordinate systems of B and A depends on the type and number of degrees of freedom:

- If the joint has one or more translational degrees of freedom, B_0 can move within the coordinate system of A. Otherwise, it is a point fixed in A.
- If the joint has one or more rotational degrees of freedom, at least two of the unit-vectors of B differ from those of A. Otherwise, both bodies have coordinate systems based on the same directions.

One generalized coordinate is introduced for each degree of freedom of the joint. The description of the joint kinematics can be separated into translational and rotational displacements.

Translational Displacement

The translational coordinates define how the origin for the new body is positioned relative to the origin of the parent. Slots in a `body` that pertain to the translational displacement of the joint are shown in Table 8.1.3.

Two of the slots are assigned to `point` objects. One is the point B_j fixed in the parent body (in slot *joint-point*), and the other is the origin of the new coordinate system, B_0 (in

Table 8.1.3. `Body` slots related to joint translational displacement.

Slot Name	Type	Definition
0-point	`point`	origin of coordinate system (also, joint attachment point in this body).
joint-point	`point`	joint attachment in parent body.
translation-coordinates	`list`	translational generalized coordinates introduced for this body.
translation-directions	`list`	directions corresponding to variables in *translation-coordinates.*
small-translations	`list`	Booleans corresponding to variables in *translation-coordinates*. `T` if variable is small, `NIL` otherwise.

slot *0-point*). The point B_j is specified by the analyst using coordinates in an existing coordinate system. (Regardless of the coordinate system used to specify the coordinates, the `point` is fixed in the parent.) The `point` object is put in the *joint-point* slot of the `body` to define the position of the origin in the nominal state when all generalized coordinates are zero. A `point` is also created for the new body to define the origin of its coordinate system. Because the coordinates of an origin are defined as (0 0 0), no input from the analyst is needed to create the origin `point`.

If the joint has translational degrees of freedom, a list of the translational directions is needed. These direction vectors, ($\vec{\mathbf{r}}^B_{t1}$, $\vec{\mathbf{r}}^B_{t2}$, $\vec{\mathbf{r}}^B_{t3}$), are each specified by the analyst in the same coordinate system as was used to define the *joint-point*. The directions are converted to the coordinate system of the parent, and multiplied by the `uv`s (unit-vectors) from the parent to create direction vectors. The list of vectors is then put into the *translation-directions* slot of the `body`.

The number of translational degrees of freedom is determined by the length of the list of directions. Translational coordinates are introduced by creating a list of `indexed-sym` objects, which is then put into the *translation-coordinates* slot. For each degree of freedom, two `indexed-sym`s are created at this time: one for a generalized coordinate (e.g., q_3) and one for its derivative (e.g., $\dot{q}_3$). The printed representation of each `indexed-sym` object is determined by the symbol put into its *symbol* slot and the number put into its *i* slot. (E.g., a variable with "Q" in its *symbol* slot and "3" in its *i* slot prints as "Q(3)." With the symbol "QP" in the *symbol* slot it prints as "QP(3)".) Slots in the

individual `indexed-sym`s that identify the objects as variables are also set. For example, (1) the *dxdt* slot of the coordinate is set to the `indexed-sym` made to represent its derivative, (2) the *const-or-var* slots of of both the coordinate and its derivative are set to the symbol `var`, (3) if the analyst specified that the translation is "small," the *small-order* slots of the coordinate and its derivative are set to 1 (otherwise the order of smallness is 0), (4) the units of the coordinate are set to the expression L (units of length) and the units of its derivative are set to L/T (length per unit time), and (5) a name is created, based on the names of the new body and the parent body and the direction of the translation. (Examples of how state variables are named by AUTOSIM appear in the next chapter.)

The list of orders of smallness specified by the analyst is put into the slot *small-translations.*

The position of point B_0 relative to point A_0 is the vector

$$\vec{\mathbf{r}}^{A_0B_0} = \vec{\mathbf{r}}^{A_0B_J} + \sum_{i=1}^{N_{td}^B} q_{i+o}\, \vec{\mathbf{r}}_{ti}^{B} \tag{8.1.1}$$

where N_{td}^B is the number of translational degrees of freedom for body B and o is an offset constant that maps the index i from the summation onto the indices of the generalized coordinates. In Figure 8.1.1, N_{td}^B is 1 and the position vector $\vec{\mathbf{r}}^{A_0B_0}$ is $\vec{\mathbf{r}}^{A_0B_J} + q_i\, \vec{\mathbf{r}}_T^B$.

Rotational Displacement

Slots in a `body` that pertain to the joint rotation are shown in Table 8.1.4.

Recall that the building-bock joint model assumes consecutive simple rotations, in which each rotation occurs about an axis fixed in B. The sequence of axes is provided by the analyst as a list of integer numbers and put by the `add-body` macro into the *rotation-axes* slot of the `body`. Two pieces of information are required in addition to the list, to specify the orientation of B relative to A when all generalized coordinates are zero. First, the orientation of the first rotation axis, $\vec{\mathbf{r}}_{rot}^B$, is in a direction fixed in the coordinate system of the parent. $\vec{\mathbf{r}}_{rot}^B$ is not always parallel with any of the axes of the coordinate system in the parent, and can be entered as a set of constant coordinates in a selected coordinate system. Those coordinates are converted to the coordinate system of the parent, and are stored in the *parent-rotation-axis* slot of the `body.` In Figure 8.1.1, the rotation axis coincides with $\vec{\mathbf{b}}_3$. Thus, B can rotate relative to A about an axis that coincides with a direction vector aligned with axis 3 in B. That same vector is described with a set of three constant coordinates in the coordinate system of A.

Table 8.1.4. `Body` slots related to joint rotation.

Slot Name	Type	Definition
parent-rotation-axis	`array`	coordinates of rotation axis in coordinate system of parent.
rotation-axes	`list`	list of axes in new body about which consecutive rotations take place.
reference-axis	`array`	coordinates of reference axis in coordinate system of parent.
rotation-coordinates	`list`	rotational generalized coordinates introduced for this body.
small-angles	`list`	Booleans corresponding to variables in *rotation-coordinates*: T if variable is small, NIL otherwise.

The third and last piece of information related to the orientation of B defines the orientation of B when all generalized coordinates are zero. This nominal orientation is defined with a vector, $\vec{\mathbf{r}}^{B}_{ref}$, called the reference axis. The rotation and reference axes are orthogonal. (If the two sets of coordinates provided by the analyst are not orthogonal, the component of the direction provided by the analyst that is orthogonal to the rotation axis is derived and used as the reference axis.) The reference axis is shown by a dashed line in the figure. In the nominal orientation, the three axes of the coordinate system of B are aligned with (1) the rotation axis, (2) the reference axis, and (3) their cross-product. The reference axis is specified by the analyst using coordinates in a selected coordinates system. Those coordinates are converted to obtain coordinates in the coordinate system of the parent. The converted coordinates are kept in the *reference-axis* slot of B.

The appropriate axis of B is aligned with the reference axis of the parent, where the "right-handed" axis to use as a reference is provided in Table 8.1.5.

Table 8.1.5. Right-handed axis convention

Rotation Axis	Reference Axis
1	2
2	3
3	1

A generalized coordinate is introduced for each rotation associated with the joint. In the figure, this variable is designated q_{i+1}. As was the case for translational variables, a new `indexed-sym` object is created for each rotational variable and another for its derivative. Slots in the `indexed-sym` are set to identify them as variables. The list of `indexed-sym`s is put into the *rotation-coordinates* list.

When the joint has three rotational degrees of freedom, a list of indices is provided to specify the sequence of rotations. For example, the list (3 2 1) has the following meaning: "Body B is initially aligned such that $\vec{\mathbf{b}}_3$ is aligned with $\vec{\mathbf{r}}^{B}_{rot}$ and $\vec{\mathbf{b}}_1$ is aligned with $\vec{\mathbf{r}}^{B}_{ref}$. B then rotates about $\vec{\mathbf{b}}_3$ by an angle q_{o+1}, where o is an index offset. From that new position, it is rotated about $\vec{\mathbf{b}}_2$ by an angle q_{o+2}. Finally, it is rotated about $\vec{\mathbf{b}}_1$ by an angle q_{o+3}. After the first rotation, the orientation is an intermediate frame, designated B" After the second rotation, the orientation is another intermediate frame, designated B'. These intermediate frames are not created as `body` objects, and cannot be referenced by any of the AUTOSIM algebra functions. If an intermediate frame is needed to develop moments or to define angles, then instead of specifying one body with 3 rotations, 3 bodies should be entered, each with one rotation. (The first two should be given zero mass and inertia values.)

In order to simplify some of the rules that follow, the building-block joint model allows zero, one, or three consecutive rotations between a body and its parent, but not two rotations. Joints which involve two consecutive rotations are represented by two building-block joints, where the first is associated with a massless body.

This representation is valid for mechanical joints that are commonly available in multibody analysis programs. Several simple joints are represented in Table 8.1.6. Other types of joints (cables, gears, cams, etc.) are described with combinations of building-block joints and constraint equations.

Table 8.1.6. Representation of simple joints with "building-block" model.

Joint Type	Translational d.o.f.	Rotational d.o.f.	No. of "building-blocks"
Prismatic	1	0	1
Revolute	0	1	1
Hooke, Gimbal	0	2	2
Planar slider	2	0	1
Ball Joint	0	3	1
Cylindrical	1	1	1
Free	3	3	1

Direction Transformations

Slots in the `body` that define direction transformations are summarized in Table 8.1.7.

Each body in the system has its own coordinate system, with an origin and three axes. The directions of the three axes are defined by unit-vectors. A direction cosine matrix is used to relate the three unit-vectors of a body with the unit-vectors of the parent. The directions of the axes of B are related to those of A by the direction cosine matrix ${}^{B}\underline{C}^{A}$, defined such that

Table 8.1.7. `Body` slots related to direction transformations.

Slot Name	Type	Definition
uvs	`array`	trio of `uv`s that defines the axis directions of the coordinate system of this body, e.g., $(\vec{\mathbf{b}}_1\ \vec{\mathbf{b}}_2\ \vec{\mathbf{b}}_3)$.
cos-matrix	`array`	direction cosine matrix relating the unit-vectors of this body to those of its parent.
basis	`dyadic`	a dyadic that transforms an arbitrary vector expression into the basis of this body, e.g., $\vec{\mathbf{b}}_1\ \vec{\mathbf{b}}_1 + \vec{\mathbf{b}}_2\ \vec{\mathbf{b}}_2 + \vec{\mathbf{b}}_3\ \vec{\mathbf{b}}_3$.
rotation-directions	`list`	list of rotation directions associated with the generalized coordinates in the *rotation-coordinates* slot.

$$\begin{Bmatrix} \vec{\mathbf{b}}_1 \\ \vec{\mathbf{b}}_2 \\ \vec{\mathbf{b}}_3 \end{Bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} \begin{Bmatrix} \vec{\mathbf{a}}_1 \\ \vec{\mathbf{a}}_2 \\ \vec{\mathbf{a}}_3 \end{Bmatrix} \tag{8.1.2}$$

Or,

$$C_{ij} = \vec{\mathbf{b}}_i \bullet \vec{\mathbf{a}}_j \tag{8.1.3}$$

For the building-block joint model just presented, there can be 0, 1, or 3 rotational degrees of freedom. The direction cosine matrix for each of these cases is described below.

Recall that the dot-product operation is performed by using the direction cosine matrix. Thus, the definition of eq. 8.1.3. cannot be used to define the direction cosine matrix, because at the time the matrix is being created, the dot-product operation will not work. Instead, the matrix is constructed by using the rotational information introduced above and stored in the slots *parent-rotation-axis, reference-axis,* and *rotation-axes.*

Bodies with Zero Rotational degrees of Freedom

The axes for the coordinate system of a body with zero rotational degrees of freedom are defined to be parallel to those of the parent body. That is, the three unit-vectors associated with the body are the same as those of the parent: $\vec{\mathbf{a}}_1$, $\vec{\mathbf{a}}_2$, and $\vec{\mathbf{a}}_3$. The direction cosine matrix implied by eq. 8.1.3 is a 3-by-3 identity matrix:

$$\underline{C}^{BA} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8.1.4}$$

In the computer representation, the contents of the *basis* and *uvs* slots of the parent `body` are copied into the corresponding slot of the new body, and the *cos-matrix* slot is set to the above identity matrix. The *rotation-directions* slot remains NIL.

Bodies with One Rotational degree of Freedom

If body B has one rotational degree of freedom with respect to A, it rotates about an axis whose direction is fixed in both B and A. Recall that the rotation axis $\vec{\mathbf{r}}^B_{rot}$ and the reference axis $\vec{\mathbf{r}}^B_{ref}$ are both vectors which are not parallel, but are not necessarily orthogonal as defined by the analyst. Two cross-product operations are used to define two unit-

vectors that are combined with $\vec{\mathbf{r}}^{B}_{rot}$ to define three orthogonal unit-vectors for the coordinate system of B:

$$\vec{\mathbf{b}}_j = \vec{\mathbf{r}}^{B}_{rot} \times \vec{\mathbf{r}}^{B}_{ref} \tag{8.1.5}$$

$$\vec{\mathbf{b}}_i = \vec{\mathbf{b}}_j \times \vec{\mathbf{r}}^{B}_{rot} \tag{8.1.6}$$

$$\vec{\mathbf{b}}_k = \vec{\mathbf{r}}^{B}_{rot} \tag{8.1.7}$$

The set of unit-vectors introduced for B are nominally designated $\vec{\mathbf{b}}_1$, $\vec{\mathbf{b}}_2$, and $\vec{\mathbf{b}}_3$, and are identical to the unit-vectors $\vec{\mathbf{b}}_i$, $\vec{\mathbf{b}}_j$, and $\vec{\mathbf{b}}_k$, where the definitions of the indices i, j, and k are obtained from Table 8.1.8.

Table 8.1.8. Indices for three possible rotation axes.

Case	i	j	k
$\vec{\mathbf{r}}^{B}_{rot} = \vec{\mathbf{b}}_1$	2	3	1
$\vec{\mathbf{r}}^{B}_{rot} = \vec{\mathbf{b}}_2$	3	1	2
$\vec{\mathbf{r}}^{B}_{rot} = \vec{\mathbf{b}}_3$	1	2	3

First, the orientation of B relative to A must be determined for the nominal condition when all generalized coordinates are zero. These are defined by dot products between $\vec{\mathbf{b}}_i$, $\vec{\mathbf{b}}_j$, and $\vec{\mathbf{b}}_k$ and $\vec{\mathbf{a}}_1$, $\vec{\mathbf{a}}_2$, and $\vec{\mathbf{a}}_3$. Because the rotation axis and reference axis are stored in the `body` as coordinates in the coordinate system of A, the dot products are simply those coordinates. Calling the rotation angle θ, two terms, s and c, are introduced as the sine and cosine of θ to account for the rotation. The creator functions `make-sin` and `make-cos` are used, so that small angle approximations are made appropriately. Each `trig` object is created just once, so all references to that `trig` object later involve a single object in the computer.

The elements of the direction cosine matrix are defined for each row using the same i, j, and k indices assigned in Table 8.1.8:

$$C_{ir} = c\,(\vec{\mathbf{a}}_r \bullet \vec{\mathbf{b}}_i) + s\,(\vec{\mathbf{a}}_r \bullet \vec{\mathbf{b}}_j) \qquad (r = 1,2,3) \tag{8.1.8}$$

$$C_{jr} = -s\,(\vec{\mathbf{a}}_r \bullet \vec{\mathbf{b}}_i) + c\,(\vec{\mathbf{a}}_r \bullet \vec{\mathbf{b}}_j) \qquad (r = 1,2,3) \tag{8.1.9}$$

$$C_{kr} = \vec{\mathbf{a}}_r \bullet \vec{\mathbf{b}}_k \qquad (r = 1,2,3) \tag{8.1.10}$$

The resulting array is placed in the *cos-matrix* slot of the `body`.

If the rotation axis is parallel to one of the unit-vectors of A, then the corresponding uv unit-vector is also used for B. For example, suppose the rotation axis is described in A as $-\vec{\mathbf{a}}_2$, and the coordinate system for B is defined such that rotation of B relative to A occurs about axis number 2. Then, the unit-vectors of B are $\vec{\mathbf{b}}_1$, $-\vec{\mathbf{a}}_2$, and $\vec{\mathbf{b}}_3$. On this occasion, one of the rows of the matrix ${}^{B}\underline{C}^{A}$ contains two zeros and a minus-one.

The rotation axis is not always parallel with a unit-vector of the parent. For example, if the rotation axis is specified as $\frac{\vec{\mathbf{a}}_2 + \vec{\mathbf{a}}_3}{\sqrt{2}}$, then three new unit-vectors are introduced for B. In this case, all three uvs in the *uvs* slot of B are new.

The *rotation-directions* slot is set to a list with one element: the uv aligned with the rotation axis.

Bodies with Three Rotational Degrees of Freedom

A body B 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 described above 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. That is,

$$ {}^{B}\underline{C}^{A} = {}^{B}\underline{C}^{B'}\,{}^{B'}\underline{C}^{B''}\,{}^{B''}\underline{C}^{A} \tag{8.1.11} $$

In addition to the direction cosine matrix, the three rotation axes ($\vec{\mathbf{r}}^{B}_{r1}$, $\vec{\mathbf{r}}^{B}_{r2}$, and $\vec{\mathbf{r}}^{B}_{r3}$) are required for some of the following analyses. Unit-vectors are not introduced for intermediate frames B' and B", and therefore these three axes must be represented using unit-vectors for the coordinate systems of B and its parent, A. The first, $\vec{\mathbf{r}}^{B}_{r1}$, is fixed in A and was stored as $\vec{\mathbf{r}}^{B}_{rot}$ using the coordinate system of the parent. The third, $\vec{\mathbf{r}}^{B}_{r3}$, is common to B' and B, and is the unit-vector associated with the third index in the body axis rotation list. The second, $\vec{\mathbf{r}}^{B}_{r2}$, is common to the intermediate frames B" and B'. It can be written in terms of the unit-vectors of B using a column of the cosine matrix ${}^{B}\underline{C}^{B'}$:

$$ \vec{\mathbf{r}}^{B}_{r2} = C_{1j}\vec{\mathbf{b}}_1 + C_{2j}\vec{\mathbf{b}}_2 + C_{3j}\vec{\mathbf{b}}_3 \tag{8.1.12} $$

where j is the index of the second rotation axis, and C_{1j}, C_{2j}, and C_{3j} are coefficients of ${}^{B}\underline{C}^{B'}$.

The list ($\vec{\mathbf{r}}^{B}_{r1}$, $\vec{\mathbf{r}}^{B}_{r2}$, and $\vec{\mathbf{r}}^{B}_{r3}$) is put into the *rotation-directions* slot of the body.

Recursive/Nonrecursive Descriptions

Nonrecursive formulations are used when there are enough degrees of freedom in the joint of a body such that its motions can be described without reference to other bodies in the system. When this is not possible, recursive formulations are used. Two slots in the `body` object store the types of analyses that are used for rotation and translation, as indicated in Table 8.1.9.

Table 8.1.9. `Body` slots related to recursion.

Slot Name	Type	Definition
recursive-r	`symbol`	formulation to use for rotation analysis.
recursive-t	`symbol`	formulation to use for translation analysis.

The *recursive-r* slot is assigned to the symbol `NIL` if a nonrecursive analysis is to be used to obtain expressions for rotational velocity and acceleration. A very simple rule is used: if the body has three rotational degrees of freedom, the nonrecursive analysis is used. Otherwise, the body is recursive. There are two variations of the recursive formulation used, indicated by setting the slot to either the symbol `t` or `rotor`. The criterion is based on the inertial properties of the body, as described in the next subsection.

The *recursive-t* slot is assigned to the symbol `NIL` if a nonrecursive analysis is to be used to obtain expressions for translational velocity and acceleration. There are two conditions in which the nonrecursive formulations are used: (1) bodies with three translational degrees of freedom, and (2) bodies with two translational degrees of freedom that are constrained to planar motions. The first case applies when the list of coordinates in the *translation-coordinates* slot has three elements. The second applies when three conditions are satisfied. First, the list of coordinates in the *translation-coordinates* slot has two elements. Second and third, the following two tests must be true:

$$\vec{\omega}^{B} \times \left(\vec{\mathbf{r}}^{B}_{t1} \times \vec{\mathbf{r}}^{B}_{t2}\right) \stackrel{?}{=} 0$$
$$\textit{and}$$
$$\vec{\mathbf{v}}^{B_J} \bullet \left(\vec{\mathbf{r}}^{B}_{t1} \times \vec{\mathbf{r}}^{B}_{t2}\right) \stackrel{?}{=} 0 \qquad (8.1.13)$$

where $\vec{\mathbf{r}}^{B}_{t1}$ and $\vec{\mathbf{r}}^{B}_{t2}$ are the two directions of the translational degrees of freedom. Otherwise, the body is recursive. There are two variations of the recursive formulation used, indicated

by setting the slot to either the symbol `t` or `fixed`. The criterion for setting the slot is based on the inertial properties of the body, as described in the next subsection.

Inertia Properties

Slots in the `body` object pertaining to inertia properties are summarized in Table 8.1.10. The table includes data provided by the analyst (as optional arguments for the `add-body` macro) for the isolated rigid body element. For the purpose of performing the dynamics analysis described in Section 8.4, the inertia properties of each body are summarized in three slots: (1) the scalar mass (a scalar expression associated with the *mass* slot), (2) the mass center, (a `point` associated with the *cm-point* slot), and (3) the inertia dyadic (an expression associated with the *inertia* slot). In certain conditions, these three inertia properties represent composite bodies, obtained by combining attributes of adjacent bodies.

Table 8.1.10. `Body` slots related to inertia.

Slot Name	Type	Definition
cm-coordinates	`3x1 array`	coordinates of rigid-body mass center.
cm-point	`point`	mass center of composite body.
inertia	`expression`	inertia dyadic of composite body.
inertia-matrix	`3x3 array`	inertia matrix for rigid body.
mass	`expression`	mass of composite body.
massb	`expression`	mass of rigid body.

Data specific to the rigid body B are kept in the slots *cm-coordinates*, *inertia-matrix*, and *massb*. The coordinates of the mass center, provided by the analyst, are converted into the coordinate system of B and stored in the *cm-coordinates* slot. The inertia matrix for B is kept in the *inertia-matrix* slot, and the mass of B is kept in the *massb* slot.

As each body is entered, an analysis is performed to set the inertia properties of the new body and all bodies “up” the tree. The procedure, initiated when body B is added, goes as follows:

1. A list is made of all children of B whose *recursive-t* slot is set to the symbol `fixed`. This list is called the *fixed children*. (When applied to a body just added,

there are no children and this is a null list. However, the procedure is also used for other bodies which do have children.)

2. The masses from the *massb* slot of B and the *mass* slots of the fixed children are summed to form the composite mass of B, which is assigned to the *mass* slot of B. That is,

$$m^{Bc} = m^{B} + \sum_{b}^{\text{fixed children}} m^{bc} \tag{8.1.14}$$

where m^{Bc} is the composite mass for B, and the sum covers the fixed children, with the index b indicating each body that is a member of the list of fixed children of B. In this summation (and all of the summations that follow in this procedure), the mass used for bodies in the list of fixed children is the composite mass, from the *mass* slot, as indicates with the superscript "bc." However, for body B, the rigid-body mass from the *massb* slot is used.

3. The coordinates of the composite mass are computed:

$$x_i^{Bc*} = \frac{x_i^{B*}\, m^{B} + \sum_{b}^{\text{fixed children}} x_i^{bc*}\, m^{bc}}{m^{Bc}} \quad (i = 1,2,3) \tag{8.1.15}$$

where x_i^{B*} is one of the three coordinates of the mass center of B and x_i^{bc*} is the corresponding coordinate of the composite mass of a fixed child of B, which has been properly converted to the coordinate system of B via the `convert-coordinates` function. The three coordinates of the composite mass center are used to create a new `point` object, which is placed in the *cm-point* slot of B.

4. An inertia matrix for the composite body is constructed using the parallel axes theorem, considering the masses of the fixed children (but not the inertia dyadics):

$$I_{11}^{Bc*} = I_{11}^{B*} + \sum_{b}^{\text{B, fixed children}} m^{b}\left[\left(x_2^{b*} - x_2^{Bc*}\right)^2 + \left(x_3^{b*} - x_3^{Bc*}\right)^2\right]$$

$$I_{22}^{Bc*} = I_{22}^{B*} + \sum_{b}^{\text{B, fixed children}} m^{b}\left[\left(x_1^{b*} - x_1^{Bc*}\right)^2 + \left(x_3^{b*} - x_3^{Bc*}\right)^2\right]$$

$$I_{33}^{Bc*} = I_{33}^{B*} + \sum_{b}^{\substack{B,\ fixed\\ children}} m^b \left[\left(x_1^{b*} - x_1^{Bc*} \right)^2 + \left(x_2^{b*} - x_2^{Bc*} \right)^2 \right]$$

$$I_{12}^{Bc*} = I_{21}^{Bc*} = I_{12}^{B*} - \sum_{b}^{\substack{B,\ fixed\\ children}} m^b \left[\left(x_1^{b*} - x_1^{Bc*} \right) \left(x_2^{b*} - x_2^{Bc*} \right) \right]$$

$$I_{13}^{Bc*} = I_{31}^{Bc*} = I_{13}^{B*} - \sum_{b}^{\substack{B,\ fixed\\ children}} m^b \left[\left(x_1^{b*} - x_1^{Bc*} \right) \left(x_3^{b*} - x_3^{Bc*} \right) \right]$$

$$I_{23}^{Bc*} = I_{32}^{Bc*} = I_{23}^{B*} - \sum_{b}^{\substack{B,\ fixed\\ children}} m^b \left[\left(x_2^{b*} - x_2^{Bc*} \right) \left(x_3^{b*} - x_3^{Bc*} \right) \right] \qquad (8.1.16)$$

In the above equation set, the summations cover the fixed children of B and also the rigid body B. For body B, the mass used for m^b is the rigid-body mass (from the *massb* slot). For the children of B, the mass m^b is the composite mass (from the *mass* slot)

5. The inertia matrix constructed in step 4 is made into a dyadic using the unit-vectors of B:

$$\overleftrightarrow{\mathbf{I}}^{Bc*} = \sum_{i=1}^{3} \sum_{j=1}^{3} \left(I_{ij}^{Bc*} \vec{\mathbf{b}}_i \vec{\mathbf{b}}_j \right) \qquad (8.1.17)$$

The dyadic is put in the *inertia* slot of the `body`.

6. The rotational category of B is determined. The inertia matrix from step 4 is used to determine if B is a rotor. It is a rotor if all three of the following conditions are satisfied: (a) the body uses the recursive formulation (that is, the *recursive-r* slot is not `NIL`), (b) the composite inertia matrix is diagonal, and (c) the two moments of inertia perpendicular to the axis of rotation are equal. If B is a rotor, the *recursive-r* slot is set to the symbol `rotor`. Otherwise, the slot is set to `T`. If the slot is set to `rotor`, the inertia dyadic is converted to the basis of the parent using the identity

$$\overleftrightarrow{\mathbf{I}}^{Bc*} = \overleftrightarrow{\mathbf{a}} \bullet \overleftrightarrow{\mathbf{I}}^{Bc*} \bullet \overleftrightarrow{\mathbf{a}} \qquad (8.1.18)$$

Otherwise, the formulation obtained from eq 8.1.17 is kept.

7. The translational category of B is determined. The coordinates of the composite mass from step 3 are used to determine if the mass of B is fixed in the coordinate system of its parent. It is fixed if the *recursive-t* slot is not `NIL` and any one of the following conditions is satisfied: (a) the composite mass is zero, (b) all three coordinates are zero, or (c) the body has no translational degrees of freedom, one rotational degree of freedom, and the only nonzero coordinate of the composite mass center is along the rotation axis. If B is fixed, the *recursive-t* slot is set to the symbol `fixed`. Otherwise, the slot is set `T` or left at `NIL`, depending its original value.

8. Unless the parent of B is the inertial reference (N), the above procedure is repeated for the parent.

The last step in the above procedure means that as each body is added to the tree, the mass and inertia properties of all bodies "up" the tree are subject to adjustment.

Velocities

Velocity information in the `body` objects is used to support algebra functions such as `rot` and `vel`. Slots related to velocity are listed in Table 8.1.11. Lists of generalized speeds corresponding to those created for translational and rotational generalized coordinates are created and put into the *translation-speeds* slot and the *rotation-speeds* slot. Slots in the `indexed-sym` objects are set as was done for the generalized coordinates, except that the *symbol* slots are set to "U" and "UP" instead of "Q" and "QP." (And of course, different names and units are put into the *name* and *units* slots of the `indexed-sym` objects.)

Table 8.1.11. `Body` slots related to velocity.

Slot Name	Type	Definition
abs-v0	`expression`	velocity of origin (point B_0).
abs-w	`expression`	rotational velocity of body.
rotation-speeds	`list`	rotational generalized speeds for this body.
translation-speeds	`list`	translational generalized speeds for this body.
translational-speed-directions	`list`	directions corresponding to variables in *translation-speeds.*

The directions associated with the speeds are not necessarily the same as those associated with the coordinates. Based on the symbols in the *recursive-r* and *recursive-t* slots, speeds for rotation and translation are defined using either a recursive formulation (in which case the speeds are the simple derivatives of the generalized coordinates) or a nonrecursive formulation (in which the speeds are defined in body-fixed directions).

Directions are determined for the translational speeds and put into lists kept in the slots *translational-speed-directions*. The directions are defined as follows:

1. If the *recursive-t* slot is not `NIL`, then the list of speed directions is identical to the list of vectors in the *translation-directions* slot.

2. If the *recursive-t* slot is `NIL` and the body has two translational degrees of freedom, then the body has one rotational degree of freedom. The list of speed directions is obtained by starting with the list of the three unit-vectors in the *uvs* slot of the `body`, and then removing the unit-vector that is parallel with the body rotation axis (that is, the first element of the list in the *body-rotation-axes* slot).

3. Otherwise, the *recursive-t* slot is `NIL` and the body has three translational degrees of freedom. This list of directions is the list of unit-vectors fixed in the body, obtained from the *uvs* slot.

The rules for rotational speeds are so simple that a corresponding list for rotational speed directions is unnecessary.

The angular velocity of the body is determined as follows, based on the number of rotational degrees of freedom (d.o.f.) of the body:

$$\vec{\omega}^B = \begin{cases} \vec{\omega}^A & (0 \text{ d.o.f.}) \\ \vec{\omega}^A + u_r \vec{\mathbf{r}}^B_{rot} & (1 \text{ d.o.f.}) \\ u_{o+1} \vec{\mathbf{b}}_1 + u_{o+2} \vec{\mathbf{b}}_2 + u_{o+3} \vec{\mathbf{b}}_2 & (3 \text{ d.o.f.}) \end{cases} \tag{8.1.19}$$

(For the case of 1 d.o.f., u_r is the generalized speed introduced for the rotational degree of freedom. For the case of 3 d.o.f., o is an offset such that the three speeds introduced for the rotational degrees of freedom are u_{o+1}, u_{o+2}, and u_{o+3}.) The angular velocity is put into the slot *abs-w*. This expression is not converted to any one basis, and can include unit-vectors from many different bodies.

The translational velocity of the origin is determined as follows, based on whether or not the body is recursive in translation. (It is recursive if the *recursive-t* slot is set to `t` or `fixed`, and nonrecursive if the slot is set to `nil`.)

$$\vec{\mathbf{v}}^{B_0} = \begin{cases} \vec{\mathbf{v}}^{A_0} + \vec{\omega}^{A} \times \vec{\mathbf{r}}^{A_0B_0} + \sum_{i=1}^{N_{td}^B} u_{i+o}\, \vec{\mathbf{r}}_{ti}^{B} & \text{(recursive)} \\ \sum_{i=1}^{N_{td}^B} u_{i+o}\, \vec{\mathbf{r}}_{tvi}^{B} - \vec{\omega}^{B} \times \vec{\mathbf{r}}^{B_0B^*} & \text{(nonrecursive)} \end{cases} \tag{8.1.20}$$

(The index o is an offset such that the speeds introduced to account for the translational degrees of freedom are o+1, … o+N_{td}^B. For the nonrecursive case, the symbol $\vec{\mathbf{r}}_{tvi}^{B}$ designates the directions of the translational speeds. The rationale for setting those directions is presented in Section 8.4.) This expression is not converted to any one basis, and can include unit-vectors from many different bodies.

With the velocity information kept in the `body` objects, functions such as `rot` and `vel` are trivial to implement. The function `(rot b)` simply returns the expression from the *abs-w* slot of the body b. The function `(vel p)` returns the expression

$$\vec{\mathbf{v}}^{P} = \vec{\mathbf{v}}^{B_0} + \vec{\omega}^{B} \times \vec{\mathbf{r}}^{B_0P} \tag{8.1.21}$$

where B is the body containing the point P and $\vec{\mathbf{r}}^{B_0P}$ is the position vector going from the origin of B to P.

The nonrecursive part of eq. 8.1.20 involves the position of the mass center of B. However, the mass center is subject to change, depending on whether bodies are added which have mass centers fixed in the coordinate system of B. Accordingly, eq. 8.1.21 is reapplied to all bodies in the system whenever a new body is added.

8.2. Kinematical Analysis

The kinematical equations are *n* ordinary differential equations that relate the derivatives of the generalized coordinates to known speeds. This set of equations is written below in matrix form:

$$\underline{S}\, \dot{\underline{q}} = \underline{v} \tag{8.2.1}$$

Where $\underline{S}$ is an $n \times n$ matrix, $\dot{\underline{q}}$ is a column array of length *n* containing the derivatives of the generalized coordinates, and $\underline{v}$ is a column array of length *n* containing the known speeds. Each row i in the arrays of eq. 8.2.1 is an equation developed by considering the

generalized coordinate q_i. Depending on whether q_i is a rotational or translational coordinate, different methods are employed.

Rotational Speeds

Each body has one generalized coordinate introduced for each rotational degree of freedom associated with the body. The angular velocity of the body, relative to its parent, can be written in terms of the derivatives of the generalized (rotational) coordinates introduced for the body:

$$ {}^{A}\vec{\omega}^{B} = \sum_{j=1}^{N_{rd}^{B}} \dot{q}_{o+j}\, \vec{\mathbf{r}}_{rj}^{B} \tag{8.2.2} $$

where N_{rd}^{B} is the number of rotational degrees of freedom introduced for B, $\dot{q}_{o+j}$ is the derivative of the generalized coordinate introduced for the j^{th} rotation of the body (o is an index offset), and $\vec{\mathbf{r}}_{rj}^{B}$ is the axis of rotation associated with q_{o+j}.

Recall that for angular velocity,

$$ {}^{A}\vec{\omega}^{B} = \vec{\omega}^{B} - \vec{\omega}^{A} \tag{8.2.3} $$

where $\vec{\omega}^{B}$ and $\vec{\omega}^{A}$ are obtained using the `rot` function.

A kinematical equation is obtained by equating eqs. 8.2.2 and 8.2.3, and dot-multiplying both sides by a suitable vector. A particularly well-suited vector for performing the dot product is $\vec{\mathbf{r}}_{rj}^{B}$ because it forces the diagonal elements of $\underline{S}$ to be unity for the rows corresponding to rotational variables. For the i^{th} generalized coordinate, q_i, introduced for a rotational degree of freedom, the kinematical equation is

$$ \sum_{j=1}^{N_{rd}^{B}} \dot{q}_{o+j}\, \vec{\mathbf{r}}_{rj}^{B} \bullet \vec{\mathbf{r}}_{r(i-o)}^{B} = \left(\vec{\omega}^{B} - \vec{\omega}^{A}\right) \bullet \vec{\mathbf{r}}_{r(i-o)}^{B} \tag{8.2.4} $$

In terms of the matrix equation, the elements of $\underline{S}$ for row i are

$$ S_{ij} = \begin{cases} \vec{\mathbf{r}}_{r(j-o)}^{B} \bullet \vec{\mathbf{r}}_{r(i-o)}^{B} & \text{for } j = o+1, \ldots o+N_{rd}^{B} \\ 0 & \text{for all other } j \end{cases} \tag{8.2.5} $$

And the i^{th} element in $\underline{v}$ is

$$v_i = \left(\vec{\omega}^B - \vec{\omega}^A\right) \bullet \vec{\mathbf{r}}^B_{r(i-o)} \tag{8.2.6}$$

Translational Speeds

Each body has one generalized coordinate associated with each translational degree of freedom of the body. The velocity of the body origin, B_0, relative to the coordinate system of A, can be written in terms of the derivatives of the generalized (translational) coordinates introduced for the body:

$$^A\vec{\mathbf{v}}^{B_0} = \sum_{j=1}^{N^B_{td}} \dot{q}_{o+j}\, \vec{\mathbf{r}}^B_{tj} \tag{8.2.7}$$

where $^A\vec{\mathbf{v}}^{B_0}$ is the velocity of B_0 with respect to a coordinate system fixed in A, N^B_{td} is the number of translational degrees of freedom introduced for B, $\dot{q}_{o+j}$ is the derivative of the generalized coordinate introduced for the j^{th} translation of the body (o is an index offset), and $\vec{\mathbf{r}}^B_{tj}$ is the direction of the translation associated with q_{o+j}.

The absolute velocity of B_0, obtained with the function `vel`, can be written as:

$$\vec{\mathbf{v}}^{B_0} = \vec{\mathbf{v}}^{A_0} + {}^A\vec{\mathbf{v}}^{B_0} + \vec{\omega}^A \times \vec{\mathbf{r}}^{A_0B_0} \tag{8.2.8}$$

Rearranging, a second expression for $^A\vec{\mathbf{v}}^{B_0}$ is obtained:

$${}^A\vec{\mathbf{v}}^{B_0} = \vec{\mathbf{v}}^{B_0} - \vec{\mathbf{v}}^{A_0} - \vec{\omega}^A \times \vec{\mathbf{r}}^{A_0B_0} \tag{8.2.9}$$

A kinematical equation is obtained for $\dot{q}_i$ by equating eqs. 8.2.7 and 8.2.9, and dot-multiplying both sides by $\vec{\mathbf{r}}^B_{t(i-o)}$:

$$\sum_{j=1}^{N^B_{td}} \dot{q}_{o+j}\, \vec{\mathbf{r}}^B_{tj} \bullet \vec{\mathbf{r}}^B_{t(i-o)} = \left(\vec{\mathbf{v}}^{B_0} - \vec{\mathbf{v}}^{A_0} - \vec{\omega}^A \times \vec{\mathbf{r}}^{A_0B_0}\right) \bullet \vec{\mathbf{r}}^B_{t(i-o)} \tag{8.2.10}$$

In terms of the matrix equation, the elements of $\underline{S}$ for row i are

$$S_{ij} = \begin{cases} \vec{\mathbf{r}}^B_{t(j-o)} \bullet \vec{\mathbf{r}}^B_{t(i-o)} & \text{for } j = o+1, \ldots o+N^B_{td} \\ 0 & \text{for all other } j \end{cases} \tag{8.2.11}$$

And the i^{th} element in $\underline{v}$ is

$$v_i = \left(\vec{\mathbf{v}}^{B_0} - \vec{\mathbf{v}}^{A_0} - \vec{\omega}^A \times \vec{\mathbf{r}}^{A_0B_0}\right) \bullet \vec{\mathbf{r}}^B_{t(i-o)} \tag{8.2.12}$$

This formulation guarantees that the diagonal elements of $\underline{S}$ corresponding to translational coordinates are unity. Because the same was true for the rotational coordinates, it follows that all diagonal elements of $\underline{S}$ are 1. Recall that a condition for the uncoupling method presented in Chapter 7 was that the diagonal elements be non-zero. The formulation here meets that condition, and also guarantees that, at most, only two off-diagonal terms in each row are non-zero. (It has been found that the permutation technique developed in section 7.2 offers no improvement, because $\underline{S}$ is is not subject to matrix fill during the LUD analysis.)

The kinematical equations are derived after any constraint equations are added by the analyst, and before any other analyses are performed on the system. The equations are inspected and any references to nonholonomic speeds that were removed by a constraint (as described in the next section) are "expanded" recursively, replacing the `indexed-sym` of the nonholonomic speed with the expression from its *exp* slot. Thus, the kinematical equations in the Fortran code include only parameters, generalized coordinates, and independent speeds.

In the Fortran simulation code, the kinematical equations cause values to be computed for the derivatives of the generalized coordinates. Because they are now "defined," expressions in following Fortran code can refer to these derivatives.

8.3. Constraint Analysis

Most of the constraints in the multibody system are accounted for in the joint characterizations. Rather than starting with a fully unconstrained system and adding constraints, we start with a fully constrained system and add degrees of freedom. For holonomic systems that have a tree topology, no further constraint analysis is required. However, if the system is subject to nonholonomic constraints (e.g., the examples in sections 9.1 and 9.2), or if it contains one or more kinematical loops (e.g., the example in section 9.3), then additional constraint equations are needed.

Nonholonomic Constraints

Dynamical degrees of freedom can be eliminated by the analyst by imposing nonholonomic constraints.

Derivation of Constraint Coefficients

A constraint equation is a scalar expression constrained to be zero, having the form:

$$f_s(q_1,q_2, \ldots q_n,u_1,u_2, \ldots u_\nu,t) = 0 \qquad (s=p+1,\ldots \nu) \tag{8.3.1}$$

Each expression f_s is associated with one speed, u_s, and is generally the result of a dot-product between a velocity (angular or translational) and a unit-vector (e.g., see examples in Sections 9.1 and 9.2). Such expressions are always linear with respect to generalized speeds, although the coefficients can be nonlinear functions of generalized coordinates and/or time. Thus, f_s can be written in the form

$$\begin{aligned} f_s &= \frac{\partial f_s}{\partial u_1} u_1 + \frac{\partial f_s}{\partial u_2} u_2 + \ldots + \frac{\partial f_s}{\partial u_\nu} u_\nu + f_{s0} \\ &= f_{s1}\, u_1 + f_{s2}\, u_2 + \ldots f_{s\nu}\, u_\nu + f_{s0} \end{aligned} \tag{8.3.2}$$

where

$$f_{si} = \frac{\partial f_s}{\partial u_i} \qquad\qquad f_{s0} = f_s - \sum_{i=1}^{\nu} f_{si}\, u_i \tag{8.3.3}$$

Recall that in Kane's formulation, constraints are defined by scalar coefficients such that nonholonomic speeds are written as linear combinations of independent speeds,

$$u_s = \sum_{r=1}^{p} A_{sr} u_r + b_s \quad (s = p+1, \ldots \nu) \tag{8.3.4}$$

At the time a constraint is applied, it is possible to solve for u_s, if u_s appears in f_s. That is, if

$$f_{ss} \neq 0 \tag{8.3.5}$$

then a replacement expression of u_s is

$$u_s = \frac{-f_{s0} - \sum_{i=1,\, i\neq s}^{\nu} f_{si}\, u_i}{f_{ss}} \tag{8.3.6}$$

At the time a constraint is added with the `add-constraint` macro, the speeds are renumbered such that the speed being eliminated is u_s, where s>p. A replacement expression for u_s is obtained via the function `solve-for` and put into the *exp* slot of the `indexed-sym` object that represents the variable u_s. The *category* slot is set to the symbol `nonholonomic`.

Note that all generalized speeds, both independent and nonholonomic, appear in 8.3.2, but only one nonholonomic speed (u_s) appears in eq. 8.3.4. Conversion from the form of eq. 8.3.2 to that of eq. 8.3.4 cannot be done at the time a constraint is specified, because at that time it is not known how many other constraints will be introduced later by the analyst. That is, the expressions obtained by eq. 8.3.6 might include speeds that will later be removed by additional constraints. However, from a different perspective, expressions obtained previously might include the speed that was just eliminated, u_s. Accordingly, when a constraint is added, all of the existing nonholonomic speeds are processed so that any occurrences of u_s are replaced with the expression in eq. 8.3.6. By performing this expansion when each constraint is added, it is certain that at all times the replacement expression for each nonholonomic speed is a function only of time, generalized coordinates, and independent speeds.

The preceding analysis is performed as each constraint is added, along with other activities that were described in Section 8.1. The above analysis was described in this section, rather than Section 8.1, to establish continuity with the analysis methods described below. The constraint analysis continues after the kinematical equations have been written. At that time, the coefficients referenced in eq. 8.3.4 are obtained:

$$A_{sr} = \frac{\partial u_s}{\partial u_r} \qquad b_s = u_s - \sum_{r=1}^{p} A_{sr} u_r \tag{8.3.7}$$

A third set of coefficients is also established:

$$c_s = \dot{b}_s + \sum_{r=1}^{p} u_r \dot{A}_{sr} \tag{8.3.8}$$

Selection of Nonholonomic Speeds

The preceding analysis is presented under the presumption that as each constraint is applied, one formerly independent speed (u_s) has been chosen for "removal" (i.e., replacement with an expression involving the remaining independent speeds). The `add-constraint` macro does in fact allow the analyst to choose the speed variable to remove. However, the speed can also be chosen automatically by the macro. The criteria for selecting a speed to eliminate are based on an inspection of the above equations.

The first criterion is that the expression f_{ss} must not be zero, because it appears in the denominator of eq. 8.3.6. The AUTOSIM function `constant-part` is applied to each partial derivative in eq. 8.3.2, and only speeds associated with partial derivatives that

include a constant component are considered. (Because most constant parameters are represented by symbols, there is some faith here that nonzero parameter values are provided by the end user of the simulation code. An option is provided for the analyst to specify the speed to remove if he or she knows that some parameters are more likely than others to never have zero values.)

Note that the coefficients A_{sr} involve the partial derivative of the expression that replaces u_s, and that the coefficients c_s involve the derivatives of A_{sr}. From eq. 8.3.6, we see that the expression f_{ss} appears in every term in u_s. Thus, if f_{ss} has a derivative that is a complicated expression, then the expressions obtained for the non-zero coefficients defined in eqs. 8.3.7 and 8.3.8 will also be complicated. For this reason, the second criterion is that the partial derivative of the constraint with respect to u_s should be a constant.

The replacement expression is more complicated than a symbol, and therefore speeds that appear rarely in the system equations are prefered when choosing u_s. The nature of a tree topology is such that variables introduced for bodies that have children are likely to appear in expressions for the children when recursive formulations are employed. Therefore, the third criterion is that the speed eliminated should correspond to a body with no children. The speeds are numbered such that the highest indices correspond to bodies with no children, whereas speeds with low indices correspond to bodies "up" the tree, with children. Therefore, this criterion is applied by choosing the speed with the highest index from the list of candidates.

The actual procedure for choosing a speed to eliminate is as follows:

1. The set of all independent speeds with non-zero `constant-parts` in the partial derivatives of the constraint equation (f_{si}, i=1, ... ν) is formed. If this set is NIL, a message is printed to the analyst and an automatic selection is not made. Otherwise, the procedure continues.

2. From the set obtained in step 1, a subset is formed that includes only speeds corresponding to constant partial derivatives of the constraint equation. However, if this set is NIL, the set from step 1 is used.

3. The speed with the highest index that appears in the set formed in step 2 is chosen.

Kinematical Loops

Kinematical loops occur when a holonomic constraint links two bodies that do not have a parent-child relationship.

Example: Four-Bar Linkage

To help describe the handling of loops, the four-bar linkage shown in Figure 8.3.1 is used as an example. (This example will appear again in Section 9.3.) Suppose bodies A and C are introduced with parent N, and B is introduced with A as its parent. Thus, the tree appears as shown in Figure 8.3.2.

In this example, the pin joint between B and C still needs to be accounted for. To do this, one or more constraint equations must be written, based on the definition of the joint. Call the location of point P on body C point C_P, and on body B, point B_P. The constraint should state mathematically that C_P and B_P coincide. Because the state variables include both coordinates and speeds, it is necessary to state two facts about the joint: (1) the position between C_P and B_P is zero, and (2) the velocity between C_P and B_P is zero.

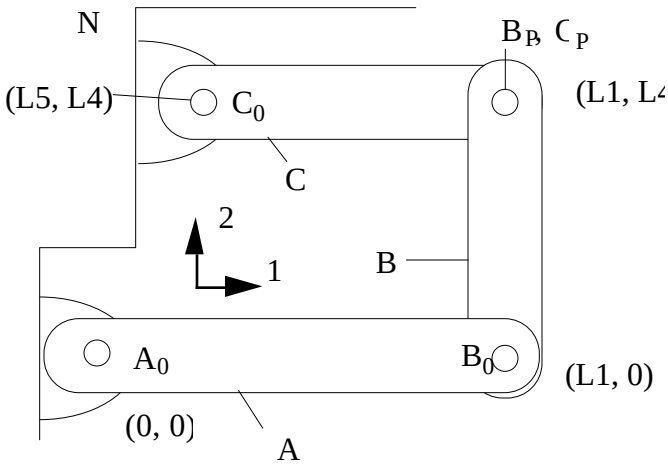

Figure 8.3.1. Four-bar linkage.

The condition that no movement exists is defined for forming expressions for the velocity and position vectors between the two points, and then dotting those vectors in appropriate directions.

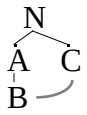

Figure 8.3.2. Tree for linkage.

The position vector between the two points can be written as

$$0 = \vec{\mathbf{r}}^{B_P C_P} = (L_1 - L_5)\,\vec{\mathbf{c}}_1 - L_1\,\vec{\mathbf{a}}_1 - L_4\,\vec{\mathbf{b}}_2 + L_5\,\vec{\mathbf{n}}_1 + L_4\,\vec{\mathbf{n}}_2 \qquad (8.3.9)$$

(The above expression is formulated in AUTOSIM using the `pos` function.) Dotting $\vec{\mathbf{r}}^{B_P C_P}$ in the direction $\vec{\mathbf{b}}_1$ yields the following scalar equation:

$$\begin{aligned} 0 &= \vec{\mathbf{r}}^{B_P C_P} \bullet \vec{\mathbf{b}}_1 \\ &= -L_1 c_2 + L_5(c_1 c_2 - s_1 s_2) + L_4(c_2 s_1 + c_1 s_2) \\ &\qquad + (L_1 - L_5)\left[-s_2(c_3 s_1 - c_1 s_3) + c_2(c_1 c_3 + s_1 s_3)\right] \end{aligned} \tag{8.3.10}$$

where sines and cosines are abbreviated as: $c_i \equiv \cos(q_i)$, $s_i \equiv \sin(q_i)$, and q_1, q_2, and q_3 are generalized coordinates introduced for the angular rotations of bodies A, B, and C, respectively.

The `vel` function is used to formulate an expression for the difference in velocity between the two points,

$$0 = \vec{\mathbf{v}}^{B_P C_P} = (L_1 - L_5) u_3 \vec{\mathbf{c}}_2 - L_1 u_1 \vec{\mathbf{a}}_2 + L_4(u_1 + u_2)\vec{\mathbf{b}}_1 \tag{8.3.11}$$

where speeds u_1, u_2, and u_3 are defined as the derivatives of q_1, q_2, and q_3, respectively. As before, a scalar constraint equation is obtained by dotting the above formulation with $\vec{\mathbf{b}}_1$:

$$\begin{aligned} 0 &= \vec{\mathbf{v}}^{B_P C_P} \bullet \vec{\mathbf{b}}_1 \\ &= (L_1 - L_5)\, u_3 \left[c_2\,(c_3 s_1 - c_1 s_3) + s_2\,(c_1 c_3 + s_1 s_3)\right] \\ &\qquad - L_1 u_1 s_2 + L_4\,(u_1 + u_2) \end{aligned} \tag{8.3.12}$$

On inspecting the above two constraint equations, it is clear that eq. 8.3.12 is well suited to solving for a speed variable (e.g., u_2) using the solution method of eq. 8.3.6. However, it is just as clear that an algebraic solution for a coordinate with eq. 8.3.10 is much more complicated.

Computational Methods

The constraints for displacement are almost unnecessary. Suppose that only two constraint equations are used, and the constraints are treated as nonholonomic. Then, we still have *n* generalized coordinates, computed by integrating the kinematical equations. We also have *p* independent speeds, computed by integrating the dynamical equations. There are just two minor problems with this formulation:

1. The initial values for all of the generalized coordinates are not known. Because some of the coordinates are not independent, they must be assigned initial values that satisfy the constraint equations.

2. The numerical integration involves some error that is very small, but which can accumulate over a long simulation run to violate the displacement constraint by a gradually increasing amount.

The approach taken is to treat the system as being nonholonomic, but to account for the above two potential problems computationally.

Let the displacement constraint equations have the form

$$g_s(q_1,q_2, \ldots q_n, t) = 0 \qquad (s=\pi+1,\ldots n) \tag{8.3.13}$$

Each expression g_s is associated with one coordinate, q_s, and is generally the result of a dot-product involving a displacement (angular or translational) and a unit-vector (e.g., eq. 8.3.10). Given a constraint of this generic form, the solution method used in eq. 8.3.6 for the speed constraint is first attempted. If it fails (as is usually the case for constraints defined from displacement equations), then numerical computational procedures are formulated. Instead of replacing the coordinate q_s with an exact analytical solution, q_s is categorized as a *computed coordinate*. The constraint equation g_s is placed in the *category* slot of the `indexed-sym`, where it is available for later writing computational procedures.

When a simulation run is started, initial values must be obtained for the computed coordinates such that the constraint equations are satisfied. The computation method used is a Newton-Raphson iteration, using an established algorithm for a set of nonlinear simultaneous algebraic equations [93]. After all of the constraints have been entered by the analyst, they are written as

$$g_{\pi+i} = 0 \qquad (i = 1, \ldots \mu) \tag{8.3.14}$$

The original constraint equations are available from the *category* slots of the computed coordinates, and are used to write a subroutine called INITNR to compute (1) the μ values of g_s, and (2) the $\mu \times \mu$ Jacobian matrix, whose elements are defined as:

$$J_{ij} = -\frac{\partial g_{\pi+i}}{\partial q_{\pi+j}} \qquad (i = 1, \ldots\mu, j = 1, \ldots\mu) \tag{8.3.15}$$

The coefficients defined in eqs. 8.3.14 and 8.3.15 provide all of the information needed to compute the correct initial values at the start of the simulation. Thus, the first potential problem is solved. (See Section 9.3 and Appendix C for the subroutines generated by AUTOSIM to perform the Newton-Raphson iteration.)

The second potential problem is that for long simulation runs, accumulated error in the numerical integration can result in violation of the displacement constraints. The solution is to include a correction of the form

$$q_s \leftarrow q_s - \frac{g_s}{\partial g_s / \partial q_s} \tag{8.3.16}$$

where the left arrow means "the value of on the left-hand side of the arrow (q_s) is replaced with the expression on the right-hand side of the arrow." Note that if the constraint is satisfied, then g_s is zero and q_s is not modified by the computation. Eq. 8.3.16 is applied at each time step, so the correction is typically very small, accounting for integration error over one time step. Interactions with other variables are numerically negligible and are not included in the computation here. Note that the denominator of the correction term in eq. 8.3.16 is a diagonal element of the Jacobian matrix defined in eq. 8.3.15.

A restriction on eq. 8.3.16 is that the denominator of the correction term (i.e., the Jacobian coefficient) must be nonzero. The current version of AUTOSIM does not automatically generate IF-THEN blocks to check for such singularities. Hence, the formulations can sometimes become singular. Generally, this is not a problem for kinematical loops occuring in vehicle systems, where motions of links in suspensions are limited to 20 or 30 degrees. (The singularity does not arise in steering systems, where angles cover the full range.) However, for applications involving general mechanism design, the possibility of a singularity should be considered. (A simple "fix" in the simulation code is to skip the correction when the absolute value of the Jacobian coefficient is smaller than some threshold, say, 10^{-10}.)

The computation indicated in eq. 8.3.16 is obtained by the `solve-for` function when an explicit expression cannot be obtained and the optional `:numerical` keyword argument is given the value T. The expression is put into the *exp* slot of the `indexed-var` and the *category* slot is set to the constraint expression g_s.

Section 9.3 and Appendix C show examples of how these computations appear in the simulation code.

Automatic Selection of Computed Coordinates

As was the case for the speeds, the above material was presented assuming an independent coordinate had been chosen by the analyst to convert to a computed coordinate. When used for constraints of displacement, the `add-constraint` macro allows the analyst to choose the coordinates to remove, just as it does when used for a constraint of speed. Also, the coordinate can also be chosen automatically. The criteria for selecting an independent coordinate to eliminate are similar, but less stringent, than those used for selecting speeds to eliminate.

First, the independent coordinates are checked, going from the coordinate with the highest index to q_1, to see if a coordinate exists whose partial derivative of g_s includes a constant part. The first one found is selected.

If the above search fails, then the independent coordinates are checked, again going from the coordinate with the highest index to q_1, to see if a coordinate exists whose partial derivative of g_s has a nominal value that is not zero. (The nominal value is the expression obtained when all generalized coordinates are zero, and is obtained with the function `nominal`.) If first one found is selected.

If the first criterion is satisfied, the Jacobian coefficients involving this computed coordinate have a denominator that is highly unlikely to be zero. If the second is satisfied, the coefficients are unlikely to be zero for configurations close to the nominal case. If both of the above methods fail to select a coordinate, a message is printed to the analyst and the constraint is not added. (To add the constraint, the analyst must choose the coordinate to eliminate.)

The No-Movement Macro

Kinematical loops require that constraint equations be added in pairs: one for speed and one for displacement. Given that there is some redundancy, it is essential that the two are consistent. To ensure this, a macro `no-movement` is used to generate holonomic constraints that eliminate motion of one point relative to another in some direction. The macro has three arguments: *point1, point2,* and *direction.* It forms an expression for the position vector between *point1* and *point2* (via the `pos` function) and dots the result with *direction* to obtain a scalar displacement constraint. That constraint is applied with the `add-constraint` function. Then, the macro forms a velocity vector between *point1*

and *point2* (via the `vel` function) and dots the result with *direction* to obtain a speed constraint that is also applied with the `add-constraint` function.

The `no-motion` macro is suitable for closing kinematical loops with the mathematical equivalents of planar slider joints (no movement in one direction), linear slider joints (no movement in two directions), pin joints (no movement in two directions), and ball joints (no movement in three directions).

Redundant Constraints

The `add-constraint` function and the `no-movement`macro do nothing if the constraint equation is already satisfied. There is usually not a problem if the analyst tries to apply too many constraints. For an example, see Section 9.2.

8.4. Dynamics Analysis

For a given multibody system, the minimal number of independent speeds is determined solely by the number of degrees of freedom. Introducing new symbols to match the number of degrees of freedom is a trivial exercise: they are variables called u_1, u_2, … u_p. However, deciding what those symbols represent physically is where modeling judgement comes into play. Because the partial velocities are used so frequently in deriving the equations of motion, their formulation is a primary factor in determining the complexity of equations of motion for the system. Specifically, for each body,

- the nonholonomic partial central velocities are dotted with the forces acting on that body,
- the nonholonomic partial angular velocities are dotted with the moments acting on that body,
- an expression for the angular acceleration remainder is developed, dotted with the inertia dyadic for the body, and the result is dotted with each nonholonomic partial angular velocity, and
- an expression for the central acceleration remainder is developed and dotted with each nonholonomic partial velocity for that body.

The nonholonomic partial angular and central velocities were defined in Chapter 6 as linear combinations of the holonomic counterparts. Similarly, the nonholonomic angular and central acceleration remainders were defined from holonomic expressions. The

previous section described how the coefficients needed to define nonholonomic expressions are obtained from the constraints. Therefore, if the holonomic terms are formulated, it will be simple to obtain the nonholonomic counterparts. Accordingly, this section focuses on the holonomic terms.

Two approaches will be used for defining holonomic partial velocities:

1. *Non-recursive* — Generalized speeds are defined such that most of the partial velocities and partial angular velocities for a body are either (1) zero, such that associated dot products are also zero, or (2) identical to other partial velocities, such that dot-products obtained for one partial velocity can be used again without further computation. Zero partial velocities are obtained when speeds are defined independently of other bodies, i.e. relative to the inertial reference. Identical partial velocities are obtained when speeds are defined for directions that are parallel to previously introduced speeds.

2. *Recursive* — Recursion is employed, such that expressions introduced for body B include results already obtained for the parent body A. This is accomplished by defining speeds associated with B relative to A.

Clearly, the two approaches conflict. In general, the first approach is preferred. However, it can be used only under certain conditions.

Before developing expressions for the partial velocities and acceleration remainder, two important algebraic considerations are discussed. These are (1) selection of a vector basis and (2) the introduction of intermediate variables. Neither is of any consequence with respect to the *correctness* of the equations of motion, but both are of great consequence with respect to the *efficiency* of the simulation code that applies the equations of motion.

Vector Bases

A vector is an expression involving products of scalars and unit-vectors. A given vector can be written in different ways, using alternative unit-vectors. A vector written using only the three unit-vectors aligned along the axes of the coordinate system of body B is said to be expressed in the *basis* of B. A vector written with no explicit trigonometric functions is said (in this dissertation) to be expressed in *native form.* For example, consider a system of two bodies A and B, where A rotates relative to N about an axis oriented in the

direction $\vec{\mathbf{n}}_1$ and B rotates relative to A about an axis oriented in the direction $\vec{\mathbf{a}}_2$. (In this example, the axes of the coordinate systems of N, A, and B are aligned in the nominal configuration. Thus, $\vec{\mathbf{n}}_1$ coincides with $\vec{\mathbf{a}}_1$ and $\vec{\mathbf{a}}_2$ coincides with $\vec{\mathbf{b}}_2$.) The angular velocity of B is $\vec{\omega}^B$:

$$\vec{\omega}^B = u_1 \vec{\mathbf{n}}_1 + u_2 \vec{\mathbf{a}}_2 \tag{8.4.1}$$

This velocity vector can be dotted with a basis dyadic without changing its magnitude or direction. That is,

$$\vec{\omega}^B = \vec{\omega}^B \bullet \overleftrightarrow{\mathbf{n}} = \vec{\omega}^B \bullet \overleftrightarrow{\mathbf{a}} = \vec{\omega}^B \bullet \overleftrightarrow{\mathbf{b}} \tag{8.4.2}$$

where

$$\begin{aligned} \overleftrightarrow{\mathbf{n}} &= \vec{\mathbf{n}}_1 \vec{\mathbf{n}}_1 + \vec{\mathbf{n}}_2 \vec{\mathbf{n}}_2 + \vec{\mathbf{n}}_3 \vec{\mathbf{n}}_3 \\ \overleftrightarrow{\mathbf{a}} &= \vec{\mathbf{n}}_1 \vec{\mathbf{n}}_1 + \vec{\mathbf{a}}_2 \vec{\mathbf{a}}_2 + \vec{\mathbf{a}}_3 \vec{\mathbf{a}}_3 \\ \overleftrightarrow{\mathbf{b}} &= \vec{\mathbf{b}}_1 \vec{\mathbf{b}}_1 + \vec{\mathbf{a}}_2 \vec{\mathbf{a}}_2 + \vec{\mathbf{b}}_3 \vec{\mathbf{b}}_3 \end{aligned} \tag{8.4.3}$$

The expressions obtained by dotting the angular velocity with the three basis dyadics are:

$$\begin{aligned} \vec{\omega}^B &= \vec{\omega}^B \bullet \overleftrightarrow{\mathbf{n}} = u_1 \vec{\mathbf{n}}_1 + u_2 c_1 \vec{\mathbf{n}}_2 + u_2 s_1 \vec{\mathbf{n}}_3 \\ &= \vec{\omega}^B \bullet \overleftrightarrow{\mathbf{a}} = u_1 \vec{\mathbf{n}}_1 + u_2 \vec{\mathbf{a}}_2 \\ &= \vec{\omega}^B \bullet \overleftrightarrow{\mathbf{b}} = u_1 c_2 \vec{\mathbf{b}}_1 + u_2 \vec{\mathbf{a}}_2 + u_1 s_2 \vec{\mathbf{b}}_3 \end{aligned} \tag{8.4.4}$$

where c_1 and s_1 are the cosine and sine functions of the rotation angle between N and A, and c_2 and s_2 are trigonometric functions for the angle between A and B.

In this example, the expression is simplest in the basis of A, being identical to the native form of eq. 8.4.1. For more complicated systems, the native form cannot be expressed in a single basis.

In the following material, the basis of each vector expression is a matter of concern. The basis is indicated with a dot product with a basis dyadic, as shown in eq. 8.4.2. When a basis is not specified, then the native form is retained.

Intermediate Variables

When the equations of motion are written into a Fortran program, it is intended that each arithmetic operation be performed only once. An expression that appears more than once is replaced with an intermediate variable, and the intermediate variable is used subsequently. The replacement of an expression with an intermediate variable is made by using the function `intro-var-if-new`. In the following material, the invocation of this function is indicated by enclosing an expression with the symbols "«" and "»." For example, the expression « $\vec{\omega}^{B} \bullet \overleftrightarrow{\mathbf{b}}$ » is interpreted as: "take the dot product as indicated, then invoke the `intro-var-if-new` function." For the above example, the result would be an expression similar to the following:

$$«\vec{\omega}^{B} \bullet \overleftrightarrow{\mathbf{b}}» \leftarrow z_8 \vec{\mathbf{b}}_1 + u_2 \vec{\mathbf{a}}_2 + z_9 \vec{\mathbf{b}}_3 \tag{8.4.5}$$

where z_8 and z_9 are intermediate variables introduced for the expressions u_1c_2 and u_1s_2, respectively. All expressions developed later involving « $\vec{\omega}^{B} \bullet \overleftrightarrow{\mathbf{b}}$ » would include z_8 and z_9, rather than u_1c_2 and u_1s_2.

The timing in the analysis at which intermediate variables are introduced is also a matter of concern. Generally, intermediate variables are introduced whenever an expression is formulated that is used in at least two products subsequently. (A "product" here means (1) a vector dot product, (2) a vector cross product, or (3) the result of a scalar multiplication.)

After all of the equations of motion have been developed, they are processed recursively with the `intro-var-if-new` function, to pick up any miscellaneous opportunities to introduce intermediate variables. However, the best efficiency is obtained by strategically introducing most of the intermediate variables as the analysis proceeds.

Initialization of Dynamics Analysis

The holonomic partial velocities and partial angular velocities are represented as arrays of dimension ν. Because ν is not known until all bodies and constraint equations have been entered by the analyst, the arrays are not formed until the system has been described in its entirety. The analysis is then performed by traversing the tree from the top down, such that the parent of each body B is analyzed before dealing with B. Given the design of

the `body` object, this form of tree traversal is very easy to implement. (Example Lisp code for performing the traversal was presented in Section 5.2.)

The analysis of the topology tree begins with the inertial reference, N. The two arrays of partial velocities and partial angular velocities associated with N are each filled with ν zeros and put into the *holo-wis and holo-v*is* slots of the worksheet. That is,

$$\vec{\mathbf{v}}_i^{N*} = \vec{\omega}_i^N = 0, \quad (i = 1, \ldots \nu) \tag{8.4.6}$$

Also, the angular velocity and angular acceleration remainder for N are set to zero:

$$\vec{\alpha}_{rem}^N = \vec{\omega}^N = 0 \tag{8.4.7}$$

The central acceleration remainder is set to the negative acceleration due to gravity:

$$\vec{\mathbf{a}}_{rem}^{N*} = -\vec{\mathbf{g}} \tag{8.4.8}$$

(If the analyst has not included gravity with the `add-gravity` macro, then the vector $\vec{\mathbf{g}}$ has a value of zero.)

The analysis is broken into two steps: rotational and translational. It will be seen that expressions developed for the partial central velocities of body B can include the partial angular velocities of B, and that the expression for the central acceleration remainder can include the angular acceleration remainder. Therefore, the rotation analysis is performed first.

Rotation Analysis

Four cases are considered for the rotation analysis: (1) a “general” recursive formulation, (2) the special case of a *rotor*, in which the rotational inertial properties about the mass center can be lumped with those of the parent body, (3) planar motions, and (4) unconstrained rotation.

Except for case 2, it is desirable to have the angular velocity and angular acceleration remainders of B expressed in the basis of B, because products taken with the inertia dyadic are projected into the basis of B. (See eqs. 6.3.19 and 6.3.20.)

General Recursive Formulation

In the general recursive formulation, expressions are developed for incremental terms that are added to expressions already developed for the parent of body B to obtain a new expression for B.

The recursive formulations for angular velocity and acceleration will be used only when the joint has zero or one rotational degree of freedom. Thus, the relative angular velocity of B with respect to its parent A is

$$^{A}\vec{\omega}^{B} = u_r \vec{\mathbf{r}}^{B}_{rot} \tag{8.4.9}$$

where u_r is the derivative of the generalized coordinate associated with the joint rotation and $\vec{\mathbf{r}}^{B}_{rot}$ is the axis of rotation. The angular velocity of body B is then

$$\vec{\omega}^{B} = \vec{\omega}^{A} \bullet \overleftrightarrow{\mathbf{b}} + {}^{A}\vec{\omega}^{B} \tag{8.4.10}$$

The dot-product with $\overleftrightarrow{\mathbf{b}}$ puts $\vec{\omega}^{A}$ into the basis of B. (The axis of rotation is already in the basis of B.) Expanding the above in terms of partial angular velocities gives the following:

$$\vec{\omega}^{B} = \sum_{i=1}^{\nu} u_i \vec{\omega}^{A}_{i} \bullet \overleftrightarrow{\mathbf{b}} + u_r \vec{\mathbf{r}}^{B}_{rot} \tag{8.4.11}$$

By inspection,

$$\vec{\omega}^{B}_{i} = \vec{\omega}^{A}_{i} \bullet \overleftrightarrow{\mathbf{b}} + {}^{A}\vec{\omega}^{B}_{i} \quad (i = 1, \ldots \nu) \tag{8.4.12}$$

where

$$^{A}\vec{\omega}^{B}_{i} = \left\{ \begin{array}{ll} \vec{\mathbf{r}}^{B}_{rot} & \text{for } i = r \\ 0 & \text{for } i \neq r \end{array} \right\} \quad (i = 1, \ldots \nu) \tag{8.4.13}$$

Recall the definition of the holonomic angular acceleration remainder (eq. 6.3.10):

$$\vec{\alpha}^{B}_{rem} = \sum_{r=1}^{\nu} u_r \frac{d\vec{\omega}^{B}_{r}}{dt} \tag{8.4.14}$$

substituting eq. 8.4.12 into 8.4.14 yields the following:

$$\vec{\alpha}^{B}_{rem} = \left(\sum_{i=1}^{\nu} u_i \frac{d\vec{\omega}^{A}_{i}}{dt} \right) + u_r \frac{d\vec{r}^{B}_{rot}}{dt}$$

$$= \vec{\alpha}^{A}_{rem} + u_r \frac{d\vec{r}^{B}_{rot}}{dt}$$

$$= \vec{\alpha}^{A}_{rem} + u_r \vec{\omega}^{A} \times \vec{r}^{B}_{rot}$$

$$= \vec{\alpha}^{A}_{rem} + \vec{\alpha}^{AB}_{rem}$$

$$= \vec{\alpha}^{A}_{rem} \bullet \overleftrightarrow{b} + \vec{\alpha}^{AB}_{rem} \qquad (8.4.15)$$

where

$$\vec{\alpha}^{AB}_{rem} = u_r \left(\vec{\omega}^{A} \times \vec{r}^{B}_{rot} \right)$$

$$= \vec{\omega}^{A} \times {}^{A}\vec{\omega}^{B}$$

$$= \left(\vec{\omega}^{A} \bullet \overleftrightarrow{b} \right) \times {}^{A}\vec{\omega}^{B} \qquad (8.4.16)$$

Formulation for a Rotor

Body B is classified as a rotor if it has one rotational degree of freedom, and the same moment of inertia is obtained about any direction normal to the rotation axis. In this case the inertia dyadic was formulated using unit-vectors from the parent. Expressions dotted with the inertia dyadic are projected into the basis of the parent. Thus, expressions for rotation are all obtained in the basis of the parent. That is,

$$\vec{\omega}^{B} = \left(\vec{\omega}^{A} + {}^{A}\vec{\omega}^{B} \right) \bullet \overleftrightarrow{a} \qquad (8.4.17)$$

$$\vec{\omega}^{B}_{i} = \left(\vec{\omega}^{A}_{i} + {}^{A}\vec{\omega}^{B}_{i} \right) \bullet \overleftrightarrow{a} \quad (i = 1, \ldots \nu) \qquad (8.4.18)$$

$$\vec{\alpha}^{B}_{rem} = \left(\vec{\alpha}^{A}_{rem} + \vec{\alpha}^{AB}_{rem} \right) \bullet \overleftrightarrow{a} \qquad (8.4.19)$$

$$\vec{\alpha}^{AB}_{rem} = \left(\vec{\omega}^{A} \times {}^{A}\vec{\omega}^{B} \right) \bullet \overleftrightarrow{a} \qquad (8.4.20)$$

where the incremental angular velocity and partial velocities are defined as before in eqs. 8.4.9 and 8.4.13.

The above formulation is more efficient than the general recursive formulation, because transformations to the coordinate system of B are avoided when dealing with the rotational dynamics of B. If B has no children, the sine and cosine of the rotation angle of B relative to A do not in appear in the equations of motion unless they are required for a force/moment-producing element or an output variable defined by the analyst.

Planar Motions

If a rigid body is constrained to planar motions, then all rotations occur about the direction perpendicular to the plane of the motion, and the velocity vector of any point on the body is always parallel to the plane.

In this case, the recursive formulation for angular velocity results in all nonzero partial angular velocities being identical, namely the unit-vector normal to the plane. The cross-product in eq. 8.4.20 is always zero, and therefore the acceleration remainder is zero. Both of these forms are desirable, and therefore the general recursive formulation is well suited for introducing partial angular velocities for bodies constrained to planar motions.

Three Rotational Degrees of Freedom

A general three-dimensional angular velocity can be completely described with three independent speeds variables. When a body has three rotational degrees of freedom relative to its parent, then the three generalized rotational speeds can be defined to characterize the angular velocity without reference to any other bodies.

For ground and air vehicles, rotational speeds named roll rate (p), pitch rate (q), and yaw rate (r) are defined about three axes fixed in the body. That is, for body axes $\vec{\mathbf{b}}_1$, $\vec{\mathbf{b}}_2$, and $\vec{\mathbf{b}}_3$, and rotational velocity $\vec{\omega}^B$, the three generalized speeds are defined as

$$p = \vec{\omega}^B \bullet \vec{\mathbf{b}}_1 \qquad q = \vec{\omega}^B \bullet \vec{\mathbf{b}}_2 \qquad r = \vec{\omega}^B \bullet \vec{\mathbf{b}}_3 \tag{8.4.21}$$

or,

$$\vec{\omega}^B = p\,\vec{\mathbf{b}}_1 + q\,\vec{\mathbf{b}}_2 + r\,\vec{\mathbf{b}}_3 \tag{8.4.22}$$

This choice of variables leads to the simplest possible expression for angular velocity that can be expressed in the basis of B. Also, a simple expression for angular acceleration is obtained:

$$\vec{\alpha}^{B} = \frac{d\vec{\omega}^{B}}{dt}$$

$$= \dot{p}\,\vec{b}_1 + \dot{q}\,\vec{b}_2 + \dot{r}\,\vec{b}_3 + p\,\vec{\omega}^{B} \times \vec{b}_1 + q\,\vec{\omega}^{B} \times \vec{b}_2 + r\,\vec{\omega}^{B} \times \vec{b}_3$$

$$\begin{aligned} = \dot{p}\,\vec{b}_1 + \dot{q}\,\vec{b}_2 + \dot{r}\,\vec{b}_3 &+ p\left(p\,\vec{b}_1 + q\,\vec{b}_2 + r\,\vec{b}_3\right) \times \vec{b}_1 \\ &+ q\left(p\,\vec{b}_1 + q\,\vec{b}_2 + r\,\vec{b}_3\right) \times \vec{b}_2 \\ &+ r\left(p\,\vec{b}_1 + q\,\vec{b}_2 + r\,\vec{b}_3\right) \times \vec{b}_3 \end{aligned}$$

$$\begin{aligned} = \dot{p}\,\vec{b}_1 + \dot{q}\,\vec{b}_2 + \dot{r}\,\vec{b}_3 &+ pr\,\vec{b}_2 - pq\,\vec{b}_3 \\ &- qr\,\vec{b}_1 + qp\vec{b}_3 \\ &+ rq\,\vec{b}_1 - rp\,\vec{b}_2 \end{aligned}$$

$$= \dot{p}\,\vec{b}_1 + \dot{q}\,\vec{b}_2 + \dot{r}\,\vec{b}_3 \qquad (8.4.23)$$

By inspecting eq. 8.4.22, we find three nonzero partial angular velocities: $\vec{b}_1$, $\vec{b}_2$, and $\vec{b}_3$. Eq. 8.4.23 indicates that the angular acceleration remainder is identically zero.

Summary

Expressions developed in this analysis are stored in worksheet objects. Each `body` in the system has an associated object, kept in its *worksheet* slot. Slots pertaining to the rotation analysis are listed in Table 8.4.1. Slots such as *w-a* that include data copied from the parent are set to the vector expressed in a basis appropriate for analyzing the current body.

Table 8.4.1. Slots in body worksheet object pertaining to rotational velocity and acceleration.

Slot Name	Type	Definition
w-a	`expression`	absolute angular velocity of A.
w-ab	`expression`	angular velocity of B relative to A.
w	`expression`	absolute angular velocity of B.
holo-wis-ab	`array`	incremental holonomic partial angular velocities of B.
holo-wis	`array`	holonomic partial angular velocities of B.
alpha-ab	`expression`	incremental holonomic angular acceleration remainder for B.
alpha-rem	`expression`	holonomic angular acceleration remainder for B.

The expressions used to fill the slots are shown in Table 8.4.2. The recursive formulation is given in three forms in the table, for zero and one rotational degree of freedom, and for the case of a rotor. The nonrecursive formulation is used for bodies with three rotational degrees of freedom. Note that replacement of expressions by intermediate variables is also indicated in the table.

Table 8.4.2. Formulas pertaining to rotational velocity and acceleration.

Slot	Symbol	0 d.o.f.	rotor	1 d.o.f.	3 d.o.f.
w-a	$\vec{\omega}^{A'}$		$\langle\langle \vec{\omega}^{A} \cdot \vec{\vec{a}} \rangle\rangle$	$\langle\langle\langle \vec{\omega}^{A} \cdot \vec{\vec{a}} \rangle\rangle \cdot \vec{\vec{b}} \rangle\rangle$	
w-ab	${}^{A}\vec{\omega}^{B}$		$u_r \vec{\mathbf{r}}^{B}_{rot}$	$u_r \vec{\mathbf{r}}^{B}_{rot}$	
w	$\vec{\omega}^{B}$	$\vec{\omega}^{A}$	$\langle\vec{\omega}^{A'} + {}^{A}\vec{\omega}^{B}\rangle$	$\langle\vec{\omega}^{A'} + {}^{A}\vec{\omega}^{B}\rangle$	$u_{o+1}\,\vec{\mathbf{b}}_1 + u_{o+2}\,\vec{\mathbf{b}}_2 + u_{o+3}\,\vec{\mathbf{b}}_3$
holo-wis-ab	${}^{A}\vec{\omega}^{B}_{i}$		$\begin{cases} \vec{\mathbf{r}}^{B}_{rot} & \text{for } i = r \\ 0 & \text{for } i \neq r \end{cases}$	$\begin{cases} \vec{\mathbf{r}}^{B}_{rot} & \text{for } i = r \\ 0 & \text{for } i \neq r \end{cases}$	
holo-wis	$\vec{\omega}^{B}_{i}$	$\vec{\omega}^{A}_{i}$	$\langle\langle \vec{\omega}^{A}_{i} \cdot \vec{\vec{a}} \rangle\rangle + {}^{A}\vec{\omega}^{B}_{i}$	$\langle\langle\langle \vec{\omega}^{A}_{i} \cdot \vec{\vec{a}} \rangle\rangle \cdot \vec{\vec{b}} \rangle\rangle + {}^{A}\vec{\omega}^{B}_{i}$	$\begin{cases} \vec{\mathbf{b}}_{i\text{-}o} & \text{i-o=1,2,3} \\ 0 & \text{otherwise} \end{cases}$
alpha-ab	$\vec{\alpha}^{AB}_{rem}$		$\vec{\omega}^{A'} \times {}^{A}\vec{\omega}^{B}$	$\vec{\omega}^{A'} \times {}^{A}\vec{\omega}^{B}$	
alpha-rem	$\vec{\alpha}^{B}_{rem}$	$\vec{\alpha}^{A}_{rem}$	$\langle\langle \vec{\alpha}^{A}_{rem} \cdot \vec{\vec{a}} + \vec{\alpha}^{AB}_{rem} \rangle\rangle$	$\langle\langle \vec{\alpha}^{A}_{rem} \cdot \vec{\vec{b}} + \vec{\alpha}^{AB}_{rem} \rangle\rangle$	0

Translation Analysis

Four cases are considered for the translation analysis: (1) a "general" recursive formulation, (2) the special case of a *fixed mass*, in which the mass center is in a location fixed in the parent body, (3) unconstrained planar translation, and (4) unconstrained three-dimensional translation.

In the remainder of this section, all mass centers refer to the composite body mass center, stored as a `point` in the *cm-point* slot of the `body`.

General Recursive Formulation

In this formulation, the partial velocities and the acceleration of body B are defined relative to corresponding terms in the parent A.

The absolute position of B* (the mass center of B) can be defined relative to the position of A* (the mass center of parent A):

$$\vec{\mathbf{r}}^{B^*} = \vec{\mathbf{r}}^{A^*} + \vec{\mathbf{r}}^{A^*B_0} + \vec{\mathbf{r}}^{B_0B^*} \tag{8.4.24}$$

The derivative of this position gives the central velocity of B:

$$\begin{aligned}
\vec{\mathbf{v}}^{B^*} &= \frac{d\vec{\mathbf{r}}^{B^*}}{dt} \\
&= \frac{d\vec{\mathbf{r}}^{A^*}}{dt} + \frac{d\vec{\mathbf{r}}^{A^*B_0}}{dt} + \frac{d\vec{\mathbf{r}}^{B_0B^*}}{dt} \\
&= \vec{\mathbf{v}}^{A^*} + \vec{\omega}^{A} \times \vec{\mathbf{r}}^{A^*B_0} + {}^{A}\vec{\mathbf{v}}^{B_0} + \vec{\omega}^{B} \times \vec{\mathbf{r}}^{B_0B^*} + {}^{B}\vec{\mathbf{v}}^{B^*}
\end{aligned} \tag{8.4.25}$$

The first local velocity, ${}^{A}\vec{\mathbf{v}}^{B_0}$, accounts for translational degrees of freedom of B between the point B_J and B_0:

$${}^{A}\vec{\mathbf{v}}^{B_0} = \sum_{i=1}^{N_{td}^{B}} u_{o+j}\, \vec{\mathbf{r}}_{tj}^{B} \tag{8.4.26}$$

where N_{td}^{B} is the number of translational degrees of freedom for B, o is a constant offset needed to map the index of the generalized speeds to the summation index j, and $\vec{\mathbf{r}}_{tj}^{B}$ is the direction of the j^{th} translation of the joint. The second local velocity, ${}^{B}\vec{\mathbf{v}}^{B^*}$, is zero because B* is fixed in the coordinate system of B. Thus, eq. 8.4.24 can be written as

$$\vec{\mathbf{v}}^{B^*} = \vec{\mathbf{v}}^{A^*} + \vec{\omega}^{A} \times \vec{\mathbf{r}}^{A^*B_0} + \sum_{i=1}^{N_{td}^{B}} u_{o+j}\, \vec{\mathbf{r}}_{tj}^{B} + \vec{\omega}^{B} \times \vec{\mathbf{r}}^{B_0B^*} \tag{8.4.27}$$

By inspection, the partial velocities are written

$$\vec{\mathbf{v}}_i^{B^*} = \vec{\mathbf{v}}_i^{A^*} + \vec{\mathbf{v}}_i^{A^*B^*} \tag{8.4.28}$$

where the incremental partial velocity, $\vec{\mathbf{v}}_i^{A^*B^*}$, is defined for two cases, corresponding to (1) speeds introduced for translational degrees of freedom of the joint of B, and (2) all other speeds.

$$\vec{\mathbf{v}}_i^{A^*B^*} = \left\{ \begin{array}{cc} \vec{\mathbf{r}}_{tj}^{B} & \text{for } i = o+j,\ j=1, \ldots N_{td}^{B} \\ \vec{\omega}_i^{A} \times \vec{\mathbf{r}}^{A^*B_0} + \vec{\omega}_i^{B} \times \vec{\mathbf{r}}^{B_0B^*} & \text{all other } i \end{array} \right\} \quad (i = 1, \ldots \nu) \tag{8.4.29}$$

The holonomic central acceleration remainder was defined in eq.6.3.17 as:

$$\vec{\mathbf{a}}_{rem}^{B^*} = \sum_{i=1}^{\nu} u_i \frac{d\vec{\mathbf{v}}_i^{B^*}}{dt} \tag{8.4.30}$$

Combining eqs. 8.4.28 and 8.4.30 yields the following:

$$\vec{\mathbf{a}}_{rem}^{B^*} = \sum_{i=1}^{\nu} u_i \frac{d\vec{\mathbf{v}}_i^{A^*}}{dt} + \sum_{i=1}^{\nu} u_i \frac{d\vec{\mathbf{v}}_i^{A^*B^*}}{dt}$$

$$= \vec{\mathbf{a}}_{rem}^{A^*} + \sum_{i=1}^{\nu} u_i \frac{d\vec{\mathbf{v}}_i^{A^*B^*}}{dt}$$

$$= \vec{\mathbf{a}}_{rem}^{A^*} + \vec{\mathbf{a}}_{rem}^{A^*B^*} \tag{8.4.31}$$

where the incremental central acceleration remainder, $\vec{\mathbf{a}}_{rem}^{A^*B^*}$, is defined as:

$$\vec{\mathbf{a}}_{rem}^{A^*B^*} = \sum_{i=1}^{\nu} u_i \frac{d\vec{\mathbf{v}}_i^{A^*B^*}}{dt}$$

$$= \sum_{i=1}^{\nu} u_i \frac{d}{dt}\left(\vec{\omega}_i^{A} \times \vec{\mathbf{r}}^{A^*B_0} + \vec{\omega}_i^{B} \times \vec{\mathbf{r}}^{B_0B^*}\right) + \sum_{i=1}^{N_{td}^B} u_{o+j} \frac{d\vec{\mathbf{r}}_{tj}^{B}}{dt}$$

$$= \vec{\alpha}_{rem}^{A} \times \vec{\mathbf{r}}^{A^*B_0} + \vec{\omega}^{A} \times \vec{\mathbf{v}}^{A^*B_0} + \vec{\alpha}_{rem}^{B} \times \vec{\mathbf{r}}^{B_0B^*} + \vec{\omega}^{B} \times \vec{\mathbf{v}}^{B_0B^*} + \sum_{i=1}^{N_{td}^B} u_{o+j} \left(\vec{\omega}^{A} \times \vec{\mathbf{r}}_{tj}^{B}\right)$$

$$= \vec{\alpha}_{rem}^{A} \times \vec{\mathbf{r}}^{A^*B_0} + \vec{\omega}^{A} \times \left(\vec{\omega}^{A} \times \vec{\mathbf{r}}^{A^*B_0} + \sum_{i=1}^{N_{td}^B} u_{o+j} \vec{\mathbf{r}}_{tj}^{B}\right) + \vec{\alpha}_{rem}^{B} \times \vec{\mathbf{r}}^{B_0B^*} + \vec{\omega}^{B} \times \left(\vec{\omega}^{B} \times \vec{\mathbf{r}}^{B_0B^*}\right) + \sum_{i=1}^{N_{td}^B} u_{o+j} \left(\vec{\omega}^{A} \times \vec{\mathbf{r}}_{tj}^{B}\right)$$

$$= \vec{\alpha}_{rem}^{A} \times \vec{\mathbf{r}}^{A^*B_0} + \vec{\omega}^{A} \times \left(\vec{\omega}^{A} \times \vec{\mathbf{r}}^{A^*B_0}\right) + \vec{\alpha}_{rem}^{B} \times \vec{\mathbf{r}}^{B_0B^*} + \vec{\omega}^{B} \times \left(\vec{\omega}^{B} \times \vec{\mathbf{r}}^{B_0B^*}\right) + 2\sum_{i=1}^{N_{td}^B} u_{o+j} \left(\vec{\omega}^{A} \times \vec{\mathbf{r}}_{tj}^{B}\right)$$

Knowing that

$$\vec{\alpha}_{rem} \times \vec{\mathbf{r}} + \vec{\omega} \times \left(\vec{\omega} \times \vec{\mathbf{r}}\right) = \left[\vec{\alpha}_{rem} \times \overleftrightarrow{\mathbf{b}} + \vec{\omega} \times \left(\vec{\omega} \times \overleftrightarrow{\mathbf{b}}\right)\right] \bullet \vec{\mathbf{r}} \tag{8.4.33}$$

a dyadic is defined to clarify the recursion inherent in eq. 8.4.32,

$$\overleftrightarrow{\mathbf{a}}^{B}_{rot} = \vec{\alpha}^{B}_{rem} \times \overleftrightarrow{\mathbf{b}} + \vec{\omega}^{B} \times \left(\vec{\omega}^{B} \times \overleftrightarrow{\mathbf{b}}\right) \tag{8.4.34}$$

Then, eq. 8.4.32 can be expressed more simply, as

$$\vec{\mathbf{a}}^{A*B*}_{rem} = \overleftrightarrow{\mathbf{a}}^{A}_{rot} \bullet \vec{\mathbf{r}}^{A*B_0} + \overleftrightarrow{\mathbf{a}}^{B}_{rot} \bullet \vec{\mathbf{r}}^{B_0B*} + 2\sum_{i=1}^{N^{B}_{td}} u_{o+j}\left(\vec{\omega}^{A} \times \vec{\mathbf{r}}^{B}_{tj}\right) \tag{8.4.35}$$

Recall that the effect of a uniform gravitational field is accounted for by subtracting the acceleration due to gravity from the acceleration remainder. Due to the recursive nature of 8.4.31, the acceleration remainder for B will properly include the effect of gravity if it was included in the acceleration remainder of A. Thus, it is only necessary to explicitly subtract the gravity acceleration term (1) in the *acc-rem* slot of the `body` of the inertial reference, and (2) in the *acc-rem* slot for bodies that are nonrecursive in translation.

Upon inspecting eqs. 6.3.19 and 6.3.20, showing the uses made of the partial central velocities and central acceleration remainder, we see that expressions appearing in the dynamical equations that are contributed from the analysis of body B are (1) the dot products of the partial velocities with each other, (2) the dot products of the partial velocities with the acceleration remainder, and (3) the dot products of the partial velocities with forces acting on B. The terms in the partial velocities and acceleration remainders occur naturally in the bases of A and B. Dot products of vectors are simplest if the vectors are both expressed in the same basis, and therefore the partial velocities and acceleration remainders are converted to the basis of B after they are formulated.

There is a special case in which simpler expressions are obtained when the partial velocities are not expressed in the basis of B. This occurs when both $\vec{\mathbf{v}}^{A*B*}_{i}$ and $\vec{\mathbf{v}}^{A*B*}_{j}$ are zero. In this case, the two partial velocities $\vec{\mathbf{v}}^{B*}_{i}$ and $\vec{\mathbf{v}}^{B*}_{j}$ are identical to the corresponding partial velocities of the parent. Also, their dot product is the same as the one obtained for the parent. Obviously, it is more efficient in the simulation code to use an existing dot product as opposed to computing a new one. (Generally, incremental partial central velocities are zero when the corresponding generalized speed was introduced to account for a translational degree of freedom in a joint.)

To locate partial velocities that are unchanged from the unfixed parent, a slot in the `body` is set to an array of length `v` that contains bodies in which the partial velocities were last changed.

Formulation for a Fixed Mass

When the center of mass of body B is fixed in the coordinate system of the parent (i.e., the *recursive-t* slot is set to the symbol `fixed`), then the mass was lumped with that of the parent. When the dynamical equations are formulated, the mass used for B is zero. Therefore, partial central velocities and the acceleration remainder are not needed to form the dynamical equations. However, the recursive formulation presented above requires that these terms be defined for the parent body, even if the parent is classified as being `fixed`. Therefore, the general recursive analysis is applied to a fixed body if it has children. If the body does not have children, the partial velocities and acceleration remainder are set to zero.

One change in the above formulation is made when the either the body or the parent is fixed. That is, the point used for the mass center is the origin. This is done so that the position vector $\vec{\mathbf{r}}^{B_0B^*}$ is identically zero.

Planar Motions

In general, expressions for the position, velocity, and acceleration of the center of mass of a body undergoing planar motions involve both directions in the coordinate system of the plane. If the body has two translational degrees of freedom, a nonrecursive formulation is used so that the central velocity and central acceleration are formulated only in terms of the two new speeds. If the body has zero or one translational degree of freedom, it is not possible to develop expressions for central acceleration that do not involve the velocity and angular velocity of the parent body. Thus, the above recursive formulation is used when the body has zero or one translational degree of freedom.

There are two obvious choices for defining speeds when the body has two translational degrees of freedom: (1) use body-based directions, or (2) use inertial directions. First, consider the body-based option. The speeds are defined as:

$$u = \vec{\mathbf{v}}^{B^*} \bullet \vec{\mathbf{b}}_1 \qquad\qquad v = \vec{\mathbf{v}}^{B^*} \bullet \vec{\mathbf{b}}_2 \tag{8.4.36}$$

or,

$$\vec{\mathbf{v}}^{B^*} = u\,\vec{\mathbf{b}}_1 + v\,\vec{\mathbf{b}}_2 \tag{8.4.37}$$

There are but two nonzero partial velocities here: $\vec{\mathbf{b}}_1$ and $\vec{\mathbf{b}}_2$. The central acceleration remainder is

$$\begin{aligned}\vec{\mathbf{a}}^{B^*}_{rem} &= \sum_{i=1}^{v} u_i \frac{d\vec{\mathbf{v}}_i^{B^*}}{dt} - \vec{\mathbf{g}} \\ &= u\frac{d\vec{\mathbf{b}}_1}{dt} + v\frac{d\vec{\mathbf{b}}_2}{dt} - \vec{\mathbf{g}} \\ &= u\,\vec{\omega}^B \times \vec{\mathbf{b}}_1 + v\,\vec{\omega}^B \times \vec{\mathbf{b}}_2 - \vec{\mathbf{g}} \\ &= \vec{\omega}^B \times \vec{\mathbf{v}}^{B^*} - \vec{\mathbf{g}}\end{aligned} \tag{8.4.38}$$

(Note that acceleration due to gravity is added to the acceleration remainder when a nonrecursive formulation is used.)

Next, consider speeds defined for the inertial reference:

$$u_{t1} = \vec{\mathbf{v}}^{B^*} \bullet \vec{\mathbf{n}}_1 \qquad\qquad u_{t2} = \vec{\mathbf{v}}^{B^*} \bullet \vec{\mathbf{n}}_2 \tag{8.4.39}$$

or,

$$\vec{\mathbf{v}}^{B^*} = u_{t1}\,\vec{\mathbf{n}}_1 + u_{t2}\,\vec{\mathbf{n}}_2 \tag{8.4.40}$$

Here, the two nonzero partial velocities are $\vec{\mathbf{n}}_1$ and $\vec{\mathbf{n}}_2$. The central acceleration remainder is $\vec{\mathbf{g}}$:

$$\begin{aligned}\vec{\mathbf{a}}^{B^*}_{rem} &= \sum_{i=1}^{n} u_i \frac{d\vec{\mathbf{v}}_i^{B^*}}{dt} - \vec{\mathbf{g}} \\ &= u_{t1}\frac{d\vec{\mathbf{n}}_1}{dt} + u_{t2}\frac{d\vec{\mathbf{n}}_2}{dt} - \vec{\mathbf{g}} \\ &= -\vec{\mathbf{g}}\end{aligned} \tag{8.4.41}$$

On the basis of simplifying the central acceleration remainder, the choice of inertial directions is better than body-fixed directions. However, for vehicle systems, there are often advantages in defining speeds using body-based directions. For one thing, forces and moments acting on a vehicle are usually directed relative to the body-based coordinate system. Thus, the dot-products between active forces and partial velocities are simpler when the partial velocities are fixed relative to the body. Another consideration is that

constraints are commonly applied that are most naturally described using body-fixed directions. For example, forward speed is often set to a constant. Therefore, body-fixed directions are used in this work.

When a body has two translational degrees of freedom and is constrained to planar motions, the *recursive-t* slot is set to `NIL`. The directions of the translational speeds are the two unit-vectors of the body that are not the rotation axis for its one rotational degree of freedom. Those two unit-vectors were put in a list assigned to the slot *translational-speed-directions* at the time the body was added.

Three Degrees of Freedom in Translation

If body B has three translational degrees of freedom, then generalized speeds can be introduced so that the velocity and acceleration of the center of mass depend only on those speeds. As before (for the planar system, with a body with two translational degrees of freedom), the generalized speeds can be defined relative to body-based directions or inertial directions. Again, because ground vehicles are the type of multibody system being considered, body-based directions are preferred. The speeds are defined as:

$$u = \vec{\mathbf{v}}^{B^*} \bullet \vec{\mathbf{b}}_1 \qquad v = \vec{\mathbf{v}}^{B^*} \bullet \vec{\mathbf{b}}_2 \qquad w = \vec{\mathbf{v}}^{B^*} \bullet \vec{\mathbf{b}}_3 \tag{8.4.42}$$

or, conversely

$$\vec{\mathbf{v}}^{B^*} = u\,\vec{\mathbf{b}}_1 + v\,\vec{\mathbf{b}}_2 + w\,\vec{\mathbf{b}}_3 \tag{8.4.43}$$

The central acceleration remainder is

$$\begin{aligned}
\vec{\mathbf{a}}^{B^*}_{rem} &= \sum_{i=1}^{\nu} u_i \frac{d\vec{\mathbf{v}}_i^{B^*}}{dt} - \vec{\mathbf{g}} \\
&= u\frac{d\vec{\mathbf{b}}_1}{dt} + v\frac{d\vec{\mathbf{b}}_2}{dt} + w\frac{d\vec{\mathbf{b}}_3}{dt} - \vec{\mathbf{g}} \\
&= u\,\vec{\omega}^B \times \vec{\mathbf{b}}_1 + v\,\vec{\omega}^B \times \vec{\mathbf{b}}_2 + w\,\vec{\omega}^B \times \vec{\mathbf{b}}_3 - \vec{\mathbf{g}} \\
&= \vec{\omega}^B \times \vec{\mathbf{v}}^{B^*} - \vec{\mathbf{g}}
\end{aligned} \tag{8.4.44}$$

Summary

As with the rotation analysis, results of the translation analysis are kept in a worksheet object. Slots pertaining to the translation analysis are listed in Table 8.4.3. The expressions used to fill the slots are shown in Table 8.4.4. Note that the conversion to the basis of the current body B is performed by dotting expressions with the basis dyadic, $\overleftrightarrow{\mathbf{b}}$, as was done for the rotational expressions. For the partial central velocities, the same approach is taken, namely, that partial velocities of A are dotted with $\overleftrightarrow{\mathbf{b}}$ and added to the incremental expressions to get the partial velocities of B. However, for the acceleration remainder, the incremental expression $\vec{\mathbf{a}}_{\text{rem}}^{\text{A*B*}}$ is most conveniently left in a mixed basis, involving unit vectors from both A and B. Only after the full acceleration remainder for B is obtained ($\vec{\mathbf{a}}_{\text{rem}}^{\text{B}}$) is is the conversion made to the basis of B.

Table 8.4.3. Slots in body worksheet object pertaining to translational velocity and acceleration.

Slot Name	Type	Definition
acc-ab	`expression`	incremental holonomic central acceleration remainder for B.
acc-dyadic	`expression`	dyadic with rotational component of incremental acceleration remainder.
acc-rem	`expression`	holonomic central acceleration remainder for B.
*holo-v*is*	`array`	holonomic partial central velocities of B.
*holo-v*is-ab*	`array`	incremental holonomic partial central velocities of B.
holo-v-bodies	`array`	bodies in which corresponding holonomic partial velocities were last modified.
*ra*b0*	`expression`	position vector going from A* to B_0.

Form Dynamical Equations

Once the terms in Tables 8.4.2 and 8.4.4 are obtained for all bodies, it is straightforward to finish the analysis to obtain the dynamical equations. Specifically, the following steps are taken for each body B:

1. The constraint coefficients, derived in Section 8.3, are combined with the holonomic partial velocities (angular and central) and acceleration remainders (angular and central) to define the nonholonomic partial velocities and

Table 8.4.4. Formulas pertaining to translational velocity and acceleration.

Slot	Symbol	nonrecursive	recursive
*ra*b0*	$\vec{\mathbf{r}}^{A*B_0}$		$\langle\langle \vec{\mathbf{r}}^{A*B_0} \rangle\rangle$
	$\vec{\omega}^{B'}$	$\langle\langle \vec{\omega}^{B} \bullet \overleftrightarrow{\mathbf{b}} \rangle\rangle$	$\langle\langle \vec{\omega}^{B} \bullet \overleftrightarrow{\mathbf{b}} \rangle\rangle$
*holo-v*is-ab*	$\vec{\mathbf{v}}_i^{A*B*}$		$\begin{cases} \vec{\mathbf{r}}_{tj}^{B} & \text{for } i = o+j,\ j=1, \ldots N_{td}^{B} \\ \langle\langle \vec{\omega}_i^{A} \bullet \overleftrightarrow{\mathbf{a}} \rangle\rangle \times \vec{\mathbf{r}}^{A*B_0} & \\ + \langle\langle \vec{\omega}_i^{B} \bullet \overleftrightarrow{\mathbf{b}} \rangle\rangle \times \vec{\mathbf{r}}^{B_0B*} & \text{otherwise} \end{cases}$
*holo-v*is*	$\vec{\mathbf{v}}_i^{B*}$	$\begin{cases} \vec{\mathbf{b}}_{i-o} & i \text{ is translation d.o.f. of B} \\ 0 & \text{otherwise} \end{cases}$	$\langle\langle \left(\vec{\mathbf{v}}_i^{A*} + \vec{\mathbf{v}}_i^{A*B*}\right) \bullet \overleftrightarrow{\mathbf{b}} \rangle\rangle$
acc-dyadic	$\overleftrightarrow{\mathbf{a}}_{rot}^{B}$	$\langle\langle \left(\vec{\alpha}_{rem}^{B} \bullet \overleftrightarrow{\mathbf{b}}\right) \times \overleftrightarrow{\mathbf{b}}$ $+ \vec{\omega}^{B'} \times \left(\vec{\omega}^{B'} \times \overleftrightarrow{\mathbf{b}}\right) \rangle\rangle$	$\langle\langle \left(\vec{\alpha}_{rem}^{B} \bullet \overleftrightarrow{\mathbf{b}}\right) \times \overleftrightarrow{\mathbf{b}}$ $+ \vec{\omega}^{B'} \times \left(\vec{\omega}^{B'} \times \overleftrightarrow{\mathbf{b}}\right) \rangle\rangle$
acc-ab	$\vec{\mathbf{a}}_{rem}^{A*B*}$		$\overleftrightarrow{\mathbf{a}}_{rot}^{A} \bullet \vec{\mathbf{r}}^{A*B_0} + \overleftrightarrow{\mathbf{a}}_{rot}^{B} \bullet \vec{\mathbf{r}}^{B_0B*}$ $+ 2\sum_{i=1}^{N_{td}^{B}} u_{o+j} \left(\langle\langle \vec{\omega}^{A} \bullet \overleftrightarrow{\mathbf{a}} \rangle\rangle \times \vec{\mathbf{r}}_{tj}^{B}\right)$
acc-rem	$\vec{\mathbf{a}}_{rem}^{B}$	$\langle\langle \vec{\omega}^{B'} \times \langle\langle \sum_{i=1}^{\nu} \vec{\mathbf{v}}_i^{B*} \rangle\rangle$ $- \vec{\mathbf{g}} \bullet \overleftrightarrow{\mathbf{b}} \rangle\rangle$	$\langle\langle \left(\vec{\mathbf{a}}_{rem}^{A*} + \vec{\mathbf{a}}_{rem}^{A*B*}\right) \bullet \overleftrightarrow{\mathbf{b}} \rangle\rangle$

Note: If B is fixed, the origin (B_0) is used for B*. If A is fixed, the origin (A_0) is used for A*

nonholonomic acceleration remainders (angular and central). The equations from Sections 6.2 and 6.3 are repeated here for reference:

$$\tilde{\vec{\omega}}_r^{B} = \vec{\omega}_r^{B} + \sum_{s=p+1}^{\nu} A_{sr}\, \vec{\omega}_s^{B} \quad (r = 1, \ldots p) \tag{8.4.45}$$

$$\tilde{\vec{\mathbf{v}}}_r^{B*} = \vec{\mathbf{v}}_r^{B*} + \sum_{s=p+1}^{\nu} A_{sr}\, \vec{\mathbf{v}}_s^{B*} \quad (r = 1, \ldots p) \tag{8.4.46}$$

$$\tilde{\vec{\alpha}}_{rem}^{B} = \vec{\alpha}_{rem}^{B} + \sum_{s=p+1}^{\nu} \vec{\omega}_s^{B}\, c_s \tag{8.4.47}$$

$$\vec{\tilde{\mathbf{a}}}^{B*}_{rem} = \vec{\mathbf{a}}^{B*}_{rem} + \sum_{s=p+1}^{v} \vec{\mathbf{v}}^{B*}_{s}\, c_s \tag{8.4.48}$$

2. The mass matrix is formed:

$$M_{ij} = \sum_{B}^{\text{all bodies}} \left(\vec{\tilde{\omega}}^{B}_{j} \bullet \left(\overleftrightarrow{\mathbf{I}}^{B*} \bullet \vec{\tilde{\omega}}^{B}_{i} \right) + \begin{cases} \vec{\tilde{\mathbf{v}}}^{B*}_{j} \bullet \left(\vec{\tilde{\mathbf{v}}}^{B*}_{i}\, m^B \right) \\ \textit{or} \\ \widehat{\vec{\tilde{\mathbf{v}}}^{B*}_{j}} \bullet \widehat{\vec{\tilde{\mathbf{v}}}^{B*}_{i}}\, m^B \end{cases} \right) \tag{8.4.49}$$

Two strategies are shown above for introducing intermediate variables for the second term, depending on the bodies referenced in the *nonholo-v-bodies* slot of B. If either of the bodies is B, indicating that the partial velocities $\vec{\tilde{\mathbf{v}}}^{B*}_{i}$ or $\vec{\tilde{\mathbf{v}}}^{B*}_{j}$ for B is different than the corresponding partial velocity for A, than the upper strategy is taken. However, if both bodies from the *nonholo-v-bodies* slot are not B, then the dot product of the two partial velocities was used when processing the parent. The lower strategy causes the dot product obtained before to be used again. Note that the caret symbol "^" appears over the partial velocities, indicating that they are taken from the body in which they were introduced.

3. The force array is formed:

$$f_i = \sum_{B}^{\text{all bodies}} \left\{ \begin{array}{l} \left(\sum_{t=1}^{N_{B,T}} \vec{\mathbf{T}}^{B}_{t} \right) \bullet \vec{\tilde{\omega}}^{B}_{i} - \left(\vec{\alpha}^{B}_{rem} \bullet \overleftrightarrow{\mathbf{I}}^{B*} + \vec{\omega}^{B} \times \overleftrightarrow{\mathbf{I}}^{B*} \bullet \vec{\omega}^{B} \right) \bullet \vec{\tilde{\omega}}^{B}_{i} \\ \quad + \left(\sum_{f=1}^{N_{B,F}} \vec{\mathbf{F}}^{B}_{f} \right) \bullet \widehat{\vec{\tilde{\mathbf{v}}}^{B*}_{i}} - \vec{\mathbf{a}}^{B*}_{rem} \bullet \left(\vec{\tilde{\mathbf{v}}}^{B*}_{i}\, m^B \right) \end{array} \right\} \tag{8.4.50}$$

The summation of moments is performed by going through the list of all `moment` objects in the system, inspecting the *body1* and *body2* slots to see if either contains B. If B is in the *body1* slot, the moment is applied with a positive magnitude. If B is in the *body2* slot, the moment is applied with a negative magnitude. The same thing is done with a list of all forces in the system. However, if B matches one of the slots in the `force` object, the force is accounted for (1) in translation by direct inclusion in the force summation, and (2) in rotation, by taking the moment of the force about the mass center of B, defined as

$$\vec{\mathbf{T}} = \vec{\mathbf{r}}^{B*P} \times \vec{\mathbf{F}} \tag{8.4.51}$$

where $\vec{\mathbf{T}}$ is the torque of the moment couple, $\vec{\mathbf{r}}^{B*P}$ is a position vector going from the mass center of B to the `point` from the *point1* slot of the `force` object, and $\vec{\mathbf{F}}$ is the product of the magnitude (from the *exp* slot) and the direction (from the *dir* slot) of the force, with the appropriate sign (positive if B was in the *body1* slot, negative if B was in the *body2* slot.) Note that the partial velocity dotted with the applied forces is expressed in its original basis (as indicated with the caret), which is either B or a body up the tree from B.

After traversing the tree and performing the above three tasks, a complete set of implicit dynamical equations exists of the form

$$\underline{M}\,\dot{\mathbf{u}} = \underline{f} \tag{8.4.52}$$

The symbolic method presented in Chapter 7 is used to uncouple these equations.

8.5 Write Fortran Program

Upon completion of the analysis of the multibody system, the equations of motion are stored in several `eqs` objects. The variables that will appear in simulation code are contained in `declaration` objects. (Each `declaration` contains (1) a list of variable names, (2) a data type such as REAL or INTEGER, and (3) a subroutine in the simulation code that will be written such as DIFEQN or OUTPUT in which the variables are used.) As bodies, forces, constraint equations, auxiliary subroutines, and output variables were entered by the analyst, the arguments were scanned and all symbols contained in expressions were added to a list assigned to a Lisp global variable. Thus, all of the information needed to write a complete simulation code is available.

Before the program is written, the equations are inspected to determine which variables and parameters are actually needed to compute (1) derivatives in the equations of motion, and (2) output variables. First, all symbols (`sym`s and `indexed-sym`s) pertaining to the multibody system are "hidden" by setting the *hide* slot to the value 0. Then, the `validate-exp` function defined in Section 5.4 is applied to (1) the output variables, (2) the arguments of external subroutines introduced by the analyst, (3) derivatives of the generalized coordinates, and (4) derivatives of the independent speeds. Through recursion, `validate-exp` encounters every expression that contributes to the above four groups of values that must be computed in the simulation code. When `validate-exp` encounters a `sym` or `indexed-sym`, it increments the value in the *hide* slot. All symbols with a value of zero in the *hide* slot were not needed, and will not be included in the Fortran program.

Next, the equations are inspected recursively a second time, to search for intermediate variables that are used but once. These are `indexed-sym` objects with the *symbol* slot set to "Z" and the *hide* slot set to 1. When such an object is encountered, the expression containing the `indexed-sym` is modified. The `indexed-sym` is expanded by replacing it with the expression it originally replaced. (That expression is obtained from its *exp* slot.) Also, the *hide* slot of the `indexed-sym` is set to zero, so that it will not appear in the Fortran program. This expansion process is also recursive, to ensure that all "Z" inermediate variables that appear only once are removed.

After the above analysis of the program code, the equations of motion for the multibody system are stored such that only expressions that are proven to be necessary are printed.

The list of symbols is inspected. A `sym` is identified as a required parameter if its *hide* slots is not zero and the *const-or-var* slot is set to the symbol `const`.

The above validation and expansion activities are performed automatically upon completion of the dynamics analysis. The analyst can then view the equations of motion, the required parameters, and the constants that are precomputed. To generate a simulation code, the function `write-sim` is invoked. This function generates a completely self-contained Fortran program whose general design was presented in Chapter 4. It generates source code by four techniques:

1. conventional "write" statements are used to print strings containing lines of source code (in Lisp, the `format` function is used),
2. special write functions are used to print commonly occuring statements, such as comments, subroutine declarations, and END statements,
3. data objects that represent simulation code (e.g., `eqs` and `declaration` objects) are simply printed, and
4. existing text files are merged into the source code. About 600 lines of code, spread over nineteen files, are copied into the appropriate parts of the simulation code generated by AUTOSIM. These files are commonly called "include files."

Appendices B and C contain simulation codes generated by AUTOSIM, which can be studied by the interested reader.

Although the simulation code is presently generated in the Fortran language, the same basic method would be used to generate code in other languages, such as C, ADA, ADSIM, etc. To generate code in a different target language, it is necessary to change the above four techniques as follows:

1. the strings printed by Lisp `format` statements must be changed to equivalent statements in the new target language,
2. the special write functions must be extended to print analogous statements in the target language (for example, the function `write-comment` would be modified to precede each line with the comment character of the target language, rather than the letter “C” as is done in Fortran),
3. the print functions for the AUTOSIM data objects must be modified so that the objects are printed in the syntax of the target language, and
4. text files corresponding to the existing Fortran “include files” must be written in the target language.

The bulk of the simulation code is generated by the second and third techniques. Thus, most of the simulation code can be generated in a different language just by changing a few selected print functions.

8.6 Summary

The entire analysis of the multibody system is performed in the five steps that have just been detailed. The process is now summarized to put into perspective the procedures and rules that have been presented.

1. *Describe System.* The analyst describes the objects comprising the multibody system using a small set of AUTOSIM macros. As each rigid body is introduced by the analyst, a `body` object is automatically created and the following steps are performed:
    a. symbols are created for generalized coordinates and speeds.
    b. a coordinate system with three unit-vectors is defined for the body.
    c. a direction cosine matrix is generated (eqs. 8.1.4 — 8.1.11).

d. the body is classified for rotational analyses as (i) nonrecursive if it has three rotational degrees of freedom, (ii) a rotor if it satisfies critera described in Section 8.1, or (iii) general recursive in all other cases.

e. the body is classified for translational analyses as (i) nonrecursive if it has three translational degrees of freedom, or two translational degrees of freedom and eq. 8.1.13 is satisfied, (ii) "fixed" if the (composite) mass is fixed in the coordinate system of the parent, or (iii) general recursive in all other cases.

f. an analysis is conducted to create "composite bodies" that include the masses of bodies classified as "fixed masses." In this analysis, the inertia dyadic, the (composite) mass, and the (composite) mass center of each body "up" the tree from the new body is established (eqs. 8.1.14 — 8.1.18).

g. an expression is derived for the rotational velocity of the body (eq. 8.1.19).

h. an expression is derived for the absolute velocity of the origin (eq. 8.1.20). Because the speeds are sometimes defined relative to the mass centers, which were possibly modified in step (f), this analysis is re-applied to every body in the system.

Points of interest on rigid bodies are identified by the analyst, and corresponding `point` objects are created by the `add-point` macro.

Active forces and moments are described, and corresponding `force` and `moment` objects are created. A gravitational field can be defined. (Gravity is added at this stage simply by setting a global called `*acceleration-due-to-gravity*`.)

Additional equations are generated for nonholonomic constraints and closed kinematical loops using the `add-constraint` and `no-movement` macros. As each constraint is added, the state variables are searched for the "best" independent variable to eliminate and one is selected automatically. An expression is derived to replace the selected variable. Speeds are usually converted from "independent" to "nonholonomic" categories. Coordinates are usually converted from "independent" to "computed" categories. Details are provided at the beginning of Section 8.3 (eqs. 8.3.1 through 8.3.6, and eqs. 8.3.13 through 8.3.16).

Output variables, auxiliary variables, and external subroutines are also described by the analyst. A system of units can be selected, and default numerical values can be provided for any parameters that appear in expressions.

2. *Kinematical Analysis.* After the system has been specified, implicit kinematical equations are formed (eqs. 8.2.1 through 8.2.12). They are solved symbolically to obtain explicit expressions for the derivatives of the generalized coordinates (eqs. 7.1.5 through 7.1.9). The explicit equations are put into an `eqs` structure.

3. *Constraint Analysis.* Three sets of scalar coefficients are derived from expressions formulated for the nonholonomic speeds in step 1 (eqs. 8.3.7 and 8.3.8).

4. *Dynamics Analysis.* Terms needed for Kane's equations are formulated and kept in worksheet objects associated with each `body`. The following steps are performed:

    a. an array of ν holonomic partial angular velocities is formed for each body. The elements are defined according to the formulations in Table 8.4.2.

    b. the holonomic angular acceleration remainder is formed, again according to the formulations in Table 8.4.2.

    c. an array of ν holonomic partial central velocities is formed for each body. The elements are defined according to the formulations in Table 8.4.4. Also, an array of "native" bodies is defined that is used to determine the body in which the partial velocity was last changed.

    d. the holonomic central acceleration remainder is formed according to the formulations in Table 8.4.4.

    e. the nonholonomic partial angular and central velocities, and the nonholonomic angular and central acceleration remainders are formulated (eqs. 8.4.45 through 8.4.48).

    f. the mass matrix is formed (eq. 8.4.49)

    g. the forces and moments acting on each body are added, and terms involving the acceleration remainders are subtracted. The results are dotted with partial velocities to form the force array (eq. 8.4.50)

    h. the mass matrix is inspected for zeros, and the independent speeds are ordered. The mass matrix, the force array, and the independent speeds are permuted as described in Section 7.2. Then, the simultaneous equations are

solved symbolically to obtain explicit expressions for the derivatives of the independent speeds (eqs. 7.1.5 through 7.1.9). The explicit equations are put into an `eqs` structure.

5. *Write Fortran Program*. The equations are inspected to determine the parameters needed to describe the system. Also, `eqs` objects with the equations of motion are manipulated as described in Section 5.3 and 8.5 to remove unnecessary code. Finally, a complete simulation code is written in Fortran that (1) reads input parameters, (2) simulates the multibody system, and (3) generates an output file with predicted time histories of output variables. The equations of motions are written into the simulation code by printing the `eqs` structures with the kinematical and dynamical equations.

9. EXAMPLES

This Chapter presents analyses of six multibody systems to illustrate how the methods developed in Chapters 5 through 8 are applied using the AUTOSIM software. Also, several of the examples were used to help validate the correctness of the equations and to compare the numerical efficiency of the equations generated by AUTOSIM with equations obtained by other methods.

The first three examples cover three types of forces and constraints. Section 9.1 discusses a three-dimensional vehicle handling model, consisting of two rigid bodies and forces and moments due to gravity, tires, and the vehicle suspension. This system is a simple example of the sort of ground vehicle model that motivated this work, and it is described in great detail. Section 9.2 presents the analysis of a system subject to extensive nonholonomic constraints. It is a cart whose wheels are subject to the constraints of no-rolling and no-slip. Section 9.3 illustrates how a closed kinematical loop is treated for a system similar to an automotive suspension. It is a four-bar linkage and a strut spring-damper element. These three examples are systems that have been analyzed before, either by hand or through the use of generalized numerical simulation codes. Prior to this work, however, such analyses have not been possible with automated symbolic multibody programs.

Three other examples are included for systems that can also be analyzed by existing symbolic multibody programs, and have been. They are provided to compare the efficiency of the simulation code generated by AUTOSIM with codes generated by alternate methods. Two of these are spacecraft vehicles and the third is a robot manipulator.

Each section is organized as follows: first, the model is described. Second, the AUTOSIM inputs necessary to obtain a simulation code are presented. Third, results are shown, in the form of time history plots and summaries of the computation needed to solve the equations of motion. Finally, selected portions of the analysis are presented to illustrate the methods developed in previous chapters.

A short summary of the AUTOSIM commands is provided in Appendix A. It may be helpful in understanding the AUTOSIM inputs that are listed in the examples. Also,

complete Fortran listings for the systems discussed in sections 9.1 and 9.3 are provided in Appendices B and C.

9.1. Passenger Car Handling Model

The model developed here has been used for over 30 years to simulate automobile handling response to driver steer inputs. Although the model is relatively simple, it has been shown to predict steering responses that closely match measurements from the test track [114]. For this example, the objective of the simulation is to obtain time histories of the yaw rate and the lateral acceleration of the body mass center in response to a step change in the steer angle of the front wheels.

The Vehicle Model

When driving an automobile on a smooth surface at a constant speed, motions of the sprung mass (the vehicle body) can be described fairly completely with a "roll axis" concept. Both the front and rear suspensions possess a *suspension roll axis*, defined as the axis about which the unsprung mass rotates when it is subjected to a torque about a longitudinal axis. Further, a *roll center* for the suspension is defined as the intersection of the suspension roll axis with the vertical plane through the centers of the two wheels on either side of the car. Finally, a *vehicle roll axis* is defined by connecting the roll centers of the front and rear suspension. A side force applied to the body of the car along the roll axis causes no body roll. When the sprung mass is subjected to a side force that does not pass through the roll axis, it rolls about that axis. This concept is illustrated in Figure 9.1.1.

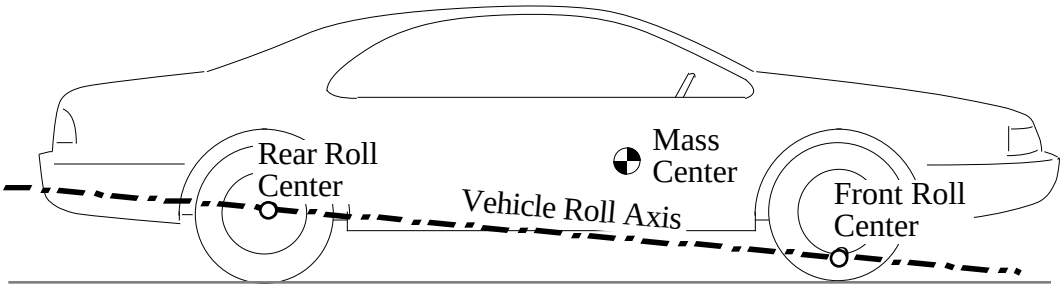

Figure 9.1.1. Roll axis in a passenger car.

Vehicles with rear axles and independent front suspensions generally have roll axes that run approximately through the center of the rear axle (a foot or so above the ground), and through a point near the ground at the mid-point of the front wheels. That is, the roll axis is tilted slightly down going from the rear to the front.

With a roll-axis model, all of the suspension properties are lumped into just a few parameters, namely,

1. inclination angle of the roll axis (determined by link lengths and locations),
2. torsional stiffness that resists roll of the sprung mass relative to the wheels (determined by suspension spring elements and their locations), and
3. torsional damping of roll motions (determined by shock absorber elements, friction elements, and the locations of such elements).

The above parameters can be computed from a detailed description of the suspension kinematics and the locations of the springs and dampers. Alternatively, they are often measured directly with special laboratory facilities that permit the "ground" to be rolled with respect to the vehicle body [133].

This vehicle model includes two rigid bodies: a sprung mass and an unsprung mass. The sprung mass includes the vehicle body and drive train, portions of the suspensions, and a portion of the front wheels. The unsprung mass includes the rear wheels, portions of the rear suspension, and portions of the front suspension and front wheels.

The vehicle responds to forces and moments generated by deforming the tires. The tire deformation is characterized by two variables: slip angle (α) and inclination angle (γ), also called camber. Both are shown in Figure 9.1.2 based on definitions established by the Society of Automotive Engineers (SAE) [1]

A simple tire model, valid for small deflections that occur under normal highway driving, defines side force (F_y) and aligning moment (M_z) as follows:

$$F_{yf} = C_{\alpha f}\,\alpha_f + C_{\gamma f}\,\gamma_f \qquad F_{yr} = C_{\alpha r}\,\alpha_r \qquad (9.1.1)$$

$$M_{zf} = C_{Mf}\,\alpha_f \qquad M_{zr} = C_{Mr}\,\alpha_r \qquad (9.1.2)$$

In the above equations, the subscripts f and r indicate front and rear tires, and the coefficients ($C_{\alpha f}$, $C_{\gamma f,}$ etc.) are summed for two tires on the left- and right-hand sides of the vehicle. No camber effect is shown for the rear, because the camber is negligible for a vehicle with a solid rear axle. The slip angle at the front includes a steer angle that is the "input" control to the system. The slip angle at the rear includes a steer proportional to the vehicle roll, as defined by a linear "roll-steer" coefficient determined by suspension kinematics. Similarly, the camber angle at the front is proportional to the vehicle roll, with a "camber roll coefficient" also determined by suspension kinematics.

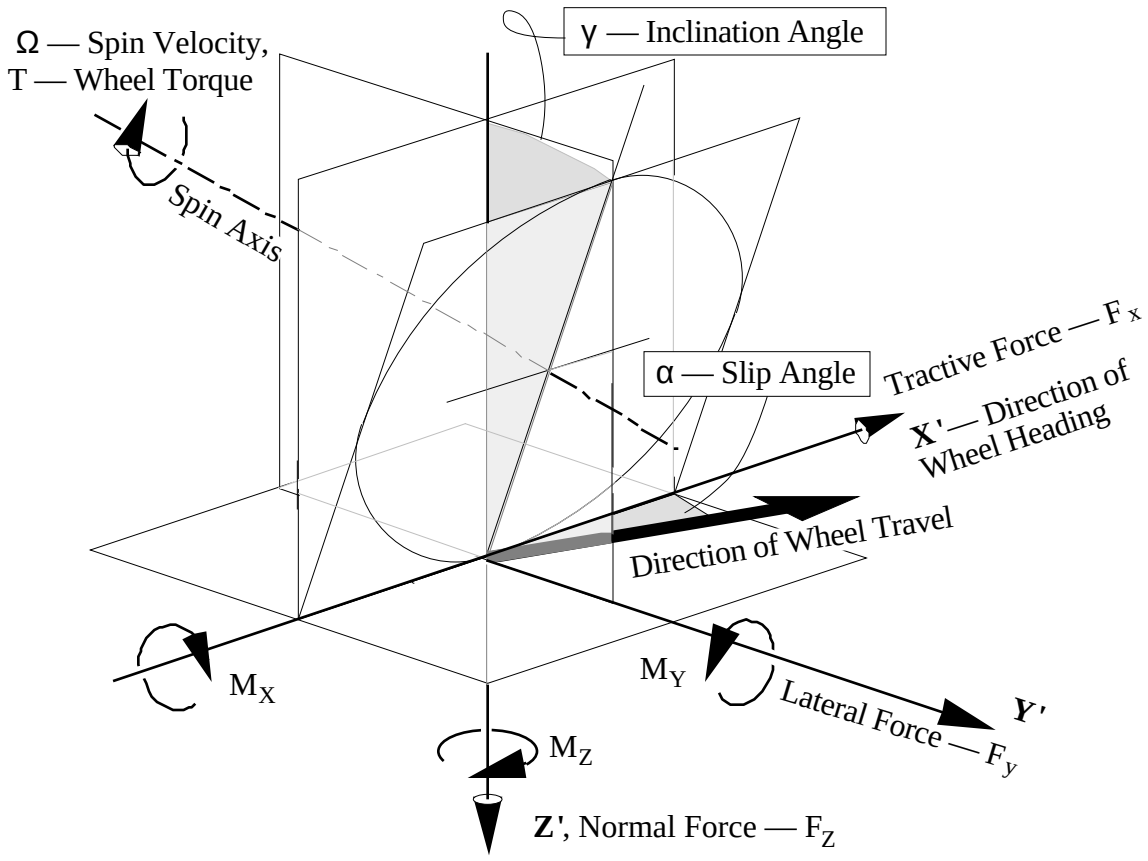

Figure 9.1.2. Tire geometry.

Gravity acts on the sprung mass, causing an overturning moment due to the lateral movement of the mass center when the sprung mass rolls. A restoring moment is generated by the suspension springs about the roll axis. Roll motions are damped by the shock absorbers, which also apply a moment about the roll axis.

The vehicle model concept has now been developed. To generate a simulation code, this concept is described to the AUTOSIM program so the appropriate data objects are created to represent the system in the computer.

AUTOSIM Inputs

The description of the system in this case includes both (1) the multibody system, and (2) the specific output variables of interest.

The Multibody System

This multibody system is comprised of three bodies: the inertial reference N, a non-rolling body NRB, and a rolling body RB. Points and coordinate systems for the system are indicated in Figure 9.1.3. The coordinate system of the inertial reference has its origin in the plane of the road, with axes defined according to the SAE convention [1]. The X, Y, and Z axis directions are defined by the unit-vectors [n1], [n2], and [n3], where the unit-vector [n3] points down[1].

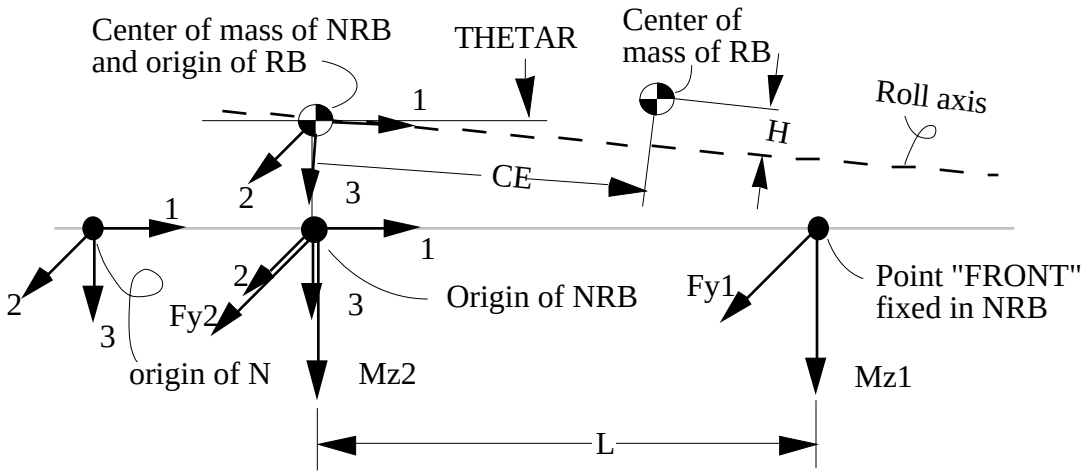

Figure 9.1.3. Points and dimensions for example vehicle model.

A direct description of the model is provided as an AUTOSIM input in Figure 9.1.4. The listing in this figure will be discussed at length over the next several pages.

Each input in the figure is a Lisp “form,” following the rules and syntax of Lisp, as summarized in Appendix A.

At the start of an AUTOSIM analysis session, the multibody system is composed of a single `body` object N, the inertial reference, and a single `point o`, the origin of the coordinate system of N. The inputs in the figure “build” the computer representation of the multibody system, adding one component at a time with macros such as `add-body`, `add-point`, `add-line-force`, and `add-moment`.

[1] In AUTOSIM unit-vectors are written enclosed with square brackets.

```
(add-body nrb :name "non-rolling body"
          :translate (1 2)
          :parent-rotation-axis 3
          :cm-coordinates #(0 0 !"-hra"))

(add-body rb :parent nrb :name "rolling body"
          :body-rotation-axes 1
          :parent-rotation-axis #(!"cos(thetar)" 0 !"sin(thetar)")
          :joint-coordinates #(0 0 !"-hra")
          :cm-coordinates #(ce 0 !"-h")
          :inertia-matrix #2a((Ixx 0  Ixz)
                              (0  Iyy   0)
                              (Ixz 0 Izzr)))

(add-constraint !"dot(vel(nrb0),[nrb1]) - speed")

(add-point front :name "front axle point"
           :body nrb
           :coordinates #(L 0 0))

(setf roll !"angle([nrb2],[rb2],[rb1])")
(setf alphaf !"angle([nrb1], vel(front), [nrb3]) - steer")
(setf alphar !"angle([nrb1], vel(nrb0), [nrb3]) - krs2 * #roll")

(add-gravity)

(add-line-force fy1 :name "Side force, front axle"
                :magnitude !"CA1 * #alphaf + CG1 * CCOEF1 * #roll"
                :point1 front
                :direction [nrb2])

(add-line-force fy2 :name "Side force, rear axle"
                :magnitude !"CA2 * #alphar"
                :point1 nrb0 :direction [nrb2])

(add-moment mz1 :name "Aligning moment, front axle"
            :direction [n3]
            :magnitude !"CAM1 * #alphaf" :body1 nrb)

(add-moment mz2 :name "Aligning moment, rear axle"
            :direction [n3]
            :magnitude !"CAM2 * #alphar" :body1 nrb)

(add-moment rollm :name "roll moment from suspension"
      :direction [rb1]
      :magnitude
        !"-Kroll * #roll - Croll* dot([nrb1], (rot(rb) - rot(nrb)))"
      :body1 rb :body2 nrb)
```

Figure 9.1.4. Description of car model in AUTOSIM.

The first `add-body` macro in Figure 9.1.4 describes several attributes of the first body added to the system. The arguments to the macro have the following meanings: (1) the symbol for the new body is NRB, (2) a more descriptive name to use in documentation is "non-rolling body," (3) NRB has two translational degrees of freedom relative to the

inertial reference, in the directions of axes 1 and 2 ([n1] and [n2]), (4) NRB has a single rotational degree of freedom about axis 3 ([n3]), and (5) the center of mass of NRB is a distance HRA above the ground. Default values are set for arguments not specified. For example, the parent body is set to N, as an alternative was not indicated. (Default values for all optional arguments are listed in Appendix A.)

Because the SAE axis convention specifies the Z axis ([n3]) pointing down, coordinates for heights above the ground are entered as negative expressions. This is not mandatory, but is done here so that the user of the simulation code will specify positive numbers for all dimensions. Expressions that are more complex than numbers or symbols are entered using a convention called the F-string. An F-string is an exclamation mark followed by a string containing a Fortran-style expression. (More information about the syntax is provided in Appendix A.) The negative symbol is an expression (a product of -1 and the symbol `hra`), and therefore an F-string was used in the `add-body` macro.

The second `add-body` macro in the input names the new body RB. Further, it indicates that (1) NRB is the parent body, (2) the descriptive name of RB is “rolling body,” (3) there is a single rotational degree of freedom, aligned with axis 1 of the coordinate system of RB, [rb1], (4) the rotation axis is oriented in the direction whose coordinates (in the frame of the parent NRB) are (COS(THETAR), 0, SIN(THETAR)) (that is, the vehicle roll axis is inclined down from axis 1 by an angle “THETAR” towards axis 3), (5) the origin of the coordinate system of RB is located at coordinates (0, 0, –HRA) in the coordinate system of NRB, (6) the center of mass is located at coordinates (CE, 0, –H) in the coordinate system of RB, and (7) the inertia matrix for RB is

$$\begin{bmatrix} \text{IXX} & 0 & \text{IXZ} \\ 0 & \text{IYY} & 0 \\ \text{IXZ} & 0 & \text{IZZR} \end{bmatrix}.$$

(The symbol for yaw inertia, IZZR, includes the letter ‘R’ to indicate that it applies to the rolling body.)

The macro `add-constraint` is used to specify that the forward speed is constant. Although we will see later that it is easy to find the name of the variable introduced by AUTOSIM for the forward speed, it is not necessary to have this information to specify the constraint. In the input, the velocity of the origin of body NRB is obtained with the function `vel(nrb0)`. The forward component is obtained by dotting the velocity with the forward direction of the vehicle, `[nrb1]`, with the expression

`dot(vel(nrb0),[nrb1])`. The macro `add-constraint` requires an expression that is constrained to be zero. Thus, the relationship

```
dot(vel(nrb0),[nrb1]) = speed
```
(9.1.3)

is expressed in the form

```
0 = dot(vel(nrb0),[nrb1]) – speed
```
(9.1.4)

for use with the `add-constraint` macro.

The macro `add-point` is used to define a point called `front` at which the front tire force is applied. (See Figure 9.1.3.)

Lisp symbols are used to store expressions developed for the roll angle (`roll`) and the slip angles for the front and rear axle centers (`alphaf` and `alphar`). The Lisp macro `setf` is used to assign expressions to the symbols, and the expressions are used in subsequent macros by preceding the symbol name with the character '#' (see Appendix A for details of the syntax). That is, when "`#roll`" appears in an expression, it indicates that we want to include the expression assigned to the Lisp symbol `roll` rather than a parameter called `roll`.

The front slip angle is defined as the angle between the velocity of a point where the wheel plane intersects the ground, and the angle of the wheel (see Figure 9.1.2). For this example, the angle between the velocity of a point and the forward direction of the vehicle is obtained with the `angle` function, and the steer angle of the wheel relative to the body is subtracted from that.

The macro `add-gravity` applies a gravitational force to all bodies that can move in the direction of the constant gravitational field, `[n3]`. (This direction is the default, but can be replaced for systems that do not follow the SAE convention.)

The macros `add-line-force` and `add-moment` are used to define tire forces and moments. The first `add-line-force` macro defines how a side force from the two front tires is generated and applied to the vehicle. The macro indicates that: (1) the force is called FY1, (2) the name is "Side force, front axle," (3) the magnitude of the force is specified with an F-string that closely matches the definition from eq. 9.1.1, (4) the line of action passes through the point `front` and acts on the body associated with that point, and (5) the direction of the force is [nrb2]. Because a second point was not provided as an optional argument, the force is assumed to act from N. Note that the F-string for the magnitude refers to the Lisp variables `alphaf` and `roll`, defined above.

The second `add-line-force` adds the side force at the rear axle and is very similar in form to the first `add-line-force` macro.

The first `add-moment` macro indicates that (1) the moment is called MZ1, (2) the name is "Aligning moment, front axle," (3) the moment is applied about the direction [n3][1], (4) the amplitude of the moment is the expression from eq. 9.1.2, and (5) the moment is applied to body NRB. Because the optional second body is not mentioned, the default N is assumed. The second `add-moment` macro is very similar to the first.

The third `add-moment` macro applies a sum of the moments generated by all of the suspension springs and dampers about the roll axis. Because this moment acts between two bodies, an optional argument is used to specify that the second body is NRB. Note that the expression for the amplitude includes a dot product written as: "`dot([nrb1], (rot(rb) - rot(nrb))`." This subexpression, which gives the rotation rate, might be written in conventional vector notation as:

$$\overrightarrow{\mathbf{nrb}}_1 \bullet \left(\vec{\omega}^{RB} - \vec{\omega}^{NRB}\right) \tag{9.1.5}$$

The description of the multibody system is now complete.

Small Terms

The formulation developed above for this vehicle model makes no use of engineering judgements regarding the significance of various terms in the equations. It will be shown later that the equations are much more complicated than they need to be, because some of the terms are always negligible. That is, terms caused by the nonlinearities contribute no insight to the system, nor do they improve the fidelity of the model. Given that a simple linear tire model is used, the vehicle model is valid only for moderate steering inputs, resulting in lateral acceleration levels of 0.3 g's or less. Also, we are interested mainly in highway speeds. This means that the contribution of the forward speed to the velocities of points in the system is much greater than contributions from any other speed variables, such as lateral speed, yaw rate, or roll rate. In other words, the yaw rate and lateral velocity are "small" with respect to the forward speed, even thought the yaw angle and X

---

[1] One could also identify the vertical direction as [nrb3]. When a direction is used to define axes of more than one coordinate system (e.g., N and NRB), AUTOSIM recognizes alternate names for the associated `uv`.

and Y position variables are not small. The roll angle is limited to a few degrees, which is also "small."

Figure 9.1.5 shows how the input is modified to declare that the above variables are "small." When the rolling body is added, an additional keyword is used to specify that the rotational degree of freedom involves a small angle. This instructs AUTOSIM to declare the associated generalized coordinate and generalized speed variables as small. Also, an additional input is used to declare that the yaw rate and the side velocity are small. The declaration is made this way, rather than with optional keywords in `add-body` (as was done for the roll angle), because the yaw angle is not small, nor is the generalized coordinate associated with the Y-position of the vehicle.

```
(add-body rb :parent nrb :name "rolling body"
          :body-rotation-axes 1
          :parent-rotation-axis #(!"cos(thetar)" 0 !"sin(thetar)")
          :joint-coordinates #(0 0 !"-hra")
          :cm-coordinates #(ce 0 !"-h")
          :inertia-matrix #2a((Ixx 0  Ixz)
                              (0  Iyy   0)
                              (Ixz 0 Izzr))
          :small-angles (t))

(small (dot (rot nrb) '[n3]) (dot (vel nrb0) '[nrb2]))
```

Figure 9.1.5. Inputs for "small" variables.

Specification of Output Variables

Before developing equations of motion, the analyst should define output variables of interest. (After all, the purpose of the simulation code is to compute output variables.) Using the F-string and the various AUTOSIM algebra functions, it is possible to define almost any position or motion variable of interest with a macro called `add-out`. The analyst has at his or her disposal all of the points introduced automatically, any points added by the analyst, and directions defined in all of the coordinate systems of the system. In this example, output variables are defined for lateral acceleration, yaw rate, front and rear slip angles, and all forces and moments.

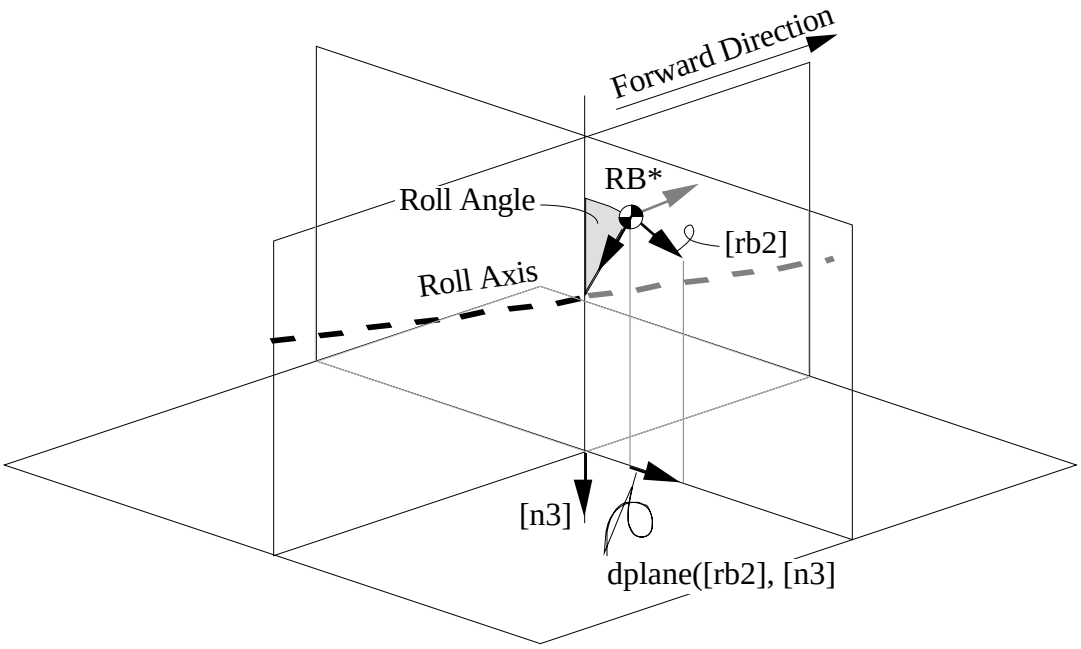

Figure 9.1.6. Definition of direction for lateral acceleration.

The lateral acceleration computed by the simulation code should match the measurement made with an accelerometer on a "stabilized platform." The instrument is mounted on a platform that is kept level using servomotors controlled with gyroscopic sensors, so that lateral acceleration measurements are made without the influence of gravity. This means that the lateral acceleration is not in the direction of the body-fixed coordinate system [rb2] (the body rolls such that the dot product between [rb2] and [n3] is not zero). The lateral direction is shown in Figure 9.1.6, and is defined mathematically in the first input Lisp form in Figure 9.1.7. The written description is interpreted as follows: (1) the laterally-oriented unit-vector fixed in the body ([rb2]) is projected onto the ground plane (perpendicular to [n3]), using the `dplane` function, and (2) the direction of that projection is obtained with the `dir` function. That direction, lying parallel to the ground plane, is dotted with the acceleration vector for the sprung mass to obtain the scalar acceleration variable that is given the short name "Ay" and the long name "Lateral Acceleration." Also, the body associated with the variable is RB and the units of the variable are "length/time^2" (entered as "`L/T**2`").

```
(add-out !"dot(dir(dplane([rb2],[n3])), dxdt(vel(rbcm)))"
         "Ay"
         :long-name "Lateral Acceleration"
         :body rb :units !"l/t**2")

(add-out !"dot(rot(nrb), [n3])" "r"
         :long-name "Yaw Rate" :body nrb :units !"a/t")

(add-out alphaf "alpha f"
         :long-name "Front slip angle"
         :body nrb
         :units a
         :gen-name "Slip Angle")

(add-out alphar "alpha r"
         :long-name "Rear slip angle"
         :body nrb
         :units a
         :gen-name "Slip Angle")

(add-forces-to-output)
(add-moments-to-output)
```

Figure 9.1.7. Specification of output variables.

The yaw rate is defined as the dot product of the rotation of NRB and the unit-vector [n3]. The slip angles had previously been assigned to the Lisp symbols `alphaf` and `alphar`, so they are easily added. The forces and moments are added with the macros `add-forces-to-output` and `add-moments-to-output`.

Now that the output variables have been defined, the analysis is performed with the form `(dynamics)` and a simulation code is generated with the form `(write-sim)`. (These last two macros are not shown in the figures.)

Parameter Identification

At the end of the analysis, each symbol that has been introduced, either explicitly by the analyst, or automatically by one of the macros, is checked to see if it actually appears in the equations of motion in the simulation code. Those symbols that do appear are considered parameters if they are known to be constants. (Symbols are known to be constants if they were not introduced by AUTOSIM as state variables and were not declared by the analyst with the `add-variables` macro.) Values are needed for all parameters in order for the simulation code to run. Accordingly, an INPUT subroutine is generated in Fortran to read values for these parameters when the simulation code is executed. Also, an ECHO subroutine is created in Fortran so that values that applied for the run can be written as an output file when the simulation is run. The list of input parameters obtained by this process

can be viewed by the analyst to help confirm that the system is properly described. The list for the example is shown in Table 9.1.1.

Note that some of the symbols that were shown in earlier printouts do not appear. For example, only one of the inertia symbols introduced for NRB (NRBI33) has any influence on the dynamic behavior of the system. Also, the height of the roll axis, HRA, has no effect and is left out.

Most of the units were correctly determined. The exceptions were two coefficients that were multiplied together: CG1 and CCOEF1. Although the units of their product can be deduced (lb/deg), there is not enough information provided to determine the units of the individual coefficients. (GC1 normally has units of lb/deg and CCOEF is dimensionless.)

Because the parameters are the main interface to the end user, it is important that they be familiar if the simulation code is to be "easy" to use. Additional inputs to AUTOSIM used to fine-tune the parameters in the system are shown in Figure 9.1.8.

Macros `set-units` and `set-name` were developed to override the default units and names. Also, alternate units systems can be set with functions `in-lb` and `mks`.

The simulation code is written such that all parameters have default values. Those parameters not mentioned by name in the input file are left at their default values. Appropriate default values can be specified by the analyst for each parameter with the macro `set-defaults`. A user not familiar with the model can use the simulation code without specifying parameter values if all of the parameters have been assigned reasonable default values by the analyst with the `set-defaults` macro. The echo file produced by the simulation code shows all of the parameters, their numerical values, their names, and their units. The echo file created with the default values used for the simulation results shown above is listed in Appendix B along with the complete source code.

Table 9.1.1. Parameters identified for the car model, with names and units deduced from context.

Parameter	Definition
CA1:	coefficient in term in negative Side force, front axle (lb/deg)
CA2:	coefficient in Side force, rear axle (lb/deg)
CAM1:	coefficient in Aligning moment, front axle (in-lb/deg)
CAM2:	coefficient in Aligning moment, rear axle (in-lb/deg)
CCOEF1:	coefficient in term in negative Side force, front axle (?)
CE:	coordinate of center of mass of the rolling body in dir 1 (in)
CG1:	coefficient in term in negative Side force, front axle (?)
CROLL:	coefficient in term in negative roll moment from suspension (in-lb-s/d)
H:	negative coordinate of center of mass of the rolling body in dir 3 (in)
IPRINT:	number of time steps between output printing (counts)
IXX:	moment of inertia of RB (in-lb-s2)
IXZ:	product of inertia of RB (in-lb-s2)
IYY:	moment of inertia of RB (in-lb-s2)
IZZR:	moment of inertia of RB (in-lb-s2)
KROLL:	coefficient in term in negative roll moment from suspension (in-lb/deg)
KRS2:	coefficient in term in coefficient in Aligning moment, rear axle (-)
L:	coordinate of front axle point in dir 1 (in)
NRBI33:	moment of inertia of NRB (in-lb-s2)
NRBM:	mass of NRB (lbm)
RBM:	mass of RB (lbm)
SPEED:	argument to ATAN in term in coefficient in Aligning moment, rear axle (in/s)
STEER:	term in coefficient in Aligning moment, front axle (deg)
STEP:	simulation time step (sec)
STOPT:	simulation stop time (sec)
THETAR:	angle in parent-rot axis for RB, coord #3 (deg)

```
(setf *multibody-system-name* "Example no. 1" )

(in-lb)

(set-units ca1 !"f/a" ca2 !"f/a" cam1 !"l*f/a" cam2 !"l*f/a" krs2 1
           thetar a steer a Kroll !"l*f/a" croll !"t*L*f/a" cg1 !"f/a"
           ccoef1 1 speed !"l/t")

(set-defaults ca1 -444 ca2 -428 cam1 1080 cam2 1000 cg1 78 ce 63.4
              krs2 -.016 ixx 5580 iyy 12000 izzr 37080 izznr 1285
              ixz 0 KROLL 6211 croll 212 ccoef1 .82  h 15.48 l 125.5
              nrbm 704 rbm 3831 speed 968 steer 1 thetar 5.1
              step .025 iprint 2 stopt 2)

(set-name ca1 "front cornering stiffness"
          ca2 "rear cornering stiffness"
          ce "distance from rear axle to sprung mass c.g."
          ccoef1 "prop. of body roll resulting in front wheel camber"
          cam1 "front aligning moment coefficient"
          cam2 "rear aligning moment coefficient"
          cg1 "front camber stiffness"
          croll "torsional damping rate for the vehicle body in roll"
          h "height of sprung mass c.g. above roll axis"
          kroll "torsional spring rate for the vehicle body in roll"
          krs2 "roll-steer coefficient for rear axle"
          L "wheelbase"
          steer "Steer angle at road"
          thetar "inclination angle of roll axis"
          speed "forward speed")
```

Figure 9.1.8. Inputs to specify characteristics of system parameters.

Results

Two versions of the model were developed earlier: a full nonlinear model, and one in which most of the variables were identified by the analyst as "small." The equations obtained with "small" variables were compared to equations obtained manually and were found to agree. Numerical results from both versions are shown to agree.

Numerical Results

Time histories from the simulation codes are compared in Figures 9.1.9 and 9.1.10. They show that the small angle assumptions have a negligible effect on the predicted responses to a step steer input of 1.0 degree. The complete simulation code for the system analyzed with small variables is included in Appendix B.

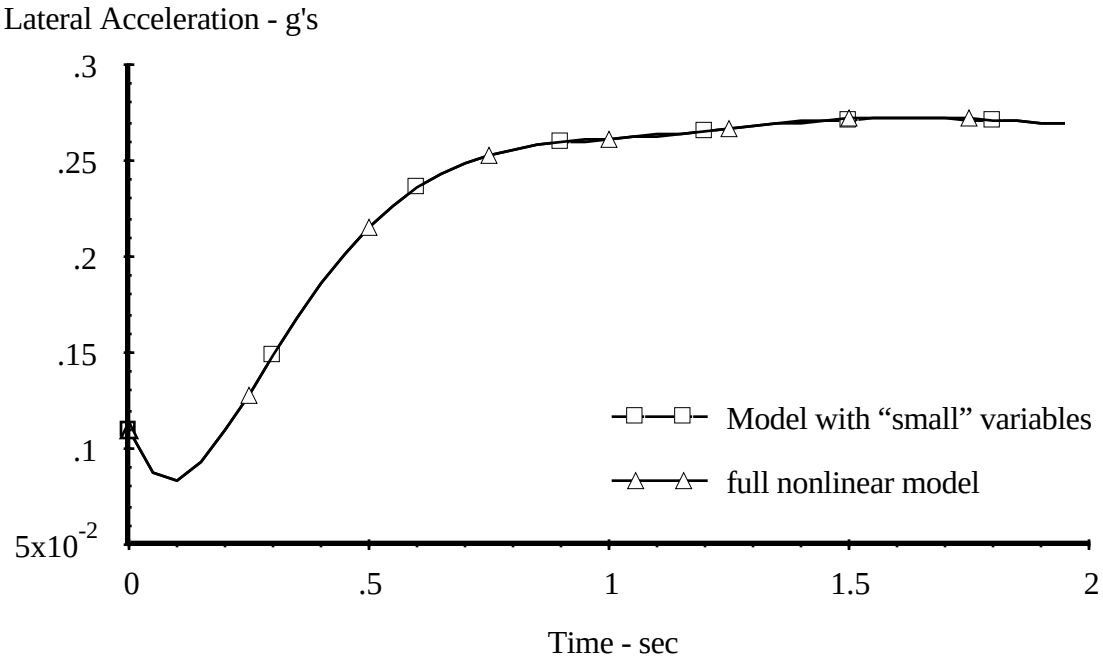

Figure 9.1.9. Step responses of two models in lateral acceleration.

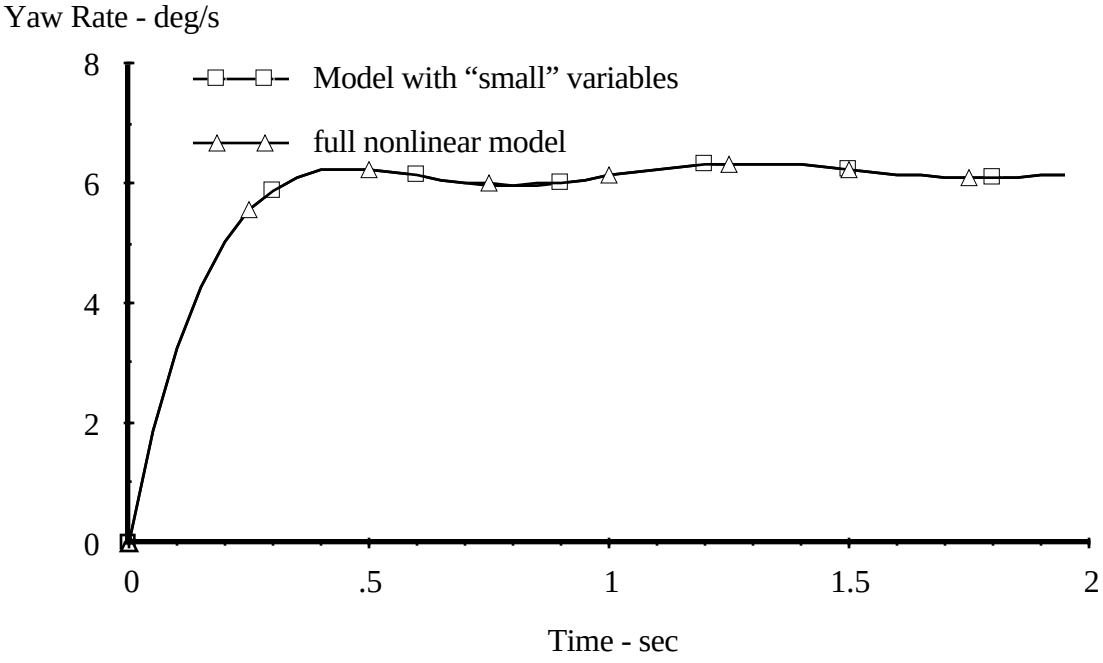

Figure 9.1.10. Step responses of two models in yaw rate.

The output files created by the simulation codes are in a format called “ERD files” that are used at UMTRI [102]. Automated post-processing software is available, including a plotter that performs scaling and labeling automatically and which has a graphical interface

that permits the engineer to select variables for plotting simply by clicking a mouse [105]. For example, Figure 9.1.11 shows the screen display when the channels are selected.

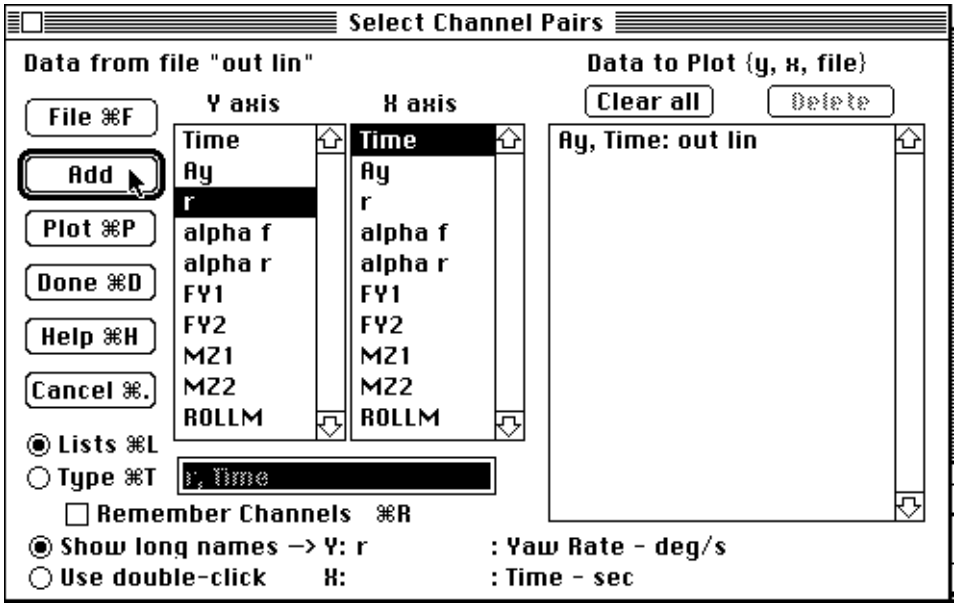

Figure 9.1.11. Use of automated plotter to view simulation results.

Computational Efficiency

Two simulations were developed above: (1) with the full, nonlinear representation of the rigid-body kinematics, and (2) with several speeds and one angle declared "small." The number of arithmetic operations in the Fortran code generated by AUTOSIM to compute the derivatives of the state variables for each case is shown in Table 9.1.2. Also, a third case is shown in which additional settings were made that disabled the automatic introduction of new symbols for each force and moment, and for the intermediate Z variables. For this simple system, the best efficiency was obtained without the use of these intermediate variables. Code generated in this case has been reported elsewhere [103]. (The improvement arises because it is possible to combine many of the terms contained in the FORCEM and Z expressions. For more complicated systems, this is generally not true.)

Table 9.1.2. Performance comparisons between three simulation codes.

Version of simulation code	adds and subtracts	multiplies, divides, and function calls
Full, nonlinear simulation.	67	108
Simulation with "small" variables.	24	36
Most efficient, with no FORCEM or Z arrays.	15	19

The effects of some of the basic algebraic simplification methods built into AUTOSIM have also been explored with this model. It was found that when no intermediate variables were introduced at all, that about three times as many arithmetic operations were required. Further, when the naturally factored form of the AUTOSIM expressions was changed to the expanded form that has been used in the NEWEUL formulation, a total of 878 multiplications, divides, and function calls were required. The difference between the "best" and "worst" cases is nearly a factor of 50 [103].

Analysis Details

Now that the routine use of AUTOSIM to generate simulation code has been seen, details of the analysis are presented that may or may not be of interest to the analyst.

At any stage of the analysis, the computer representation of the multibody system can be inspected by the analyst. Printouts obtained in this way are shown below to illustrate the types of expressions that are introduced and manipulated as the automated analysis progresses.

The `body` object created to represent the non-rolling body is printed in Table 9.1.3, showing the values associated with some of the slots. Most of the values were obtained by the first `add-body` macro.

Generalized coordinates and generalized speeds were introduced, a direction cosine matrix was developed, and expressions were obtained for the rotational velocity and the velocity of the origin of the coordinate system. Because this body has one rotational degree of freedom, one of the unit-vectors ([n3]) is from the parent body. Note that several symbols were generated automatically. Lacking any information about mass and inertia, the `add-body` macro introduced the symbol NRBM for the mass of the body, and six symbols (NRBI11, NRBI12, NRBI13, NRBI22, NRBI23, and NRBI33) for the moments and products of inertia. Those are used, in turn, to build an expression for the inertia

dyadic. The printout used to prepare the table was generated after the next body was introduced, as evidenced by the list (RB) in the *children* slot.

Table 9.1.3. Data associated with slots of body NRB.

Summary of body:	NRB
parent:	N
level:	1
children:	(RB)
name:	Non-Rolling Body
mass:	NRBM
inertia:	(NRBI33*([N3].[N3]) + NRBI13*([NRB1].[N3]) + NRBI13*([N3].[NRB1]) + NRBI23*([N3].[NRB2]) + NRBI23*([NRB2].[N3]) + NRBI11*([NRB1].[NRB1]) + NRBI12*([NRB2].[NRB1]) + NRBI12*([NRB1].[NRB2]) + NRBI22*([NRB2].[NRB2]))
unit-vectors:	#([NRB1] [NRB2] [N3])
translation-coordinates:	(Q(1) Q(2))
translation-speeds:	(U(1) U(2))
rotation-coordinates:	(Q(3))
rotation-speeds:	(U(3))
rotation-directions:	([N3])
translation-directions:	([N1] [N2])
joint-pos:	(Point O: Body N: #(0 0 0): fixed origin)
cm-pos:	(Point NRBCM: Body NRB: #(0 0 -HRA): center of mass of the non-rolling body)
abs-w:	U(3)*[N3]
abs-vj:	(U(1)*[NRB1] + U(2)*[NRB2])
cos matrix:	#(COS(Q(3)) SIN(Q(3)) 0) #(-SIN(Q(3)) COS(Q(3)) 0) #(0 0 1.0)

The definitions of the state variables can be printed for inspection by the analyst. The summaries printed by AUTOSIM are shown in Table 9.1.4. Note that both names and units were generated for all of the variables. The equations of motion are derived for any

set of units in which conversions are not needed to apply kinematic analysis or Newton's laws. Thus, angles in the equations necessarily have units of radians. However, if the units system chosen by the analyst involves units that require conversions, such as for angles (deg), mass (lbm), acceleration (g's), and so forth, the simulation code generated by AUTOSIM performs the necessary conversions when input data are read and when output data are written. Thus, from the perspective of the end user, the units of the variables and parameters are those shown in listing such as Table 9.1.4. (The units conversions can be found in the subroutines INPUT, ECHO, and OUTPUT, listed in Appendix B.)

Table 9.1.4. Printed summary of state variables.

Generalized Coordinates
Q(1): Translation of NRB relative to the fixed origin along [n1]. (in)
Q(2): Translation of NRB relative to the fixed origin along [n2]. (in)
Q(3): Rotation of the non-rolling body relative to the inertial reference about axis #3. (deg)
Q(4): Rotation of the rolling body relative to the non-rolling body about axis #1. (deg)

Generalized Speeds (before `add-constraint` macro is used)
U(1): Abs. trans. speed of NRB along axis 1. (in/s)
U(2): Abs. trans. speed of NRB along axis 2. (in/s)
U(3): Abs. rotation of NRB, axis 3. (deg/s)
U(4): Rotation of RB relative to NRB, axis 1. (deg/s)

The generalized speeds U(1) and U(2) are not derivatives of the generalized coordinates, but are instead defined as "quasi-coordinates" parallel to body axes. This is according to the rules established in Section 8.1 and 8.4 for bodies with two translational degrees of freedom that are constrained to planar motions.

Table 9.1.4 was obtained after the second body was added, but before the constraint was defined. After the constraint is added, the system has only three degrees of freedom.

The macro `add-constraint` removes a dynamical degree of freedom by changing slot values in the `indexed-sym` object that represents a generalized speed, and then renumbering the remaining speeds. In the example, the forward vehicle speed, initially printed as "U(1)" is selected as the best generalized speed to remove. The macro solves for U(1) and determines that the constraint is satisfied when the U(1) is replaced with the parameter `speed`. Accordingly, it changes the *const-or-var* slot to `const`, the *dxdt* slot to

0, the *exp* slot to `speed`, and the *i* slot to 0. The renumbering is performed by changing the *i* slot in all `indexed-sym` objects that represent generalized speeds. After renumbering, the speeds appear as shown in Table 9.1.5. The table also includes a summary of the nonholonomic constraint equations (there is but one in this example).

Table 9.1.5. Summary of generalized speeds after constraint is added.

Generalized Speeds (After `add-constraint` macro is used)
U(1): Abs. trans. speed of NRB along axis 2. (in/s)
U(2): Abs. rotation of NRB, axis 3. (deg/s)
U(3): Rotation of RB relative to NRB, axis 1. (deg/s)
Constraints
Abs. trans. speed of NRB along axis 1.: SPEED

Printing of expressions is performed recursively, with every type of object having an associated print function. If an object is changed such that it prints differently, all expressions containing that object will also print with the "updated" form. Thus, all expressions that contain the generalized speed originally named "U(2)" will now print that object as "U(1)."

Because AUTOSIM freely renames objects, the analyst must be careful when referring to state variables by name. The possibility of naming the wrong variable can be eliminated by referring to the variable as an expression involving positions, angle, velocities, and rotational velocities of bodies and points in the system, as was done in this example.

After the forces and moments are entered, they can be viewed also. Table 9.1.6 shows the summary of the forces printed by AUTOSIM. (The equations are shown later.)

Once the system is described to AUTOSIM, the equations of motion are derived by a function named `dynamics`. The analysis proceeds automatically as follows.

First, the size of the system is determined so that matrices can be introduced to store indexed variables such as partial velocities and constraint coefficients. The kinematical equations are derived by the method presented in Section 8.2. Next, constraint coefficients are computed. For this example, these analyses are very simple and not discussed further.

(The most complicated kinematical equations are generated in the two spacecraft models. The constraint analysis is examined in detail for the four-bar linkage example.)

The dynamical analysis is performed in several stages. The tree is traversed from top to bottom so that expressions can be derived and put into worksheets associated with each body.

Table 9.1.6. Listing of forces and moments.

Forces
(RBW: gravity force on the rolling body: Expression = RBM*GEES: Direction = [n3]. Acts on the rolling body from the inertial reference through center of mass of the rolling body)
(FY1: Side force, front axle: Expression = FORCEM(1): Direction = [nrb2]. Acts on the non-rolling body from the inertial reference through front axle point)
(FY2: Side force, rear axle: Expression = FORCEM(2): Direction = [nrb2]. Acts on the non-rolling body from the inertial reference through coord. origin of the non-rolling body)
Moments
(MZ1: Aligning moment, front axle: Expression = -FORCEM(3): Direction = [n3]. Acts on the non-rolling body from the inertial reference)
(MZ2: Aligning moment, rear axle: Expression = FORCEM(4): Direction = [n3]. Acts on the non-rolling body from the inertial reference)
(ROLLM: roll moment from suspension: Expression = -FORCEM(5): Direction = [rb1]. Acts on the rolling body from the non-rolling body)

Tables 9.1.7 and 9.1.8 show the contents of the worksheets created for two of the body objects in this example. (Body N also has a worksheet in which all expressions are zero.)

The slots in the tables are defined in Section 8.4 for the worksheet object. All of the expressions in the worksheet are either vectors or dyadics, as indicated by the presence of unit-vectors.

Note that there are four holonomic partial velocities and three nonholonomic partial velocities. Recall that in Kane's convention for manually analyzing nonholonomic systems, independent speeds are numbered from 1 to *p* and dependent speeds are numbered from *p+1* to *n*. However, the holonomic arrays retain the ordering they had before any nonholonomic constraints were applied, which is generally not in accordance with Kane's convention. Numbering of dependent speeds is of no importance because the

dependent speeds, having been replaced by expressions involving independent speeds, do not appear anywhere in the equations.

Table 9.1.7. Dynamics worksheet for the non-rolling body.

Worksheet for body:	NRB
recursive-r:	T
recursive-t:	NIL
w:	U(2)*[N3]
wis-a array:	(0, 0, 0, 0)
wis-ab array:	(0, [N3], 0, 0)
wis array:	(0, [N3], 0, 0)
nhwis array:	(0, [N3], 0)
alpha-rem:	0
alpha-ab:	0
nh-alpha-rem:	0
*ra*b0:*	0
*v*is array:*	([NRB2], 0, 0, [NRB1])
*v*is bodies:*	(NRB, NRB, NRB, NRB)
*nhv*is array:*	([NRB2], 0, 0)
*nhv*is bodies:*	(NRB, NRB, NRB)
acc-rem:	(-Z(18)*[NRB1] + Z(19)*[NRB2])
nh-acc-rem:	(-Z(18)*[NRB1] + Z(19)*[NRB2])
acc-dyadic:	-(U(2)**2*([NRB2].[NRB2]) + U(2)**2*([NRB1].[NRB1]))

In this example, the dependent speed was originally named U(1). Hence, the first element in each of the holonomic arrays corresponds to this speed. In this example, the nonholonomic partial velocities are identical to the holonomic equivalents, except that elements corresponding to the dependent speed are eliminated.

Note that intermediate variables appear in many of the expressions. During the dynamics analysis, intermediate variables are introduced liberally to prevent the expressions from growing too large. (The intermediate variables and constants are defined later.)

Table 9.1.8. Dynamics worksheet for the rolling body.

Worksheet for body:	RB
recursive-r:	T
recursive-t:	T
w:	(Z(7)*[RB2] + Z(8)*[RB3] + Z(12)*[RB1])
wis-a array:	(0, (PC(4)*[RB1] + Z(10)*[RB2] + Z(11)*[RB3]), 0, 0)
wis-ab array:	(0, 0, [RB1], 0)
wis array:	(0, (PC(4)*[RB1] + Z(10)*[RB2] + Z(11)*[RB3]), [RB1], 0)
nhwis array:	(0, (PC(4)*[RB1] + Z(10)*[RB2] + Z(11)*[RB3]), [RB1])
alpha-rem:	(Z(16)*[RB2] -Z(17)*[RB3])
alpha-ab:	(-U(3)*Z(7)*[RB3] + U(3)*Z(8)*[RB2])
nh-alpha-rem:	(Z(16)*[RB2] -Z(17)*[RB3])
*ra*b0:*	0
*v*is-a array:*	([NRB2], 0, 0, [NRB1])
*v*is-ab array:*	(0, (-H*Z(10)*[RB1] -CE*Z(10)*[RB3] + (PC(5) +
:	CE*Z(11))*[RB2]), H*[RB2], 0)
*v*is array:*	((C(4)*[RB2] -S(4)*[RB3]), (Z(13)*[RB2] -Z(14)*[RB3] -Z(15)*[RB1]), H*[RB2], (PC(2)*[RB1] - PC(4)*S(4)*[RB2] -PC(4)*C(4)*[RB3]))
*v*is bodies:*	(NRB, RB, RB, NRB)
*nhv*is array:*	((C(4)*[RB2] -S(4)*[RB3]), (Z(13)*[RB2] -Z(14)*[RB3] -Z(15)*[RB1]), H*[RB2])
*nhv*is bodies:*	(NRB, RB, RB)
acc-rem:	(Z(23)*[RB2] + Z(24)*[RB3] -(PC(2)*Z(18) + H*(Z(16) + Z(20)) + CE*(Z(22) + Z(8)**2))*[RB1])
acc-ab:	-(H*(Z(16) + Z(20))*[RB1] -CE*-(Z(16) -Z(20))*[RB3] - CE*(Z(7)*Z(12) -Z(17))*[RB2] + H*Z(7)*Z(8)*[RB2] + CE*(Z(22) + Z(8)**2)*[RB1] -H*(Z(21) + Z(22))*[RB3])
nh-acc-rem:	(Z(23)*[RB2] + Z(24)*[RB3] -(PC(2)*Z(18) + H*(Z(16) + Z(20)) + CE*(Z(22) + Z(8)**2))*[RB1])

After all of the slots in the worksheet objects are set, the mass matrix and the force array are created and filled with zeros. Then, the tree is traversed one more time and the contribution from each body is added to the arrays. After this traversal, the symbolic equation solver is employed to derive a series of equations that defines the derivatives of the generalized speeds.

The equations of motion for the nonlinear system are printed in Figures 9.1.12 through 9.1.14, exactly as generated by AUTOSIM. The Fortran code in Figure 9.1.12 is for precomputing constants. The code for computing derivatives of state variables is in Figures 9.1.13 and 9.1.14. (The corresponding code for the case in which some variables are "small" is in Appendix B.)

```
PC(1) = CG1*CCOEF1                  PC(13) = (RBM + NRBM)
PC(2) = COS(THETAR)                 PC(14) = H*RBM*GEES*COS(THETAR)
PC(3) = CROLL*COS(THETAR)           PC(15) = (RBM*H**2 + IXX)
PC(4) = SIN(THETAR)                 PC(16) = 1.0/PC(13)
PC(5) = H*SIN(THETAR)               PC(17) = PC(2)**2
PC(6) = RBM*GEES                    PC(18) = H*PC(17)
PC(7) = (IXX -IYY)                  PC(19) = CE*PC(2)
PC(8) = (IYY -IZZR)                 PC(20) = H*PC(2)
PC(9) = (IXX -IZZR)                 PC(21) = PC(11)*PC(4)
PC(10) = IXZ*SIN(THETAR)            PC(22) = (NRBI33 + PC(21))
PC(11) = IXX*SIN(THETAR)            PC(23) = PC(6)*PC(4)
PC(12) = H*RBM                      PC(24) = PC(6)*PC(2)
```

Figure 9.1.12. Fortran code for precomputing constants.

In these figures, most of the expressions involve either elements of the array Z or the array PC. The 'Z' variables are intermediate variables introduced for redundant variable expressions. The 'PC' variables are expressions involving constants that can be precomputed before the numerical integration loop is started.

Note that the solution for the accelerations (the variables in the Fortran array UP) are recursive. The equation for UP(1) includes Z(35) and Z(36), which are the values for UP(3) and UP(2).

Recall from Chapter 5 that all unused code is removed, and that every 'Z' variable is referenced at least twice. Z variables that appear but once are replaced with the original expressions.

```
C  Equations of motion, from subroutine DIFEQN
C
C  Each derivative evaluation requires 108 multiply/divides, 69
C  add/subtracts, and 6 function/subroutine calls.
C
      S(3) = SIN(Q(3))
      S(4) = SIN(Q(4))
      C(3) = COS(Q(3))
      C(4) = COS(Q(4))
C
C
C  Kinematical equations
C
      QP(1) = (SPEED*C(3) -U(1)*S(3))
      QP(2) = (U(1)*C(3) + SPEED*S(3))
      QP(3) = U(2)
      QP(4) = U(3)
C
C  define expression for Side force, front axle
C
      Z(1) = (STEER -ATAN2((L*U(2) + U(1)), SPEED))
      FORCEM(1) = (PC(1)*Q(4) -CA1*Z(1))
C
C  define expression for Side force, rear axle
C
      Z(2) = (-KRS2*Q(4) + ATAN2(U(1), SPEED))
      FORCEM(2) = CA2*Z(2)
C
C  define expression for Aligning moment, front axle
C
      FORCEM(3) = CAM1*Z(1)
C
C  define expression for Aligning moment, rear axle
C
      FORCEM(4) = CAM2*Z(2)
C
C  define expression for roll moment from suspension
C
      FORCEM(5) = (KROLL*Q(4) + PC(3)*U(3))
```

Figure 9.1.13. First part of Fortran code for computing derivatives of state variables.

```
C  Dynamical equations
C
     Z(3) = PC(2)*U(2)*S(4)
     Z(4) = PC(2)*U(2)*C(4)
     Z(5) = PC(4)*U(2)
     Z(6) = PC(2)*S(4)
     Z(7) = PC(2)*C(4)
     Z(8) = (U(3) + Z(5))
     Z(9) = (PC(5) + CE*Z(7))
     Z(10) = CE*Z(6)
     Z(11) = H*Z(6)
     Z(12) = U(3)*Z(4)
     Z(13) = U(3)*Z(3)
     Z(14) = U(2)*U(1)
     Z(15) = SPEED*U(2)
     Z(16) = Z(4)*Z(8)
     Z(17) = Z(8)**2
     Z(18) = Z(3)**2
     Z(19) = (-H*Z(3)*Z(4) + CE*(Z(3)*Z(8) -Z(13)) + Z(15)*C(4) +
    &         PC(4)*Z(14)*S(4))
     Z(20) = -(CE*(Z(12) -Z(16)) -H*(Z(17) + Z(18)) -PC(4)*Z(14)*C(4)
    &         + Z(15)*S(4))
     Z(21) = IXZ*Z(4)
     Z(22) = (Z(3)*(PC(8)*Z(4) -IXZ*Z(8)) + IXZ*Z(13))
     Z(23) = RBM*Z(11)
     Z(24) = RBM*Z(10)
     Z(25) = RBM*Z(9)
     Z(26) = IXZ*Z(7)
     Z(27) = (-NRBM*Z(15) + FORCEM(1) + FORCEM(2) -RBM*(Z(19)*C(4)
    &         -Z(20)*S(4)))
     Z(28) = (-PC(23)*Z(11) -Z(6)*(IYY*Z(12) -IXZ*Z(17) + Z(4)
    &        *(PC(9)*Z(8) + Z(21))) + Z(7)*(IZZR*Z(13) + Z(3)
    &        *(PC(7)*Z(8) + Z(21))) + PC(4)*Z(22) -(PC(2)*Z(14) + H
    &        *(Z(12) + Z(16)) + CE*(Z(18) + Z(4)**2))*Z(23) +
    &         Z(20)*Z(24) -Z(19)*Z(25) + L*FORCEM(1) -FORCEM(3) +
    &         FORCEM(4) + PC(24)*(-Z(10)*C(4) + Z(9)*S(4)))
     Z(29) = (Z(25)*C(4) + Z(24)*S(4))
     Z(30) = PC(12)*C(4)
     Z(31) = (PC(11) + PC(12)*Z(9) + Z(26))
     Z(32) = PC(16)*Z(29)
     Z(33) = PC(16)*Z(30)
     Z(34) = (PC(22) + IYY*Z(6)**2 + (PC(10) + IZZR*Z(7))*Z(7) +
    &         Z(11)*Z(23) + Z(10)*Z(24) + Z(9)*Z(25) + PC(4)*Z(26)
    &         -Z(29)*Z(32))
     Z(35) = (Z(31) -Z(29)*Z(33))/Z(34)
     Z(36) = (Z(31) -Z(30)*Z(32))
     Z(37) = Z(27)*Z(32)
     Z(38) = -(PC(12)*Z(19) -Z(22) + Z(27)*Z(33) + Z(35)*(Z(28)
    &         -Z(37)) + FORCEM(5) -PC(14)*S(4))/(PC(15) -Z(30)*Z(33)
    &         -Z(35)*Z(36))
     Z(39) = (Z(28) -Z(37) -Z(36)*Z(38))/Z(34)
     UP(3) = Z(38)
     UP(2) = Z(39)
     UP(1) = PC(16)*(Z(27) -Z(30)*Z(38) -Z(29)*Z(39))
```

Figure 9.1.14. Continuation of Fortran code for computing derivatives of state variables.

9.2 Four-Wheeled Cart

The example described in this section illustrates how typical nonholonomic constraints are handled. Also, it shows how masses and inertias are combined into composite bodies.

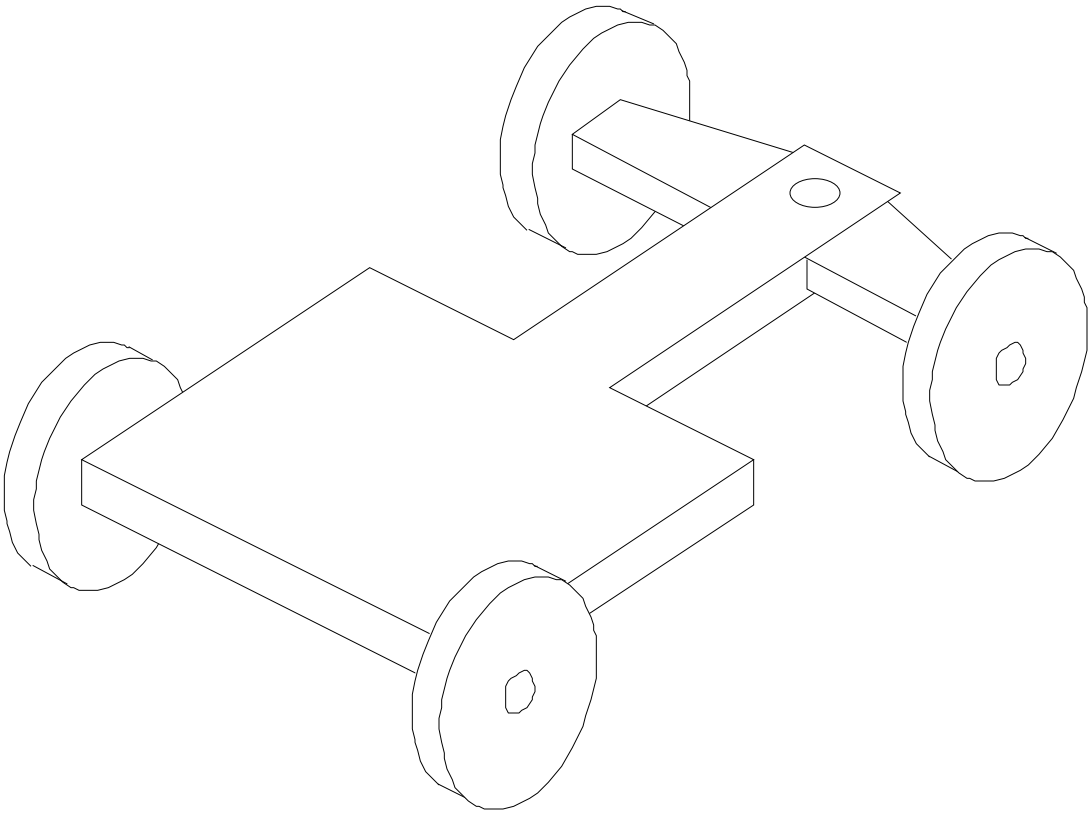

Figure 9.2.1. Four-wheeled cart.

Model Description

A cart with four wheels and a steered front axle is shown in Figure 9.2.1. Dimensions, bodies, and reference points are defined in Figure 9.2.2. The cart rolls without slipping on a smooth flat surface. The front axle steers about a point F_0 that is located slightly in front of the axle, and which is shown by a black dot in Figure 9.2.2. The cart is pushed from rest by a constant force applied to the point B_0, in a direction oriented along the longitudinal axis of the cart, [b1]. Given an initial steer angle (nominally, 0.25 radian), the objective of

the simulation is to study the motions of the cart over the first few seconds, to determine characteristics of the response.

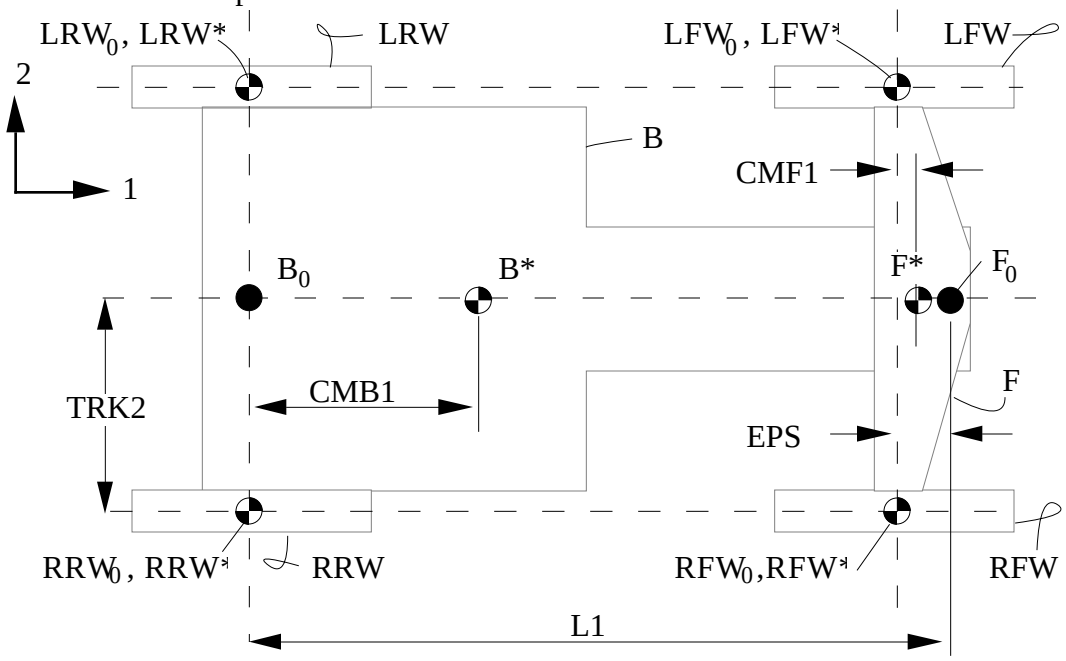

Figure 9.2.2. Bodies, reference points, and dimensions for cart.

The inspiration for this model is a "Rocket Car" system analyzed by Ge and Cheng [28] as an example of a system with a variable mass. The system shown is used in as an example in a course taught at The University of Michigan[1]. Equations of motion for the system as shown have been derived previously and used to compute time responses of some of the variables.

The constraints imposed by the condition that the wheels roll without slipping are described mathematically in two ways: (1) the forward velocity of each wheel center must equal the spin of the wheel multiplied by its radius, and (2) the lateral velocity of each wheel center is zero. That is, for arbitrary wheel W, whose center point is W*,

$$\vec{\mathbf{v}}^{W*} \bullet \vec{\mathbf{w}}_1 = \Omega R \tag{9.2.1}$$

where $\vec{\mathbf{w}}_1$ is a unit-vector oriented along the centerline of the wheel, Ω is the spin of the wheel, and R is its radius. Also,

$$\vec{\mathbf{v}}^{W*} \bullet \vec{\mathbf{w}}_2 = 0 \tag{9.2.2}$$

where $\vec{\mathbf{w}}_2$ is a unit-vector oriented along the spin axis.

---

[1] Course notes for "Computational Mechanics," Aero 541, taught by Prof. D. Greenwood.

AUTOSIM Description

The rigid-body information from Figure 9.2.2 is entered into AUTOSIM as shown in Figure 9.2.3.

The first Lisp form introduces the body B of the cart with two translational degrees of freedom and one rotation. That is, it is free to move in the plane normal to [n3]. The second form introduces the front axle F. Body B is its parent, and F is able to rotate about axis 3 relative to B. Note that because the dimensions in the figure are defined relative to the spin axis of the front wheels, rather than the pivot point for the front axis, the first coordinate for the center of mass is not a parameter, but the expression "`-eps + cmf1`."

```
(add-body b :name "body of cart" :translate (1 2)
          :parent-rotation-axis 3
          :cm-coordinates #(cmb1 0 0))

(add-body f :parent b
          :name "front axle"
          :parent-rotation-axis 3
          :cm-coordinates #(!"-eps + cmf1" 0 0)
          :joint-coordinates #(L1 0 0))

(add-body lrw :parent b :name "left-rear wheel"
          :parent-rotation-axis 2
          :joint-coordinates #(0 !"-trk2" 0)
          :inertia-matrix #(it ia it) :mass mw)

(add-body rrw :parent b :name "right-rear wheel"
          :parent-rotation-axis 2
          :joint-coordinates #(0 trk2 0)
          :inertia-matrix #(it ia it) :mass mw)

(add-body lfw :parent f :name "left-front wheel"
          :parent-rotation-axis 2
          :joint-coordinates #(!"-eps" !"-trk2" 0)
          :inertia-matrix #(it ia it) :mass mw)

(add-body rfw :parent f :name "right-front wheel"
          :parent-rotation-axis 2
          :joint-coordinates #(!"-eps" trk2 0)
          :inertia-matrix #(it ia it) :mass mw)

(add-line-force F :point1 b0 :direction [b1])
```

Figure 9.2.3. AUTOSIM description of cart example.

The next four entries define the four wheels. The rear two wheels have B as their parent body, and the front two wheels have F as their parent. Because all four wheels have the same mass and inertia properties, the mass and inertia matrices are explicitly identified

in the inputs. Also, note that the inertia properties are summarized by two moments of inertia: one about the spin axis (IA) and one about any axis normal to the spin axis (IT).

The force that pushes the cart is described very simply, since it is a constant F.[1]

Next, the nonholonomic constraints are entered, as shown in figure 9.2.4. The nonslipping condition for each wheel can be used to generate two constraint equations, one based on eq. 9.2.1 and one based on eq. 9.2.2. Thus, eight constraints are entered in the figure. (Two of these are redundant, however. If one wheel on an axle is constrained to have zero lateral velocity, the other wheel on that axle is also constrained. However, if the analyst does not realized this, an entry such as that shown in the figure can be handled. It will be seen later that the redundant constraints are ignored by AUTOSIM.)

```
;; constrain spin of wheels to define zero longitudinal slip

(add-constraint !"r*dot([b2], (rot(rrw) - rot(b)))
                 - dot(vel(rrw0),[b1])")
(add-constraint !"r*dot([b2], (rot(lrw) - rot(b)))
                 - dot(vel(lrw0),[b1])")
(add-constraint !"r*dot([f2], (rot(rfw) - rot(f)))
                 - dot(vel(rfw0),[f1])")
(add-constraint !"r*dot([f2], (rot(lfw) - rot(f)))
                 - dot(vel(lfw0),[f1])")

;; define zero lateral velocity for each wheel

(add-constraint !"dot([b2], vel(rrw0))")
(add-constraint !"dot([b2], vel(lrw0))")
(add-constraint !"dot([f2], vel(rfw0))")
(add-constraint !"dot([f2], vel(lfw0))")
```

Figure 9.2.4. AUTOSIM description of nonholonomic constraints for cart example.

The remainder of the AUTOSIM input, shown in Figure 9.2.5, defines the yaw rate of the body, the steer rate for the front axle, and all generalized coordinates as output variables to be generated by the simulation code. The figure also lists inputs that set the units system to be metric (mks), and default values for the parameters of the system.

---

[1] Although the same symbol (F) is used for both a force and a body, there is no conflict in AUTOSIM because the symbol for a body does not appear in the equations of motion.

```
;; define output variables

(add-out !"dot([n3], rot(b))" "r"
         :long-name "yaw rate" :body b :units !"a/t")
(add-out !"dot([n3], (rot(f) - rot(b)))" "deldot"
          :body f :long-name "steer rate"  :units !"a/t")
(add-coordinates-to-output)

(dynamics)
(MKS)
(setf *multibody-system-name* "Cart--Example #2")

(set-defaults BM 20 FM 5 BI33 12 FI33 .5 TRK2 .5 L1 1.5 CMB1 .5
              CMF1 .02 EPS .1 R .2 MW 1 IT .01 IA .02 F 100)
```

Figure 9.2.5. AUTOSIM description of cart output variables and parameter values.

Results

The complete list of output variables (obtained with the plotter [105]) is shown in Table 9.2.1.

Table 9.2.1. List of output channels generated by simulation code for cart.

```
  0 -         :  Time - sec
  1 - r       :  yaw rate - rad/s
  2 - deldot  :  steer rate - rad/s
  3 - Q(1)    :  Trans. of B0 rel. to O, dir=[n1] - m
  4 - Q(2)    :  Trans. of B0 rel. to O, dir=[n2] - m
  5 - Q(3)    :  Rot. of B rel. to N, axis #3 - rad
  6 - Q(4)    :  Rot. of F rel. to B, axis #3 - rad
  7 - Q(5)    :  Rot. of LFW rel. to F, axis #2 - rad
  8 - Q(6)    :  Rot. of RFW rel. to F, axis #2 - rad
  9 - Q(7)    :  Rot. of LRW rel. to B, axis #2 - rad
 10 - Q(8)    :  Rot. of RRW rel. to B, axis #2 - rad
```

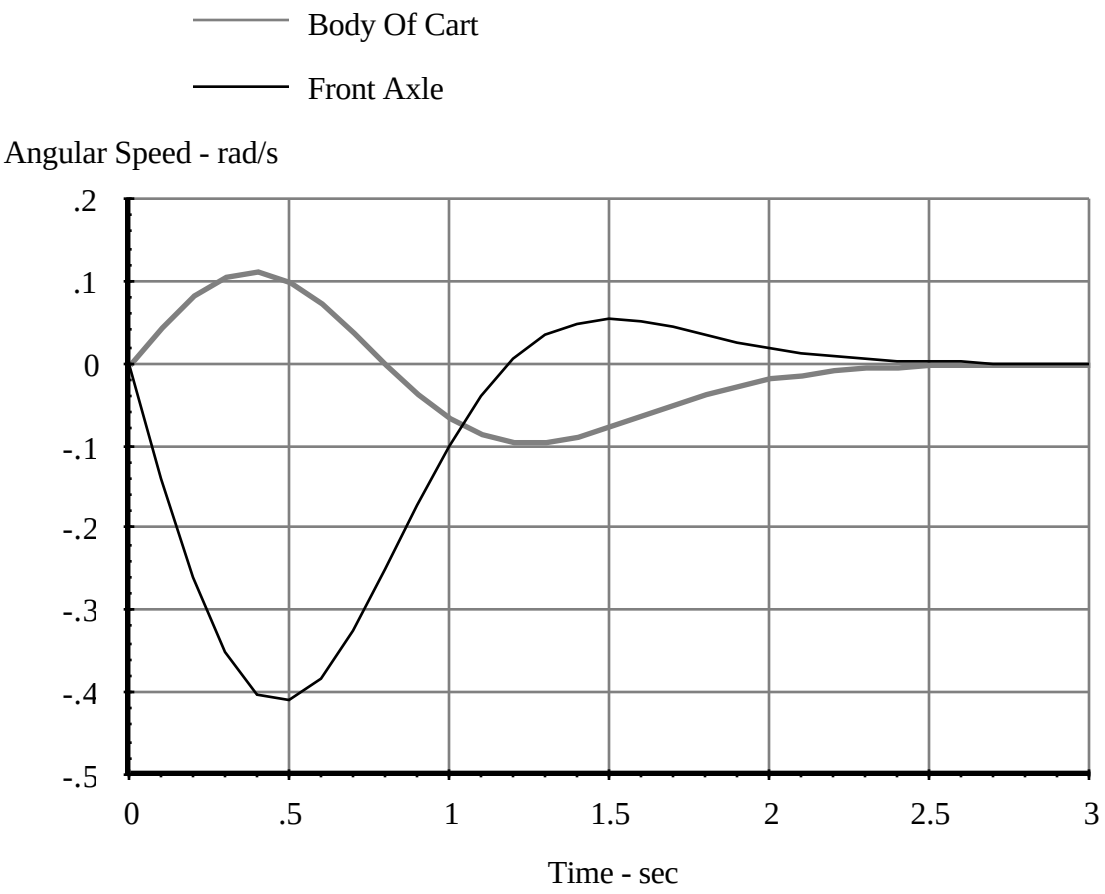

Figure 9.2.6. Transient responses of yaw rate and steer rate.

The time histories of the yaw rate and the steer rate are shown in Figure 9.2.6 From the list of output variables, the yaw angle and steer angle are seen to be named Q(3) and Q(4). The time histories for these variables are shown in Figure 9.2.7.

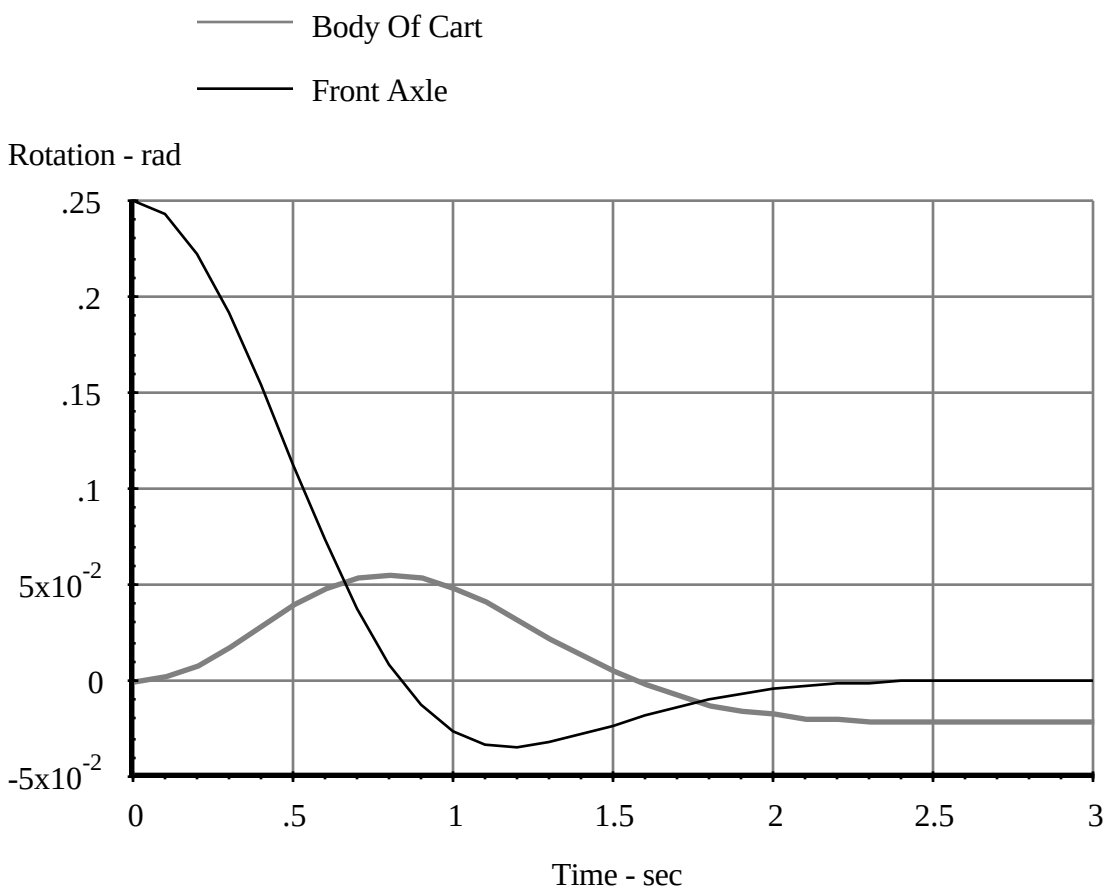

Figure 9.2.7. Transient responses of yaw angle and steer angle.

The above numerical results were compared with numerical results obtained using equations that were formulated manually, and were found to agree.

The complete parameter list, produced as an echo file by the simulation code, is shown in Table 9.2.2.

Table 9.2.2. Parameter values and initial conditions for cart.

* PARAMETER VALUES		
BI33	12.0000	moment of inertia of B (kg-m2)
BM	20.0000	mass of B (kg)
CMB1	.500000	coordinate of center of mass of the body of cart in dir 1 (m)
CMF1	0.200000E-01	negative term in negative coordinate of center of mass of the front axle in dir 1 (m)
EPS	.100000	negative coordinate of attachment point for the right-front wheel in dir 1 (m)
F	100.000	F (N)
FI33	.500000	moment of inertia of F (kg-m2)
FM	5.00000	mass of F (kg)
IA	0.200000E-01	moment of inertia of RFW (kg-m2)
IPRINT	5.00000	number of time steps between output printing (counts)
IT	0.100000E-01	moment of inertia of RFW (kg-m2)
L1	1.50000	coordinate of attachment point for the front axle in dir 1 (m)
MW	1.00000	mass of RFW (kg)
R	.200000	NIL (m)
STEP	0.200000E-01	simulation time step (sec)
STOPT	3.00000	simulation stop time (sec)
TRK2	.500000	coordinate of attachment point for the right-front wheel in dir 2 (m)
* INITIAL CONDITIONS		
Q(1)	.000000	Translation of B0 relative to the fixed origin along [n1]. (m)
Q(2)	.000000	Translation of B0 relative to the fixed origin along [n2]. (m)
Q(3)	.000000	Rotation of the body of cart relative to the inertial reference about axis #3. (rad)
Q(4)	.250000	Rotation of the front axle relative to the body of cart about axis #3. (rad)
Q(5)	.000000	Rotation of the left-front wheel relative to the front axle about axis #2. (rad)
Q(6)	.000000	Rotation of the right-front wheel relative to the front axle about axis #2. (rad)
Q(7)	.000000	Rotation of the left-rear wheel relative to the body of cart about axis #2. (rad)
Q(8)	.000000	Rotation of the right-rear wheel relative to the body of cart about axis #2. (rad)
U(1)	.000000	Abs. trans. speed of B* along axis 1. (m/s)
U(2)	.000000	Abs. trans. speed of B* along axis 2. (m/s)

Analysis Details

This system was included in part to show details in the analysis of a system with extensive nonholonomic constraints. Also, the dynamics analysis is of interest because significant modeling simplifications involving the wheels are possible.

The Constraint Analysis

Each `add-constraint` form nominally removes one degree of freedom by changing a generalized speed from an "independent speed" to a "nonholonomic speed." When a degree of freedom is removed, a description of the variable that is eliminated is printed on the screen together with the replacement expression. However, when the constraint is already identically zero, AUTOSIM merely prints a message to this effect. To show this, the `add-constraint` forms are shown again in Figure 9.2.8, along with the responses. The inputs are shown in boldface type and the AUTOSIM responses are shown in plainface.

Six of the `add-constraint` forms result in the removal of generalized speeds. The replacement expressions involve remaining independent speeds. In two cases, the constraint is already satisfied as a consequence of previously introduced constraints.

The current generalized speeds and nonholonomic constraint equations can be viewed at any stage of the analysis. Tables 9.2.3 through 9.2.5 list the speeds and constraints at three stages of the analysis: (1) before any constraints are removed, (2) after the first four constraints are added, and (3) after all constraints are added.

```
;; constrain spin of wheels to be zero-slip

(add-constraint !"r*dot([b2], (rot(rrw) - rot(b)))
                  - dot(vel(rrw0),[b1])")
Replace Rot. of RRW relative to B, axis 2.
-(TRK2*U(3) -U(1))/R

(add-constraint !"r*dot([b2], (rot(lrw) - rot(b)))
                  - dot(vel(lrw0),[b1])")
Replace Rot. of LRW relative to B, axis 2.
(TRK2*U(3) + U(1))/R

(add-constraint !"r*dot([f2], (rot(rfw) - rot(f)))
                  - dot(vel(rfw0),[f1])")
Replace Rot. of RFW relative to F, axis 2.
(-TRK2*(U(3) + U(4)) + U(1)*C(4) + ((L1 -CMB1)*U(3) + U(2))*S(4))/R

(add-constraint !"r*dot([f2], (rot(lfw) - rot(f)))
                  - dot(vel(lfw0),[f1])")
Replace Rot. of LFW relative to F, axis 2.
(TRK2*(U(3) + U(4)) + U(1)*C(4) + ((L1 -CMB1)*U(3) + U(2))*S(4))/R

;; define zero sideslip for axles

(add-constraint !"dot([b2], vel(rrw0))")
Replace Abs. rot. of B, axis 3.
U(2)/CMB1

(add-constraint !"dot([b2], vel(lrw0))")
"Constraint equation is already zero."

(add-constraint !"dot([f2], vel(rfw0))")
Replace Rot. of F relative to B, axis 3.
-(U(2)/CMB1 -(L1*U(2)*C(4)/CMB1 -U(1)*S(4))/EPS)

(add-constraint !"dot([f2], vel(lfw0))")
"Constraint equation is already zero."
```

Figure 9.2.8. AUTOSIM responses to constraint definitions.

Table 9.2.3. Generalized speeds before any constraints are added.

```
  U(1): Abs. trans. speed of B* along axis 1. (m/s)
  U(2): Abs. trans. speed of B* along axis 2. (m/s)
  U(3): Abs. rot. of B, axis 3. (rad/s)
  U(4): Rot. of F relative to B, axis 3. (rad/s)
  U(5): Rot. of LFW relative to F, axis 2. (rad/s)
  U(6): Rot. of RFW relative to F, axis 2. (rad/s)
  U(7): Rot. of LRW relative to B, axis 2. (rad/s)
  U(8): Rot. of RRW relative to B, axis 2. (rad/s)
```

Table 9.2.4. Generalized speeds and constraints, after four constraints are added.

```
  U(1): Abs. trans. speed of B* along axis 1. (m/s)
  U(2): Abs. trans. speed of B* along axis 2. (m/s)
  U(3): Abs. rot. of B, axis 3. (rad/s)
  U(4): Rot. of F relative to B, axis 3. (rad/s)

Rot. of LFW relative to F, axis 2.: (TRK2*(U(3) + U(4)) + U(1)*C(4)
        + ((L1 -CMB1)*U(3) + U(2))*S(4))/R
Rot. of RFW relative to F, axis 2.: (-TRK2*(U(3) + U(4)) + U(1)*C(4)
        + ((L1 -CMB1)*U(3) + U(2))*S(4))/R
Rot. of LRW relative to B, axis 2.: (TRK2*U(3) + U(1))/R
Rot. of RRW relative to B, axis 2.: -(TRK2*U(3) -U(1))/R
```

Table 9.2.5. Generalized speeds and constraints, after all constraints are added.

```
  U(1): Abs. trans. speed of B* along axis 1. (m/s)
  U(2): Abs. trans. speed of B* along axis 2. (m/s)

Abs. rot. of B, axis 3.: U(2)/CMB1
Rot. of F relative to B, axis 3.: -(U(2)/CMB1 -(L1*U(2)*C(4)/CMB1
        -U(1)*S(4))/EPS)
Rot. of LFW relative to F, axis 2.: (TRK2*(U(2)/CMB1 + -(U(2)/CMB1
        -(L1*U(2)*C(4)/CMB1 -U(1)*S(4))/EPS)) + U(1)*C(4) + ((L1
        -CMB1)*U(2)/CMB1 + U(2))*S(4))/R
Rot. of RFW relative to F, axis 2.: (-TRK2*(U(2)/CMB1 + -(U(2)/CMB1
        -(L1*U(2)*C(4)/CMB1 -U(1)*S(4))/EPS)) + U(1)*C(4) + ((L1
        -CMB1)*U(2)/CMB1 + U(2))*S(4))/R
Rot. of LRW relative to B, axis 2.: (TRK2*U(2)/CMB1 + U(1))/R
Rot. of RRW relative to B, axis 2.: -(TRK2*U(2)/CMB1 -U(1))/R
```

Note that in the intermediate stage (Table 9.2.4) some of the constraints include the speeds U(3) and U(4), which are subsequently removed. In the later stage (Table 9.2.5) , those constraint equations have been updated so that the only speeds referenced are the two remaining generalized speeds, U(1) and U(2).

The Dynamics Analysis

In this example, the four wheels have mass centers that cannot move relative to the coordinate system of their parent (the parent is B for the rear wheels and F for the front ones). Further, the moment of inertia of each wheel is the same about any axis normal to the spin axis. Thus, the wheels fit the special cases identified in Chapter 8 for “fixed masses” and “rotors.”

Because the four wheels were classified as “fixed masses,” their masses were lumped with the inertia properties of their parents (bodies B and F). Table 9.2.6 lists all of the `point` objects created for the system. Note that for bodies B and F, two points were

created for the center of mass. One (e.g., BCMB) applies to the mass center of the body alone. The other (e.g., BCM) applies to a composite mass and includes the mass properties of all children that are "fixed masses" (e.g., bodies LRW and RRW).

Table 9.2.6. Points in the cart example.

Points	**Description**
Point B0:	Body B: #(0 0 0): coord. origin of the body of cart
Point BCM:	Body B: #(BM*CMB1/(BM + 2.0*MW) 0 0): center of mass of B
Point BCMB:	Body B: #(CMB1 0 0): center of mass of the body of cart
Point F0:	Body F: #(0 0 0): coord. origin of the front axle
Point FCM:	Body F: #(-FM*(EPS -CMF1)/(FM + 2.0*MW) 0 0): center of mass of F
Point FCMB:	Body F: #(-(EPS -CMF1) 0 0): center of mass of the front axle
Point FJ:	Body B: #(L1 0 0): attachment point for the front axle
Point LFW0:	Body LFW: #(0 0 0): coord. origin of the left-front wheel
Point LFWCM:	Body LFW: #(0 0 0): center of mass of LFW
Point LFWJ:	Body F: #(-EPS -TRK2 0): attachment point for the left-front wheel
Point LRW0:	Body LRW: #(0 0 0): coord. origin of the left-rear wheel
Point LRWCM:	Body LRW: #(0 0 0): center of mass of LRW
Point LRWJ:	Body B: #(0 -TRK2 0): attachment point for the left-rear wheel
Point O:	Body N: #(0 0 0): fixed origin
Point RFW0:	Body RFW: #(0 0 0): coord. origin of the right-front wheel
Point RFWCM:	Body RFW: #(0 0 0): center of mass of RFW
Point RFWJ:	Body F: #(-EPS TRK2 0): attachment point for the right-front wheel
Point RRW0:	Body RRW: #(0 0 0): coord. origin of the right-rear wheel
Point RRWCM:	Body RRW: #(0 0 0): center of mass of RRW
Point RRWJ:	Body B: #(0 TRK2 0): attachment point for the right-rear wheel

The `body` object for B is shown in Table 9.2.7, after the constraints have been applied. As mentioned before, the inertia properties apply to a composite body comprised of bodies B, LRW and RRW. Hence, the *mass* slot of B contains the combined mass, the *inertia* slot contains an inertia dyadic that includes the effects of the wheel masses, and the *cm-point* slot contains a point with the center of mass of the composite body. Also, the two translation speeds U(1) and U(2) are defined as scalar values of the velocity of the mass center of the composite body. Hence, the velocity vector in the *abs-v0* slot is derived from the mass center of the composite body.

Table 9.2.7. Slots in body B.

Summary of body:	**B**
parent:	N
recursive-r:	T
recursive-t:	NIL
level:	1
children:	(F LRW RRW)
Name:	Body Of Cart
mass:	(BM + 2.0*MW)
inertia:	(... + (BI33 + BM*((1 -BM/(BM + 2.0*MW))*CMB1)**2 + MW*(2.0*(BM*CMB1)**2/(BM + 2.0*MW)**2 + 2.0*TRK2**2))*([N3].[N3]))
unit-vectors:	#([B1] [B2] [N3])
translation-coordinates:	(Q(1) Q(2))
translation-speeds:	(U(1) U(2))
rotation-coordinates:	(Q(3))
rotation-speeds:	((BM + 2.0*MW)*U(2)/BM/CMB1)
rotation-directions:	([N3])
translation-directions:	([N1] [N2])
joint-pos:	(Point O: Body N: #(0 0 0): fixed origin)
cm-pos:	(Point BCM: Body B: #(BM*CMB1/(BM + 2.0*MW) 0 0): center of mass of B)
abs-w:	(BM + 2.0*MW)*U(2)/BM/CMB1*[N3]
abs-v0:	-((BM*CMB1*(BM + 2.0*MW)*U(2)/BM/CMB1/(BM + 2.0*MW) -U(2))*[B2] -U(1)*[B1])
cos matrix:	#(C(3) S(3) 0)
:	#(-S(3) C(3) 0)
:	#(0 0 1.0)

After the constraints are applied, the entire system has only two independent speeds: U(1) and U(2). The yaw rate, originally designated U(3), was removed. The replacement expression appears everywhere the symbol U(3) originally appeared, such as in the listing of Table 9.2.7 for the *rotation-speeds* slot and the *abs-w* slot.

Table 9.2.8. Slots in `body` LRW

Summary of body:	**LRW**
parent:	B
recursive-r:	ROTOR
recursive-t:	FIXED
level:	2
children:	NIL
Name:	Left-Rear Wheel
mass:	MW
inertia:	(IT*([B1].[B1]) + IA*([B2].[B2]) + IT*([N3].[N3]))
unit-vectors:	#([LRW1] [B2] [LRW3])
rotation-coordinates:	(Q(7))
rotation-speeds:	(((BM + 2.0*MW)*TRK2*U(2)/BM/CMB1 + U(1))/R)
rotation-directions:	([B2])
joint-pos:	(Point LRWJ: Body B: #(0 -TRK2 0): attachment point for the left-rear wheel)
cm-pos:	(Point LRWCM: Body LRW: #(0 0 0): center of mass of LRW)
abs-w:	((BM + 2.0*MW)*U(2)/BM/CMB1*[N3] + ((BM + 2.0*MW)*TRK2*U(2)/BM/CMB1 + U(1))/R*[B2])
abs-v0:	((TRK2*(BM + 2.0*MW)*U(2)/BM/CMB1 + U(1))*[B1] -(BM*CMB1*(BM + 2.0*MW)*U(2)/BM/CMB1/(BM + 2.0*MW) -U(2))*[B2])
cos matrix:	#(COS(Q(7)) 0 -SIN(Q(7)))
:	#(0 1.0 0)
:	#(SIN(Q(7)) 0 COS(Q(7)))

Table 9.2.8 shows the slots in the `body` object created by the `add-body` macro for the left-rear wheel, LRW. Here also, the speed introduced for the rotational degree of freedom has been removed, and is replaced with an expression involving the two independent speeds. Because the wheel is categorized as a `rotor`, the inertia dyadic is written using unit-vectors of the parent.

Two worksheet objects are shown to illustrate how the partial velocities and acceleration remainders are defined for this system. Table 9.2.9 shows the worksheet for body B and Table 9.2.10 shows the worksheet for body LRW.

Table 9.2.9. Worksheet for body B of cart.

Worksheet for body:	B
recursive-r:	T
recursive-t:	NIL
w:	QP(3)*[N3]
wis-a array:	(0, 0, 0, 0, 0, 0, 0, 0)
wis-ab array:	(0, 0, [N3], 0, 0, 0, 0, 0)
wis array:	(0, 0, [N3], 0, 0, 0, 0, 0)
nhwis array:	(0, (BM + 2.0*MW)/BM/CMB1*[N3])
alpha-rem:	0
alpha-ab:	0
nh-alpha-rem:	0
*ra*b0:*	0
*v*is array:*	([B1], [B2], 0, 0, 0, 0, 0, 0)
*v*is bodies:*	(B, B, B, B, B, B, B, B)
*nhv*is array:*	([B1], [B2])
*nhv*is bodies:*	(B, B)
acc-rem:	-(-Z(12)*[B2] + Z(13)*[B1])
nh-acc-rem:	-(-Z(12)*[B2] + Z(13)*[B1])
acc-dyadic:	-(QP(3)**2*([B2].[B2]) + QP(3)**2*([B1].[B1]))

For all bodies, there are eight holonomic partial angular and central velocities, and two nonholonomic counterparts. For this system, it so happens that the two independent speeds were originally numbered U(1) and U(2), so they never changed indices as the constraints were added. For body B, the holonomic and nonholonomic partial velocities are identical. However, holonomic partial angular velocities differ from the nonholonomic ones, because the rotational speed of B (originally U(3)) was removed by a constraint.

Body LRW is identified as being recursive with respect to both rotation and translation. Further, it is identified as a “fixed mass” for the translational part of the analysis, and as a “rotor” for the rotational part. Note that all unit-vectors that appear in the various terms

shown in Table 9.2.10 are based in either bodies N or B. That is, none of the terms are defined in the basis of LRW. The equations of motions are kept simpler than they might be otherwise, by not transforming any vectors into the coordinate system of LRW. Also, all of the quantities used in the translational analysis are assigned to zero, since the mass of RRW was accounted for in the analysis of B.

Table 9.2.10. Worksheet for body RRW of cart.

Worksheet for body:	**RRW**
recursive-r:	ROTOR
recursive-t:	FIXED
w-a:	QP(3)*[N3]
w-ab:	(U(1) -(BM + 2.0*MW)*TRK2*U(2)/BM/CMB1)/R*[B2]
w:	(QP(3)*[N3] + QP(8)*[B2])
wis-a array:	(0, 0, [N3], 0, 0, 0, 0, 0)
wis-ab array:	(0, 0, 0, 0, 0, 0, 0, [B2])
wis array:	(0, 0, [N3], 0, 0, 0, 0, [B2])
nhwis array:	(1.0/R*[B2], (-(BM + 2.0*MW)*TRK2/R/BM/CMB1*[B2] + (BM + 2.0*MW)/BM/CMB1*[N3]))
alpha-rem:	-PC(8)*(U(1) -Z(8))*QP(3)*[B1]
alpha-ab:	-(U(1) -(BM + 2.0*MW)*TRK2*U(2)/BM/CMB1)*QP(3)/R*[B1]
nh-alpha-rem:	-PC(8)*(U(1) -Z(8))*QP(3)*[B1]
*v*is array:*	(0, 0, 0, 0, 0, 0, 0, 0)
*v*is bodies:*	(B, B, B, B, B, B, B, B)
*nhv*is array:*	(0, 0)
*nhv*is bodies:*	(B, B)
acc-rem:	0
nh-acc-rem:	0

Equations of Motion

The equations of motion for this system are shown in Figures 9.2.8, 9.2.9 and 9.2.10. There is an interesting use of intermediate variables in the listings here. Without the nonholonomic constraints, the kinematical equations for generalized coordinates Q(3) through Q(8) would simply be of the form:

```
QP(3) = U(3)
QP(4) = U(4)
QP(5) = U(5)
QP(6) = U(6)
QP(7) = U(7)
QP(8) = U(8)
```

However, the speeds U(3) through U(8) were eliminated as independent variables. Hence, more complicated kinematical equations appear, which are essentially statements of the constraint equations. Later, when a dependent speed might normally appear (e.g., in an acceleration remainder), the derivatives QP(3) through QP(8) are likely to appear. For example, they appeared in expressions for angular velocity and angular acceleration remainder in the listing of the worksheet of body RRW in Table 9.2.9.

```
  PC(1) = 2.0*MW
  PC(2) = (BM + 2.0*MW)
  PC(3) = (BM + 2.0*MW)/BM/CMB1
  PC(4) = (L1 -BM*CMB1/(BM + 2.0*MW))
  PC(5) = (1 + (BM + 2.0*MW)*(L1 -BM*CMB1/(BM + 2.0*MW))/BM/CMB1)
  PC(6) = 1.0/EPS
  PC(7) = TRK2/EPS
  PC(8) = 1.0/R
  PC(9) = (BM + 2.0*MW)*TRK2/BM/CMB1
  PC(10) = (FM + 2.0*MW)
  PC(11) = (2.0*MW*EPS + FM*(EPS -CMF1))/(FM + 2.0*MW)
  PC(12) = (2.0*MW*EPS + FM*(EPS -CMF1))/(FM + 2.0*MW)/EPS
  PC(13) = (1 -(2.0*MW*EPS + FM*(EPS -CMF1))/(FM + 2.0*MW)/EPS)
  PC(14) = (1 + (BM + 2.0*MW)*(L1 -BM*CMB1/(BM +
 &          2.0*MW))/BM/CMB1)/EPS
  PC(15) = (FI33 + FM*(-EPS + CMF1 + (2.0*MW*EPS + FM*(EPS
 &          -CMF1))/(FM + 2.0*MW))**2 + MW*(2.0*(EPS -(2.0*MW*EPS +
 &          FM*(EPS -CMF1))/(FM + 2.0*MW))**2 + 2.0*TRK2**2))/EPS
  PC(16) = (1 + (BM + 2.0*MW)*(L1 -BM*CMB1/(BM + 2.0*MW))/BM/CMB1)
 &         *(FI33 + FM*(-EPS + CMF1 + (2.0*MW*EPS + FM*(EPS
 &          -CMF1))/(FM + 2.0*MW))**2 + MW*(2.0*(EPS -(2.0*MW*EPS +
 &          FM*(EPS -CMF1))/(FM + 2.0*MW))**2 + 2.0*TRK2**2))/EPS
  PC(17) = IA/R
  PC(18) = IT/EPS
  PC(19) = IA*(1 + (BM + 2.0*MW)*(L1 -BM*CMB1/(BM +
 &          2.0*MW))/BM/CMB1)/R
  PC(20) = IT*(1 + (BM + 2.0*MW)*(L1 -BM*CMB1/(BM +
 &          2.0*MW))/BM/CMB1)/EPS
  PC(21) = 2.0*IA/R
  PC(22) = (BM + 2.0*MW + (2.0*IT + BI33 + BM*((1 -BM/(BM +
 &          2.0*MW))*CMB1)**2 + MW*(2.0*(BM*CMB1)**2/(BM +
 &          2.0*MW)**2 + 2.0*TRK2**2) + 2.0*IA*TRK2**2/R**2)*(BM +
 &          2.0*MW)**2/(BM*CMB1)**2)
  PC(23) = (BM + PC(1))
  PC(24) = PC(23)/BM/CMB1
  PC(25) = BM*CMB1/PC(23)
  PC(26) = (L1 -PC(25))
  PC(27) = PC(23)*PC(26)/BM/CMB1
  PC(28) = (1 + PC(27))
  PC(29) = 2.0*PC(18)
  PC(30) = (PC(15) + PC(29))
  PC(31) = PC(5)*PC(17)
  PC(32) = PC(3)*PC(11)
  PC(33) = PC(3)*PC(4)
  PC(34) = PC(11)*PC(14)
  PC(35) = (1 + PC(33) -PC(34))
  PC(36) = (1 + PC(33))
  PC(37) = 2.0*PC(20)
  PC(38) = (PC(16) + PC(37))
  PC(39) = PC(5)*PC(8)
  PC(40) = PC(5)*PC(6)
  PC(41) = PC(6)*PC(30)
  PC(42) = PC(8)*PC(21)
  PC(43) = PC(8)*PC(17)
  PC(44) = (PC(2) + PC(42))
```

Figure 9.2.8. Constants that are precomputed for the cart.

```
C Equations of Motion
C ===================
C  Each derivative evaluation requires 94 multiply/divides, 48
C  add/subtracts, and 4 function/subroutine calls.
C
   C(3) = COS(Q(3))
   C(4) = COS(Q(4))
   S(3) = SIN(Q(3))
   S(4) = SIN(Q(4))
C
C  Kinematical equations
C
   QP(1) = U(1)*C(3)
   QP(2) = U(1)*S(3)
   QP(3) = PC(3)*U(2)
   Z(1) = U(2)*C(4)
   Z(2) = U(1)*S(4)
   Z(3) = (PC(5)*Z(1) -Z(2))
   Z(4) = PC(6)*Z(3)
   QP(4) = (Z(4) -QP(3))
   Z(5) = U(1)*C(4)
   Z(6) = PC(7)*Z(3)
   Z(7) = PC(5)*U(2)*S(4)
   QP(5) = PC(8)*(Z(5) + Z(6) + Z(7))
   QP(6) = PC(8)*(Z(5) -Z(6) + Z(7))
   Z(8) = PC(9)*U(2)
   QP(7) = PC(8)*(U(1) + Z(8))
   QP(8) = PC(8)*(U(1) -Z(8))
```

Figure 9.2.9. Kinematical equations for the cart.

The equations shown here are significantly more complex than those developed by Ge and Cheng [28]. Neglecting the variable mass, their equations of motion require only 34 multiplications. The reason for this is that the “rocket car” had a front axle with no offset from the steer point. That is, the mass center of the axle coincided with the steer point. Also, the steer point was located along the spin axis of the front wheels. The input to AUTOSIM was modified to match the description in [28], and interesting results were obtained. First, it was found to be necessary to change the relationship between the steered front axle and the car body, such that the axle was the parent of the body. This ensured that the constraint equations were non-singular.

The independent speeds defined by AUTOSIM were the forward velocity and yaw rate of the steered axle. (Ge and Cheng used the forward velocity of the steered axle, and the steer rate of the axle.) The AUTOSIM formulation had the same efficiency (34 multiplications), but one of the dynamical equations is trivial: UP(2) = 0. (U(2) is the symbol for the yaw rate of the front axle.) The equations of motion obtained by Ge and

Cheng can be transformed to show the same thing, but it is not obvious from a casual inspection of the equations.

```
C
C  Dynamical equations
C
   Z(9) = PC(13)*S(4)
   Z(10) = PC(35)*C(4)
   Z(11) = PC(36)*S(4)
   Z(12) = U(1)*QP(3)
   Z(13) = U(2)*QP(3)
   Z(14) = (Z(13) + PC(4)*QP(3)**2)
   Z(15) = (PC(11)*Z(4)**2 -Z(14)*C(4) + Z(12)*S(4))
   Z(16) = (Z(5) + Z(7))*QP(4)
   Z(17) = PC(10)*C(4)
   Z(18) = PC(10)*Z(9)
   Z(19) = PC(10)*Z(11)
   Z(20) = PC(10)*Z(10)
   Z(21) = PC(7)*S(4)
   Z(22) = (-Z(21) + C(4))
   Z(23) = PC(7)*C(4)
   Z(24) = (Z(23) + S(4))
   Z(25) = -(-PC(31)*U(2)*Z(22) + PC(17)*U(1)*Z(24))*QP(4)
   Z(26) = PC(19)*Z(24)
   Z(27) = (Z(21) + C(4))
   Z(28) = (-Z(23) + S(4))
   Z(29) = -(-PC(31)*U(2)*Z(27) + PC(17)*U(1)*Z(28))*QP(4)
   Z(30) = PC(19)*Z(28)
   Z(31) = PC(30)*Z(16)
   Z(32) = (PC(12)*Z(16) + Z(12)*C(4) + Z(14)*S(4))
   Z(33) = (F + PC(2)*Z(13) -Z(15)*Z(17) -PC(8)*(Z(22)*Z(25) +
  &         Z(27)*Z(29)) + Z(18)*Z(32) -PC(6)*Z(31)*S(4))
   Z(34) = (PC(44) + Z(9)*Z(18) + PC(43)*(Z(22)**2 + Z(27)**2) +
  &         Z(17)*C(4) + PC(41)*S(4)**2)
   Z(35) = PC(38)*C(4)
   Z(36) = (Z(9)*Z(20) -PC(8)*(Z(22)*Z(26) + Z(27)*Z(30))
  &         -Z(19)*C(4) + PC(6)*Z(35)*S(4))
   Z(37) = Z(36)/Z(34)
   Z(38) = (-PC(2)*Z(12) -Z(15)*Z(19) -PC(39)*(Z(24)*Z(25) +
  &         Z(28)*Z(29)) -Z(20)*Z(32) + Z(33)*Z(37) +
  &         PC(40)*Z(31)*C(4))/(PC(22) + Z(11)*Z(19) + Z(10)*Z(20) +
  &         PC(39)*(Z(24)*Z(26) + Z(28)*Z(30)) -Z(36)*Z(37) +
  &         PC(40)*Z(35)*C(4))
   UP(2) = Z(38)
   UP(1) = (Z(33) + Z(36)*Z(38))/Z(34)
```

Figure 9.2.10. Dynamical equations for the cart.

9.3. Four-bar Linkage with Spring

The example described in this section illustrates (1) how closed kinematical loops are handled, (2) the use of alternative coordinate systems in the input description, and (3) use of a "strut" force element.

Model Description

The system is comprised of a four-bar linkage with a strut, shown in Figure 9.3.1. In this figure, the coordinates of key points are shown using a global coordinate system fixed in N. The system has three bodies, A, B, and C. However, all mass is lumped in body B. Bodies A and C are massless links and there is a spring/damper combination that is fixed between two points. (The spring/damper strut is shown as a simple spring.) The system is subject to a uniform gravitational field. The system is planar. The coordinates of points of joint locations, the mass center of B, and the points of attachment of the spring are shown for the nominal configuration.

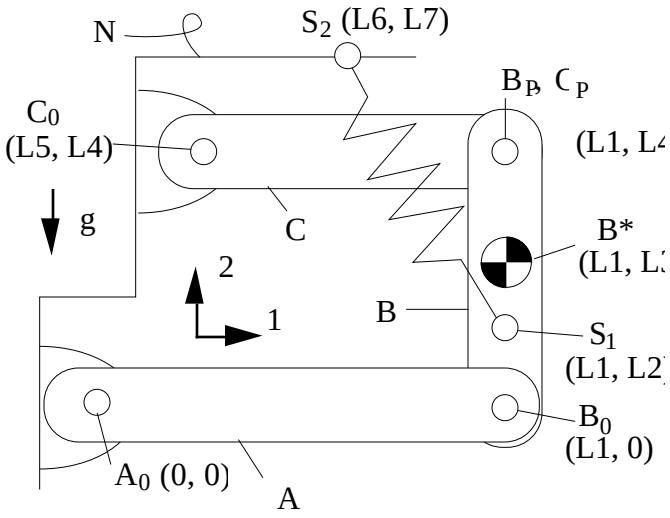

Figure 9.3.1. Four-bar linkage.

The simulation will be used to obtain time histories of the angles of the three bodies, the trajectory of the mass center of B, and the force produced by the strut.

AUTOSIM Description

The complete description of the system is shown in Figure 9.3.2. The first three input macros define the three rigid bodies, A, B, and C. The optional arguments `:mass`, `:inertia-matrix`, `:body-rotation-axes`, `:joint-coordinates`, and `:cm-coordinates` were used in previous examples and should be familiar by now. The next two macros, `add-point`, should also be familiar. An additional optional argument named `:coordinate-system` is used in most of these macros. This

argument is used to specify an alternative coordinate system for coordinates provided to the macro. In this example, it is used for the `add-body` and `add-point` macros to indicate that coordinates are in the coordinate system of N. Thus, the global coordinates shown in Figure 9.3.1 are provided directly as arguments.

```
(add-body a :mass 0
    :inertia-matrix 0
    :body-rotation-axes 3)

(add-body b :parent a
    :body-rotation-axes 3
    :joint-coordinates #(L1 0 0)
    :cm-coordinates #(L1 L3 0)
    :coordinate-system n)

(add-body c :mass 0
    :inertia-matrix 0
    :body-rotation-axes 3
    :joint-coordinates #(L5 L4 0)
    :coordinate-system n)

(add-point bp :name "b-point"
    :body b
    :coordinates #(L1 L4 0)
    :coordinate-system n)

(add-point cp :name "c-point"
    :body c
    :coordinates #(L1 L4 0)
    :coordinate-system n)

(no-movement bp cp [b1])
(no-movement bp cp [c2])

;; add gravity and strut force

(add-gravity
    :direction !"-[n2]")

(add-point s1:body b
    :name "strut pt 1"
    :coordinates #(L1 L2 0)
    :coordinate-system n)
```

```
(add-point s2 :body n
    :name "strut pt 2"
    :coordinates #(L6 L7 0))

(add-strut f  :name "strut"
    :magnitude
        !"-k*(x - x0) - v*d"
    :point1 s1 :point2 s2)

;; describe output variables

(add-out !"-fm(f)" "F" :body b
    :long-name "strut force")

(add-out !"dot([n1], pos(bcm))" "B*
    X" :body b
    :long-name
       "X coordinate of B*")

(add-out !"dot([n2], pos(bcm))"
    "B* Y" :body b
    :long-name
       "Y coordinate of B*")

(add-coordinates-to-output)

(add-out !"q(1) + q(2)" "B-angle"
    :long-name
        "angle of B rel. to N"
    :body b)

(mks)
(set-defaults L1 .5 l2 .1 l3 .2
    l4 .3 l5 .1 L6 .3 L7 .5
    k 10000 d 10
    bm 10 bi33 1
    step .005 stopt 1 iprint 5)
```

Figure 9.3.2. Description of kinematics of four-bar linkage.

The closed kinematical loop is described by declaring that there is no movement between two points: one in body B and one in C. First, the two points are defined with `add-point` macros, and called BP and CP. Then, the `no-movement` macro is used twice to add the constraints. The directions of the constraints ([b1] and [c2]) were chosen with the rotational speeds of B and C in mind, such that the new constraints would not duplicate constraints inherent in other joint kinematics. The component of the velocity of

point BP due to the rotational speed of B is in the direction [b1], and the component of the velocity of point CP due to the rotational speed of C is in the direction of [c2]. On the other hand, the component of the velocity of BP in direction [b2] is the same as the component of the velocity of B_0 in direction [b2], and has no relationship whatsoever to the rotational speed of B. (Depending on the orientation of the bodies, it may or may not be related to other generalized speeds.) No matter how the bodies in the system are oriented, coefficients obtained for speed constraints defined for the directions [b1] and [c2] are nonsingular.

The macros `add-gravity` and `add-strut` apply forces due to gravity and the strut, respectively. The `add-strut` macro is used for a force whose direction changes as needed so that the force passes through two known points. The magnitude of the force is provided as an expression with the keyword `:magnitude`. Three dummy variables can appear in the expression, and all three are used in the example: (1) the symbol `x` is replaced by an expression for the distance between the two points, (2) the symbol `x0` is replaced by a constant expression for the nominal distance between the two points, and (3) the symbol `v` is replaced by an expression for the speed between the two points, along line connecting the points. Because the spring force is proportional to the distance `(x - x0)`, the free spring length is the nominal length. That is, when the system is oriented as drawn in Figure 9.3.1, the spring produces zero force.

Next, output variables are defined. The simulation code will include the strut force, the coordinates of the mass center of B, the generalized coordinates of the system, and the absolute rotation angle of B.

Results

Time history plots are shown in Figures 9.3.3 through 9.3.6 for two sets of initial conditions: (1) the nominal configuration, and (2) the lower link rotated down by an angle of 0.5 radian. In the first case, the orientation is initially trivial to compute, because it exactly matches the drawing of figure 9.3.1. It is not in equilibrium, however, because the spring is not producing a tensile force to balance the weight of B. In the second case, the initial values of the angles of bodies B and C must be computed to maintain the constraints.

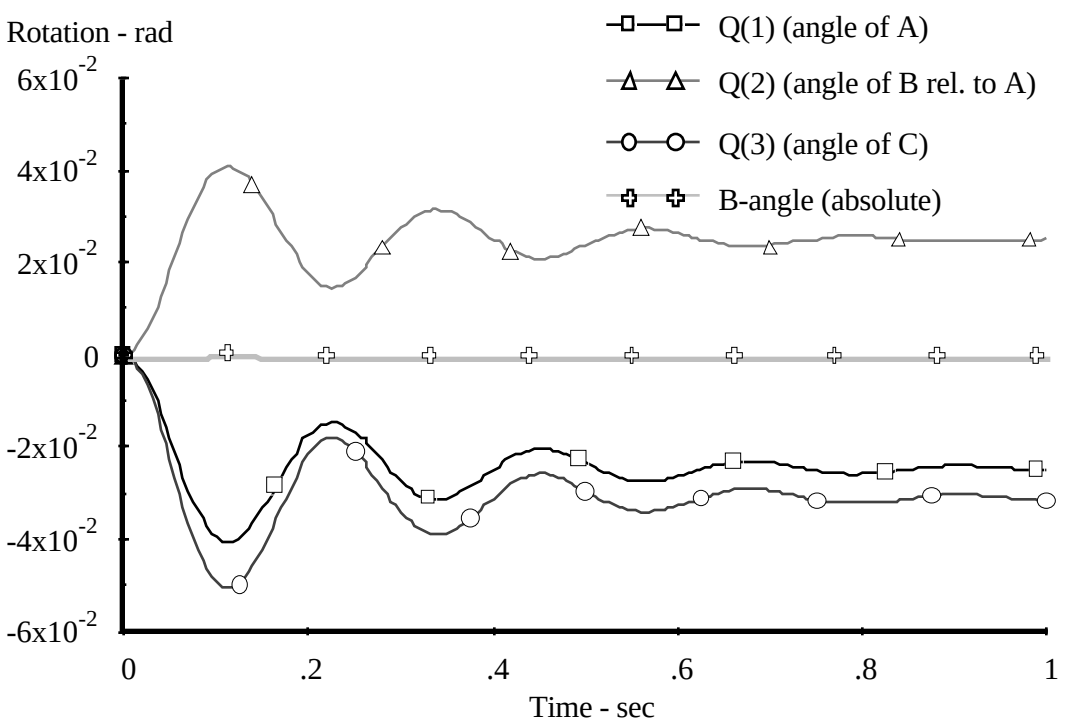

Figure 9.3.3. Time histories of rotation angles for nominal initial conditions.

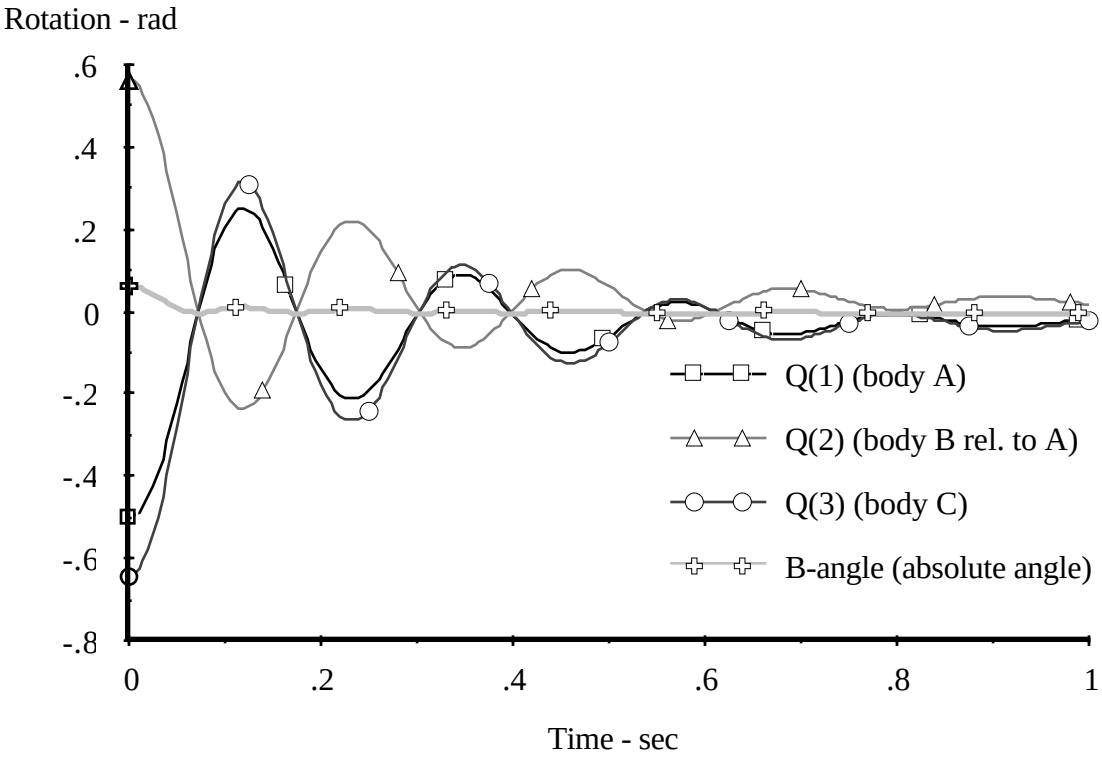

Figure 9.3.4. Time histories of rotation angles for displaced initial conditions.

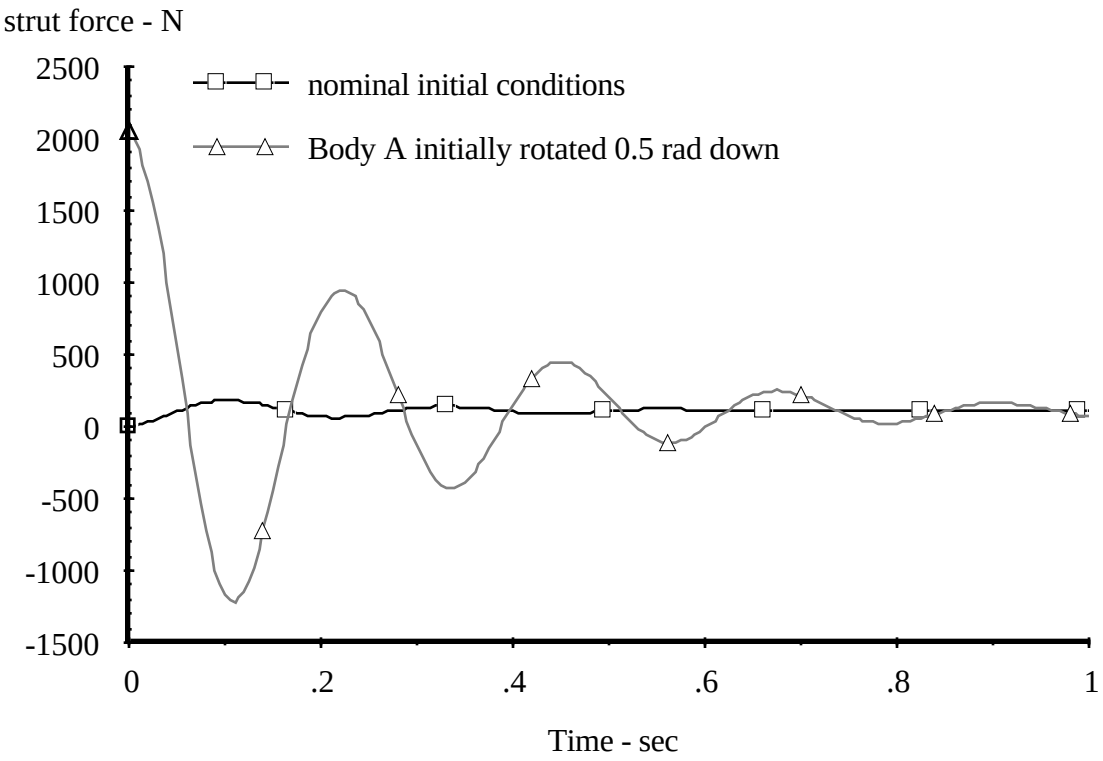

Figure 9.3.5. Time histories of strut force.

The model was validated by running a similar model through the DADS program. To simplify the representation in DADS, the model was modified to include nonzero mass and inertia values for bodies A and C. When a corresponding simulation code was generated with AUTOSIM (i.e., with bodies A and C having nonzero masses and moments of inertia) the results from DADS and the simulation code generated by AUTOSIM agreed.

Analysis Details

This system involves several analysis methods that were not used in previous examples. First, the coordinates of points in the system were all provided in the global coordinate system. When the AUTOSIM inputs were processed, the coordinates of each point were converted to the coordinate system of the body containing the point. This can be seen by viewing all of the points in the systems, shown in Table 9.3.1.

The state variables and constraint equations are shown in Table 9.3.2. The constraints applied by the `no-movement` macros reduce the number of degrees of freedom to one: the rotational speed of body A. Also, two of the coordinates are classified as “computed coordinates,” rather than as “independent coordinates.”

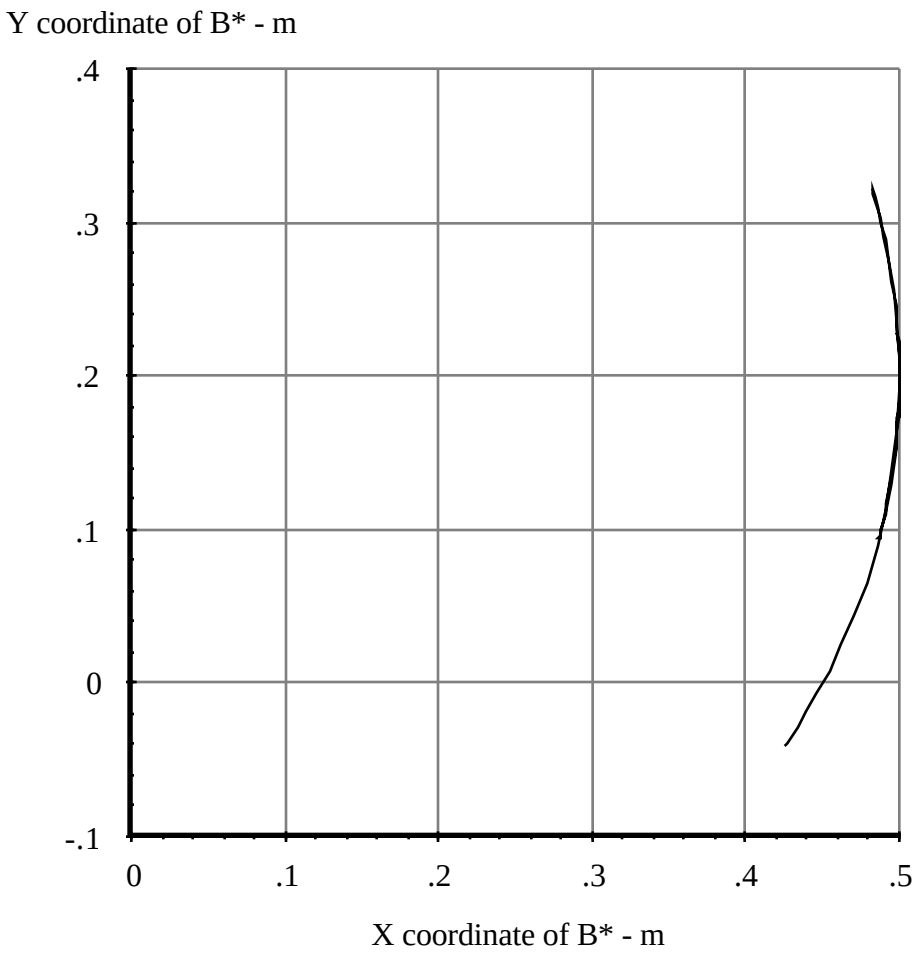

Figure 9.3.6. Trajectory of mass center of body B.

Table 9.3.1. Points defined for four-bar linkage.

Point:	**Description**
Point O:	Body N: #(0 0 0): fixed origin
Point A0:	Body A: #(0 0 0): coord. origin of A
Point BJ:	Body A: #(L1 0 0): attachment point for B
Point B0:	Body B: #(0 0 0): coord. origin of B
Point BCMB:	Body B: #(0 L3 0): center of mass of B
Point BCM:	Body B: #(0 L3 0): center of mass of B
Point CJ:	Body N: #(L5 L4 0): attachment point for C
Point C0:	Body C: #(0 0 0): coord. origin of C
Point BP:	Body B: #(0 L4 0): B-point
Point CP:	Body C: #((L1 -L5) 0 0): C-point
Point S1:	Body B: #(0 L2 0): strut pt 1
Point S2:	Body N: #(L6 L7 0): strut pt 2

Table 9.3.2. State variables and speed constraints for four-bar linkage.

Generalized Coordinates:

Q(1): Rotation of A relative to the inertial reference about axis #3. (rad)

Q(2): Rotation of B relative to A about axis #3. (rad)

Q(3): Rotation of C relative to the inertial reference about axis#3. (rad)

Independent Speeds:

U(1): Abs. rot. of A, axis 3. (rad/s)

Nonholonomic Constraints:

Rot. of B relative to A, axis 3.: -U(1)*(1 -L1*(S(2) -(L1 -L5)*(C(3)*(C(1)*C(2)**2 - C(2)*S(1)*S(2)) + C(2)*(C(2)*S(1) + C(1)*S(2))*S(3))*(S(2)*(C(1)*C(3) + S(1)*S(3)) + C(2)*(C(3)*S(1) -C(1)*S(3)))/(L1 -L5 -(L1 -L5)*(S(2)*(C(1)*C(3) + S(1)*S(3)) + C(2)*(C(3)*S(1) -C(1)*S(3)))**2))/L4)

Abs. rot. of C, axis 3.: L1*U(1)*(C(3)*(C(1)*C(2)**2 -C(2)*S(1)*S(2)) + C(2)*(C(2)*S(1) + C(1)*S(2))*S(3))/(L1 -L5 -(L1 -L5)*(S(2)*(C(1)*C(3) + S(1)*S(3)) + C(2)*(C(3)*S(1) -C(1)*S(3)))**2)

The simulation code, listed in Appendix C, includes several subroutines that are not written for the other examples in this chapter. After the input data are read, the main program calls the subroutine MNEWT to solve the multiple equation set for the initial conditions using a Newton-Raphson iteration. The subroutine MNEWT in turn solves multiple linear equations with subroutines LUDCMP and LUBKSB. The algorithms used are fairly standard (the subroutines were adopted from listings provided in []). The subroutine INITR, generated by AUTOSIM, computes the constraint errors for the system and the Jacobian coefficients to define the linear equations solved by LUDCMP and LUBKSB.

A section of the code in the subroutine INITNR is shown in Figure 9.3.7. Each of the two constraint equations has an error function (BETA) that is zero when the constraint is satisfied. The Jacobian (ALPHA) provides the partial derivative of each error function with respect to a computed variable. In this subroutine, the independent variable is called Q(1), just as it is elsewhere in the simulation code. However, the two computed variables, normally called Q(2) and Q(3), are called X(1) and X(2) in this subroutine. (The change in names is made to accommodate the subroutine MNEWT that computes candidate values of the computed coordinates.)

```
      BETA(1) = (-L1*COS(X(1)) + L4*(COS(X(1))*SIN(Q(1)) +
     &      COS(Q(1))*SIN(X(1))) + L5*(COS(Q(1))*COS(X(1))
     &      -SIN(Q(1))*SIN(X(1))) + (L1 -L5)*(-SIN(X(1))
     &     *(COS(X(2))*SIN(Q(1)) -COS(Q(1))*SIN(X(2))) + COS(X(1))
     &     *(COS(Q(1))*COS(X(2)) + SIN(Q(1))*SIN(X(2)))))
      ALPHA(1,1) = -(-L5*(COS(X(1))*SIN(Q(1)) + COS(Q(1))*SIN(X(1))) +
     &      L4*(COS(Q(1))*COS(X(1)) -SIN(Q(1))*SIN(X(1))) +
     &      L1*SIN(X(1)) -(L1 -L5)*(SIN(X(1))*(COS(Q(1))*COS(X(2)) +
     &      SIN(Q(1))*SIN(X(2))) + COS(X(1))*(COS(X(2))*SIN(Q(1))
     &      -COS(Q(1))*SIN(X(2))))
      ALPHA(1,2) = -(L1 -L5)*(COS(X(1))*(COS(X(2))*SIN(Q(1))
     &      -COS(Q(1))*SIN(X(2))) + SIN(X(1))*(COS(Q(1))*COS(X(2)) +
     &      SIN(Q(1))*SIN(X(2))))
      BETA(2) = -(L5*SIN(X(2)) -L4*(COS(X(2)) + SIN(X(1))
     &     *(COS(X(2))*SIN(Q(1)) -COS(Q(1))*SIN(X(2))) -COS(X(1))
     &     *(COS(Q(1))*COS(X(2)) + SIN(Q(1))*SIN(X(2)))) + L1
     &     *(COS(X(2))*SIN(Q(1)) -COS(Q(1))*SIN(X(2))))
      ALPHA(2,1) = -L4*(SIN(X(1))*(COS(Q(1))*COS(X(2)) +
     &      SIN(Q(1))*SIN(X(2))) + COS(X(1))*(COS(X(2))*SIN(Q(1))
     &      -COS(Q(1))*SIN(X(2))))
      ALPHA(2,2) = (L5*COS(X(2)) + L4*(SIN(X(2)) + SIN(X(1))
     &     *(COS(Q(1))*COS(X(2)) + SIN(Q(1))*SIN(X(2))) + COS(X(1))
     &     *(COS(X(2))*SIN(Q(1)) -COS(Q(1))*SIN(X(2)))) -L1
     &     *(COS(Q(1))*COS(X(2)) + SIN(Q(1))*SIN(X(2))))
```

Figure 9.3.7. Jacobian matrix (ALPHA) and error function (BETA) used to compute initial conditions for four-bar linkage.

The computer code shown in Figure 9.3.7 is obviously not optimized in the same fashion as the Fortran code appearing elsewhere in the simulation code. The Newton-Raphson iteration is performed only once in each simulation run, as part of the initialization. Thus, the numerical efficiency of this code has a negligible effect on the efficiency of the simulation code as a whole.

The computed initial conditions are written in the echo file, made for each simulation run. The simulation run that generated the data plotted in in Figures 9.3.3 through 9.3.6 produced the echo file shown in Table 9.3.3.

After the initialization, constraints on the coordinates are handled largely by the inclusion of constraints on the corresponding speeds in the formulation of the dynamical equation. However, to avoid violating a constraint due to accumulated integration error, a correction is made each time the subroutine DIFEQN is called. The code that “corrects” the computed coordinates is listed in Figure 9.3.8. The purpose of the code is to correct the values of the variables Q(2) and Q(3), which are both shown in boldface.

Table 9.3.3. Echo file for 4-bar linkage with displaced initial conditions.

```
PARSFILE
Echo file created by:
4-bar linkage simulation program.
Version created December 11, 1989 by AUTOSIM

TITLE  Default parameter values

* Input File: echo ic
* Run was made 13:02 on Dec 11, 1989

* PARAMETER VALUES

BI33     1.00000       moment of inertia of B (kg-m2)
BM       10.0000       mass of B (kg)
D        100.000       coefficient in term in strut (N-sec/rad/m)
IPRINT   1.00000       no. of time steps between printing (counts)
K        10000.0       stiffness coefficient in term in strut (N/m)
L1       .500000       coord. of attachment point for B in dir 1 (m)
L2       .100000       coordinate of strut pt 1 in dir 2 (m)
L3       .200000       coordinate of center of mass of B in dir 2 (m)
L4       .300000       coordinate of b-point in dir 2 (m)
L5       .100000       coord. of attachment point for C in dir 1 (m)
L6       .300000       coordinate of strut pt 2 in dir 1 (m)
L7       .500000       coordinate of strut pt 2 in dir 2 (m)
STEP    0.500000E-02   simulation time step (sec)
STOPT    1.00000       simulation stop time (sec)

* INITIAL CONDITIONS

Q(1)    -.500000       Rotation of A relative to the inertial reference about
    axis #3. (rad)
Q(2)     .563473       Rotation of B relative to A about axis #3. (rad)
Q(3)    -.644491       Rotation of C relative to the inertial reference about
    axis #3. (rad)
U(1)     .000000       Abs. rot. of A, axis 3. (rad/s)

END
```

The corrections made by the code in Figure 9.3.8 are recursive. That is, the new values of Q(2) and Q(3) are computed by adjusting the values that were provided by numerically integrating their derivatives. If there were no error in the numerical integration, then the correction terms Z(21) and Z(41) would be zero. With the second-order numerical integration method used in Appendix C, the error is typically on the order of 10^{-6} radian. By using the correction technique illustrated in the above listing, acceptable accuracy is obtained with simple integration algorithms and single precision variables.

```
Z(1) = L5*S(3)
Z(2) = C(3)*S(1)
Z(3) = C(1)*S(3)
Z(4) = (Z(2) -Z(3))
Z(5) = Z(4)*S(2)
Z(6) = C(1)*C(3)
Z(7) = S(1)*S(3)
Z(8) = (Z(6) + Z(7))
Z(9) = Z(8)*C(2)
Z(10) = (Z(5) -Z(9) + C(3))
Z(11) = L4*Z(10)
Z(12) = L1*Z(4)
Z(13) = (Z(1) -Z(11) + Z(12))
Z(14) = L5*C(3)
Z(15) = Z(8)*S(2)
Z(16) = Z(4)*C(2)
Z(17) = (Z(15) + Z(16) + S(3))
Z(18) = L4*Z(17)
Z(19) = L1*Z(8)
Z(20) = (Z(14) + Z(18) -Z(19))
Z(21) = Z(13)/Z(20)
Z(22) = (-Q(3) + Z(21))
Q(3) = -Z(22)
```

```
Z(23) = L1*C(2)
Z(24) = C(1)*C(2)
Z(25) = S(1)*S(2)
Z(26) = (Z(24) -Z(25))
Z(27) = L5*Z(26)
Z(28) = C(2)*S(1)
Z(29) = C(1)*S(2)
Z(30) = (Z(28) + Z(29))
Z(31) = L4*Z(30)
Z(32) = (-Z(5) + Z(9))
Z(33) = PC(1)*Z(32)
Z(34) = (Z(23) -Z(27) -Z(31) -Z(33))
Z(35) = L4*Z(26)
Z(36) = L5*Z(30)
Z(37) = L1*S(2)
Z(38) = (Z(15) + Z(16))
Z(39) = PC(1)*Z(38)
Z(40) = (Z(35) -Z(36) + Z(37) -Z(39))
Z(41) = Z(34)/Z(40)
Q(2) = (Q(2) + Z(41))
```

Figure 9.3.8. Correction of integration error in computed coordinates Q(2) and Q(3) for four-bar linkage.

Forces

F: strut: Expression = FORCEM(1): Direction = -(L7/SQRT(L6*(L6-L1*C(1) + L2*(C(2)*S(1) + C(1)*S(2))) + L2*(L2 + L6*(C(2)*S(1) +C(1)*S(2)) -L1*S(2) - L7*(C(1)*C(2) -S(1)*S(2))) + L1*(L1 -L6*C(1)-L7*S(1) -L2*S(2)) + L7*(L7 - L1*S(1) -L2*(C(1)*C(2)-S(1)*S(2))))*[N2] + L6/SQRT(L6*(L6 -L1*C(1) + L2*(C(2)*S(1) +C(1)*S(2))) + L2*(L2 + L6*(C(2)*S(1) + C(1)*S(2)) -L1*S(2) - L7*(C(1)*C(2) -S(1)*S(2))) + L1*(L1 -L6*C(1) -L7*S(1) -L2*S(2)) +L7*(L7 - L1*S(1) -L2*(C(1)*C(2) -S(1)*S(2))))*[N1] -L2/SQRT(L6*(L6-L1*C(1) + L2*(C(2)*S(1) + C(1)*S(2))) + L2*(L2 + L6*(C(2)*S(1) +C(1)*S(2)) -L1*S(2) - L7*(C(1)*C(2) -S(1)*S(2))) + L1*(L1 -L6*C(1)-L7*S(1) -L2*S(2)) + L7*(L7 - L1*S(1) -L2*(C(1)*C(2)-S(1)*S(2))))*[B2] -L1/SQRT(L6*(L6 -L1*C(1) + L2*(C(2)*S(1) +C(1)*S(2))) + L2*(L2 + L6*(C(2)*S(1) + C(1)*S(2)) -L1*S(2) - L7*(C(1)*C(2) -S(1)*S(2))) + L1*(L1 -L6*C(1) -L7*S(1) -L2*S(2)) +L7*(L7 - L1*S(1) -L2*(C(1)*C(2) -S(1)*S(2))))*[A1]).

Acts on B from the inertial reference through strut pt 1 and strut pt 2

Figure 9.3.9 `Force` object created to represent strut.

One final note about this example is that it includes a force-producing component that involves complicated algebraic expressions. The `add-strut` macro shown in Figure 9.3.2 creates the `force` object listed in Figure 9.3.9.

9.4. "Spacecraft #1"

A spacecraft model with 10 degrees of freedom is described in the SD/FAST Users Manual [5]. That model was analyzed with AUTOSIM to determine how the simulation codes produced using the methods described in this dissertation compare with those produced by SD/FAST in terms of efficiency and agreement of the predicted variables. Also, this example illustrates how external subroutines are incorporated into the simulation codes generated by AUTOSIM.

Model Description

The spacecraft is composed of three rigid bodies and one flexible body modeled as a rigid body with a u-joint. The rigid bodies are the main body of the craft, called the bus and designated body B, a camera (body D), and a supporting shaft called a clock (body C). The flexible member is called the boom (massless body E and body F). Figure 9.4.1 shows a sketch of the rigid bodies, reference points, and dimensional parameters.

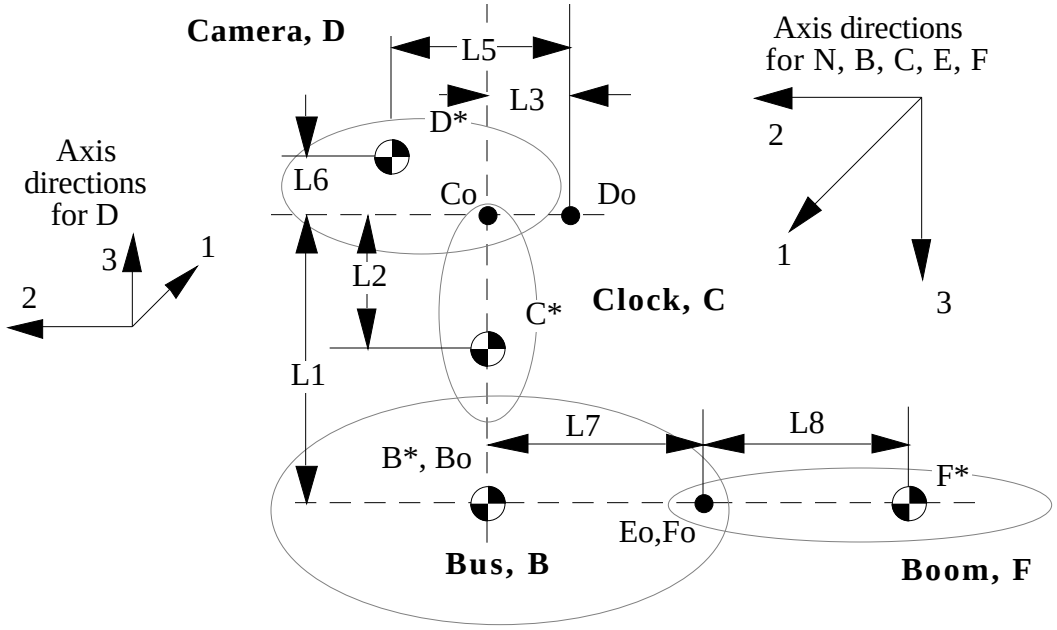

Figure 9.4.1. Sketch of bodies in Spacecraft #1.

The flexibility of the boom is modeled with a two-degree of freedom hinge, with torsional stiffness KB and torsional damping rate BB in the directions 1 and 3.

Movements of the clock and camera are controlled. The controller is modeled as applying a torque through a massless element with torsional stiffness KCLOCK and torsional damping rate BCLOCK to the clock. The torque applied to the camera is also through a massless element with the same stiffness and damping properties as used for the clock.

The object of the simulation is to simulate a "slew maneuver" in which the clock and camera are moved from initial values of 4 and -0.5 radians, respectively, to final values of 3.75 and -0.4 radians, over a ten-second interval. A Fortran subroutine, based on the example in [5], is listed in the left-hand column of Figure 9.4.2. The simulation code should include this subroutine to obtain the control signals.

```
SUBROUTINE CMD(T, CLKCMD, CAMCMD)
IF (T .LT. 1.) THEN
  CLKCMD = 4.
  CAMCMD = -.5
ELSE IF (T .LT. 11.) THEN
  CLKCMD = 4. -.025*(T-1.)
  CAMCMD = -.5 + .01*(T-1.)
ELSE
  CLKCMD = 3.75
  CAMCMD = -.4
END IF
RETURN
```

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

Figure 9.4.2. Subroutines for computing control signals and couples from thrusters.

The orientation of the spacecraft body is controlled by three pairs of thrusters that fire bursts of propellent when the angle of the craft drifts beyond a "dead zone" tolerance. Each pair of thrusters is balanced to apply a pure couple to B about the directions 1, 2, and 3.

The control laws of the thrusters used in Ref. [5] are shown in the Fortran listing in the right-hand column of Figure 9.4.2. Each of three thruster pairs fires when an error signal exceeds a threshold of 0.0025, and remains on for at least a time duration of 0.02 seconds.

The algorithm shown in the figure assumes that the function is always called with increasing values of time.

AUTOSIM Description

The inputs to AUTOSIM that define the rigid bodies of the system are shown in the listing of Figure 9.4.4. The two-degree-of-freedom joint between the bus and the boom is entered as two bodies, each with a single rotational degree of freedom. (The first, E, is massless.) The rotation axis of the camera (axis #1) is reversed from the direction of axis #1 in the clock. Hence, the coordinate system of D in the nominal orientation is reversed relative to the coordinate systems of the other bodies.

Two simulation codes are generated for this model. The first uses the full, nonlinear equations generated with the input shown in Figure 9.4.3. The other makes use of the knowledge that some of the variables are numerically small. To make the small-variable equations, the input of Figure 9.4.3 is modified as indicated in Figure 9.4.4. (Changes are shown in boldface.)

Describing the moments acting between bodies is made quite simple if the names of the state variables are known. The listings obtained from AUTOSIM are shown for reference in Tables 9.4.1 and 9.4.2.

The moments acting on the multibody system are described in the listing of Figure 9.2.5, which continues the input to AUTOSIM started in Figure 9.4.3.

The first two lines define moments acting on the boom (body F) from the bus (body B). The magnitudes of the moments are specified with equations defining simple torsional springs and dampers, involving the coordinates and speeds defined in Table 9.4.1 and 9.4.2.

In order to obtain expressions for the moments acting on the clock and camera, the subroutine that computes new control signals must be included. As shown by the listing in Figure 9.4.2, the subroutine CMD computes two control variables as functions of time. First, two variables are defined in AUTOSIM to pass as arguments to this subroutine, using the macro `add-variables`[1]. The macro indicates that (1) the variables will be

[1] If the `add-variables` macro were not used, AUTOSIM would assume that the symbols CLKCMD and CAMCMD are parameters, and would write code to read them from the input file. As it turns out, the simulation code would run correctly. However, its operation might be obscure to a person

used in the subroutine DIFEQN that AUTOSIM will soon generate, (2) the variables are REAL, and (3) there are two variables, called CLKCMD and CAMCMD. The next input, with the macro `add-subroutine`, instructs AUTOSIM to include the subroutine CMD when DIFEQN is written. Also, the arguments to CMD are specified.

```
(add-body B :name "Bus"
          :translate (1 2 3)
          :body-rotation-axes (1 2 3))

(add-body c :name "clock"
          :parent b
          :inertia-matrix #(ci ci 0)
          :body-rotation-axes 3
          :joint-coordinates #(0 0 !"-L1")
          :cm-coordinates #(0 0 L2))

(add-body d :name "Camera"
          :parent c
          :joint-coordinates #(0 !"-L3" 0)
          :cm-coordinates #(0 L5 L6)
          :body-rotation-axes 1
          :parent-rotation-axis #(-1 0 0))

(add-body e :parent b
          :Joint-coordinates #(0 !"-L7" 0)
          :inertia-matrix 0 :mass 0
          :parent-rotation-axis 3
          :body-rotation-axes 3)

(add-body f :name "Boom"
          :parent e
          :inertia-matrix #(FI1 FI2 FI1)
          :cm-coordinates #(0 !"-L8" 0)
          :parent-rotation-axis 1
          :body-rotation-axes 1)
```

Figure 9.4.3. Description of spacecraft bodies for AUTOSIM

Next, the two torques generated by the clock and camera motors are added. The first, CLOCKT, defines a torque acting between the bus and the clock. The magnitude is an expression involving the relative angular position of the clock, Q(7), the relative angular speed, U(7), and the control variable, CLKCMD. The second, CAMT, defines a similar torque acting between the clock and the camera.

The last three inputs describe the moments applied by the thruster pairs. The function THRUST, listed earlier, is referenced by name in the expressions for the magnitude of the active moments.

---

perusing the source code.

```
(add-body B :name "Bus"
          :small-angles (t t t)
          :small-translations (t t t)
          :translate (1 2 3)
          :body-rotation-axes (1 2 3))

(add-body e :parent b
          :Joint-coordinates #(0 !"-L7" 0)
          :inertia-matrix 0 :mass 0
          :small-angles (t)
          :parent-rotation-axis 3
          :body-rotation-axes 3)

(add-body f :name "Boom"
          :parent e
          :inertia-matrix #(FI1 FI2 FI1)
          :cm-coordinates #(0 !"-L8" 0)
          :small-angles (t)
          :parent-rotation-axis 1
          :body-rotation-axes 1)

;;; declare parameters “large” so moments resulting from “small”
;;; variables are not small.

(large kb bb)
```

Figure 9.4.4. Modifications to define “small” variables.

Table 9.4.1. Generalized coordinates for Spacecraft #1.

Q(1): Translation of B0 relative to the fixed origin along [n1]. (m)
Q(2): Translation of B0 relative to the fixed origin along [n2]. (m)
Q(3): Translation of B0 relative to the fixed origin along [n3]. (m)
Q(4): Rotation of Bpp relative to N about axis #1. (rad)
Q(5): Rotation of Bp relative to Bpp about axis #2. (rad)
Q(6): Rotation of B relative to Bp about axis #3. (rad)
Q(7): Rotation of the clock relative to the bus about axis #3. (rad)
Q(8): Rotation of the camera relative to the clock about axis #1. (rad)
Q(9): Rotation of E relative to the bus about axis #3. (rad)
Q(10): Rotation of the boom relative to E about axis #1. (rad)

Table 9.4.2. Independent speeds for Spacecraft #1.

U(1): Abs. trans. speed of B* along axis 1. (m/s)
U(2): Abs. trans. speed of B* along axis 2. (m/s)
U(3): Abs. trans. speed of B* along axis 3. (m/s)
U(4): Abs. rotation of B about axis #1. (rad/s)
U(5): Abs. rotation of B about axis #2. (rad/s)
U(6): Abs. rotation of B about axis #3. (rad/s)
U(7): Rot. of relative to B, axis 3. (rad/s)
U(8): Rot. of D relative to C, axis 1. (rad/s)
U(9): Rot. of E relative to B, axis 3. (rad/s)
U(10): Rot. of F relative to E, axis 1. (rad/s)

```
;;; Add moments that are due to flexing of the boom

(add-moment bt1 :name "boom-torque Z"
    :direction [e3] :body1 f :body2 b
    :magnitude !"-kb*q(9) - bb*u(9)")

(add-moment bt2 :name "boom-torque X"
    :direction [f1] :body1 f :body2 b
    :magnitude !"-kb*q(10) - bb*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 :name "torque from clock motor"
    :direction [c3] :body1 c :body2 b
    :magnitude !"kclock*(-q(7) + clkcmd) - bclock*u(7)")

(add-moment camt :name "torque from camera motor"
    :direction [d1] :body1 d :body2 c
    :magnitude !"kclock*(-q(8) + camcmd) - bclock*u(8)")

;;; add moments from thrusters

(add-moment tt1 :name "thruster moment #1"
    :direction [b1]  :body1 b
    :magnitude !"ltt1*func(thrust, t, 1, (gyro*u(4) + q(4)))")

(add-moment tt2 :name "thruster moment #2" :direction [b2] :body1 b
    :magnitude !"ltt2*func(thrust, t, 2, (gyro*u(5) + q(5)))")

(add-moment tt3 :name "thruster moment #3" :direction [b3]
    :magnitude !"ltt3*func(thrust, t, 3, (gyro*u(6) + q(6)))"
    :body1 b)
```

Figure 9.4.5. AUTOSIM description of active moments.

The remaining inputs to AUTOSIM are shown in the listing of Figure 9.4.6. They specify (1) that the units system is metric, (2) default values for the parameters, (3) that the simulation code should include all coordinates, speeds, and moments as output variables, and (4) that the dynamics analysis should be performed.

```
(mks)
(set-defaults L1 1.5 L2 .75 L3 .1 L5 .22 L6 .2 L7 1.2 L8 3.3
              BM 410  CM 6.8 DM 57.5 FM 10.7
              BI11 115 BI12 -14 BI13 14
              BI22 316 BI23 -34.6 BI33 440
              CI .35
              DI11 4.85 DI12 -0.41 DI13 -.07
              DI22 2.2 DI23 -0.54 DI32 -0.54 DI33 5.5
              FI1 27.2 FI2 0.2
              LTT1 .23 LTT2 .21 LTT3 .31 GYRO 2
              KCLOCK 3500 BCLOCK 20 KB 2000 BB 10
              STEP .02 STOPT 30)

(add-coordinates-to-output)
(add-speeds-to-output)
(add-moments-to-output)
(dynamics)
```

Figure 9.4.6. Define units, default values, output variables, and name of multibody system.

Results

Time histories from the simulated slew maneuver are shown in Figures 9.4.7 and 9.4.8. Performances of the different simulation codes are summarized in Table 9.4.3.

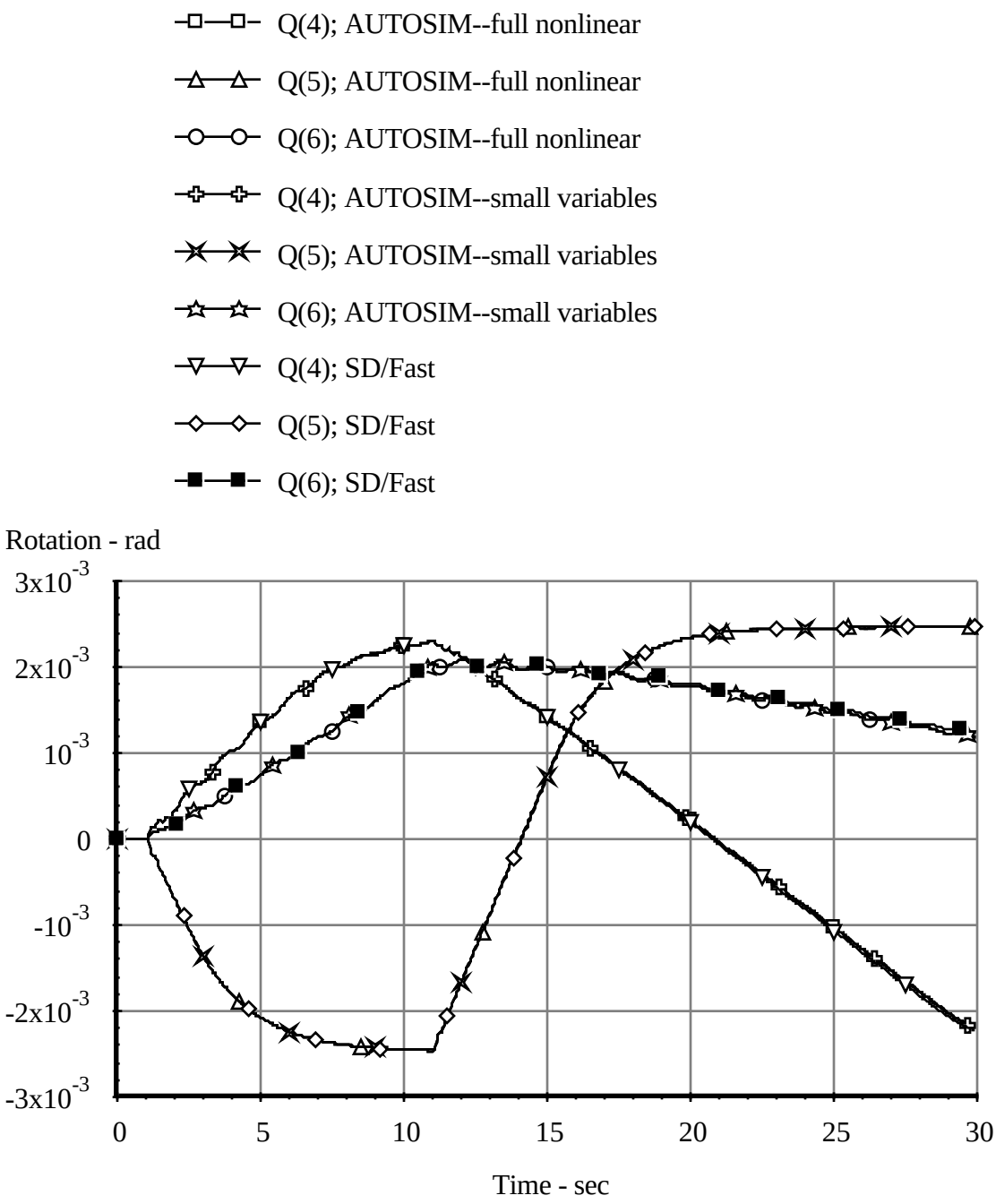

Figure 9.4.7. Time histories of satellite attitude variables during slew maneuver.

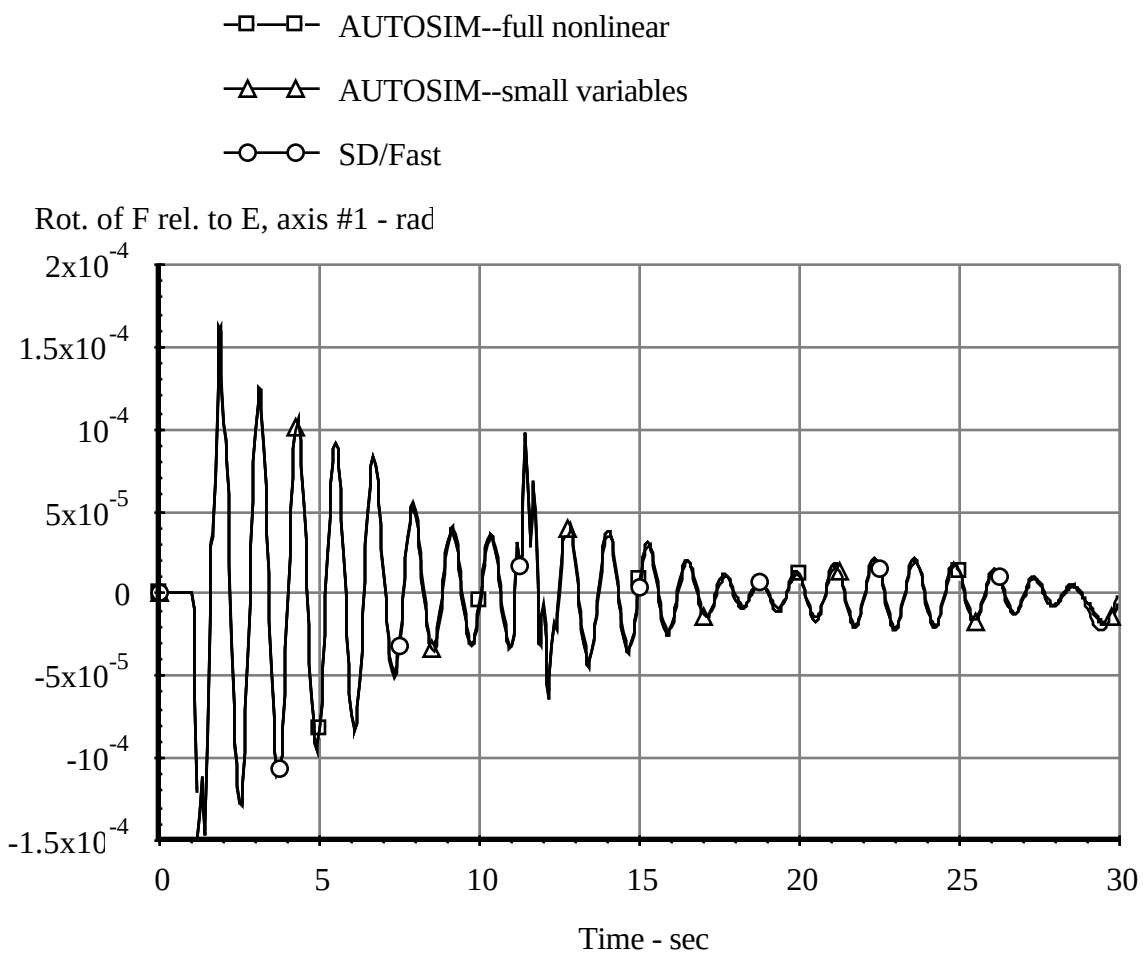

Figure 9.4.8. Time histories of boom deflection during slew maneuver.

Table 9.4.3. Performance comparisons between three simulation codes.

Source	adds and subtracts	multiplies, divides, and function calls
SD/FAST Users Manual[1]	709	1094
AUTOSIM, using full, nonlinear formulation	628	791
AUTOSIM, using small variables for 8 d.o.f.	442	514

[1] The subroutine SDNSIM contains 920 multiply/divides, 576 add/subtracts, and 14 trig function evaluations. The solution of 7 simultaneous equations adds 139 multiply/divides and 126 add/subtracts. The DERIV subroutine and additional routines add 14 multiplies, 7 adds, and 6 function/subroutine calls.

9.5. "Spacecraft #2"

A spacecraft model with 10 degrees of freedom was used to demonstrate the methods used by Nielan in his SYMBA symbolic analysis code. This example is provided mainly to compare the efficiencies of AUTOSIM and SYMBA for spacecraft vehicles.

Model Description

The spacecraft is composed of five rigid bodies: a main body W and four antennas, A1, A2, A3, and A4. Dimensions and points in the spacecraft are shown in figure 9.5.1.

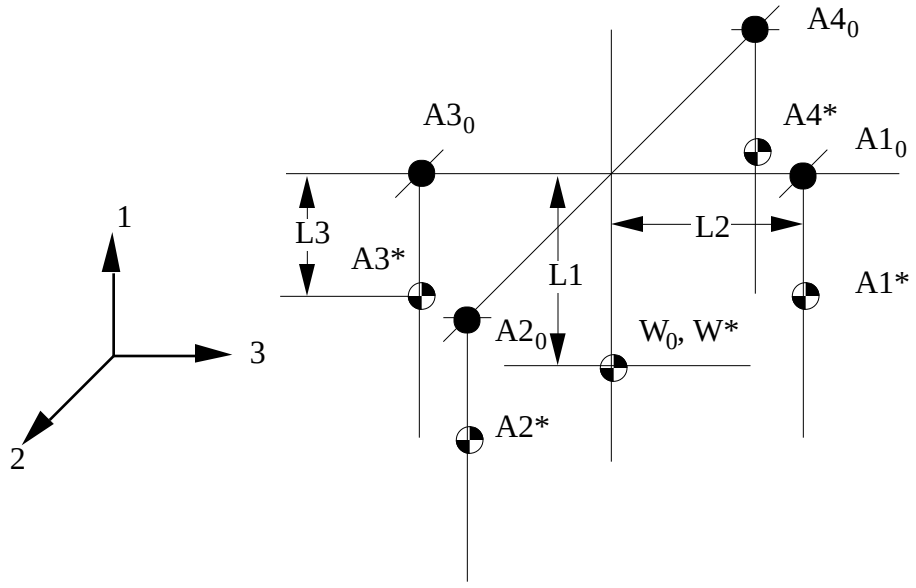

Figure 9.5.1. Dimensions of "Spacecraft #2."

The hinge points for the four antennas are located at points $A1_0$, $A2_0$, $A3_0$, and $A4_0$. The coordinates of those points are (L1, 0, L2), (L1, L2, 0), (L1, 0, –L2), and (L1, –L2, 0), respectively, in the coordinate system of W. The centers of mass of the four antennas are located a distance L3 from the hinge points, as shown.

The antenna hinges have torsional stiffness K and damping rate D. The rotation axes for A1 and A3 lie parallel with the #3 axis of W, for A2 and A4 the rotation is parallel with the #2 axis. Given the initial conditions of the antennas all aligned as shown, and an initial angular rotation vector for W, the objective of the simulation is to view the time histories of the body velocity components and the antenna angular displacements.

AUTOSIM Description

The description of this system in AUTOSIM is straightforward and is presented in Figure 9.5.2. All of the inputs have been described in previous examples and should be familiar to the reader. The state variables introduced for the system are listed in Table 9.5.1. (They appear in the expressions for the moments applied by the torsional springs and dampers.)

```
(add-body W
  :inertia-matrix #(Ixx Iyy Izz)
  :translate (1 2 3)
  :body-rotation-axes (3 2 1)
  :mass m1)

(add-body a1 :parent w
  :inertia-matrix #(0 ia ia)
  :mass m2
  :joint-coordinates #(L1 0 L2)
  :cm-coordinates #(L3 0 0)
  :body-rotation-axes (2))

(add-body a2 :parent w
  :inertia-matrix #(0 ia ia)
  :mass m2
  :joint-coordinates #(L1 L2 0)
  :cm-coordinates #(L3 0 0)
  :body-rotation-axes (3))

(add-body a3 :parent w
  :inertia-matrix #(0 ia ia)
  :mass m2
  :joint-coordinates
              #(L1 0 !"-L2")
  :cm-coordinates #(L3 0 0)
  :body-rotation-axes (2))

(add-body a4 :parent w
  :inertia-matrix #(0 ia ia)
  :mass m2
  :joint-coordinates
              #(L1 !"-L2" 0)
  :cm-coordinates #(L3 0 0)
  :body-rotation-axes (3))
```

```
(add-moment t1
    :direction [w2]
    :magnitude
         !"-K*q(7) - D*u(7)"
    :body1 a1 :body2 w)

(add-moment t2
    :direction [w3]
    :magnitude
         !"-K*q(8) - D*u(8)"
    :body1 a2 :body2 w)

(add-moment t3
    :direction [w2]
    :magnitude
          !"-K*q(9) - D*u(9)"
    :body1 a3 :body2 w)

(add-moment t4
    :direction [w3]
    :magnitude
          !"-K*q(10) - D*u(10)"
    :body1 a4 :body2 w)

(mks)
(setf *multibody-system-name* "Symba
    spacecraft")

(set-defaults IXX 110 IYY 100 IZZ 70
    IA .02 M1 500 M2 2
    K .0000285 D .001359
    L1 .5 L2 .3 L3 .2)

(add-coordinates-to-output)
(add-speeds-to-output)
```

Figure 9.5.2. Description of Spacecraft #2 in AUTOSIM.

Table 9.5.1. State variables for Spacecraft #2.

Generalized Coordinates

Q(1): Translation of W0 relative to the fixed origin along [n1]. (m)
Q(2): Translation of W0 relative to the fixed origin along [n2]. (m)
Q(3): Translation of W0 relative to the fixed origin along [n3]. (m)
Q(4): Rotation of Wpp relative to N about axis #3. (rad)
Q(5): Rotation of Wp relative to Wpp about axis #2. (rad)
Q(6): Rotation of W relative to Wp about axis #1. (rad)
Q(7): Rotation of A1 relative to W about axis #2. (rad)
Q(8): Rotation of A2 relative to W about axis #3. (rad)
Q(9): Rotation of A3 relative to W about axis #2. (rad)
Q(10): Rotation of A4 relative to W about axis #3. (rad)

Independent Speeds

U(1): Abs. trans. speed of W* along axis 1. (m/s)
U(2): Abs. trans. speed of W* along axis 2. (m/s)
U(3): Abs. trans. speed of W* along axis 3. (m/s)
U(4): Abs. rotation of W about axis #3. (rad/s)
U(5): Abs. rotation of W about axis #2. (rad/s)
U(6): Abs. rotation of W about axis #1. (rad/s)
U(7): Rot. of A1 relative to W, axis 2. (rad/s)
U(8): Rot. of A2 relative to W, axis 3. (rad/s)
U(9): Rot. of A3 relative to W, axis 2. (rad/s)
U(10): Rot. of A4 relative to W, axis 3. (rad/s)

Results

The vehicle was simulated with the initial conditions U(4) = .0017 rad/sec, U(5) = U(6) = .00017 rad/sec. Figure 9.5.3 shows time histories of the first 1000 seconds in response to those initial conditions.

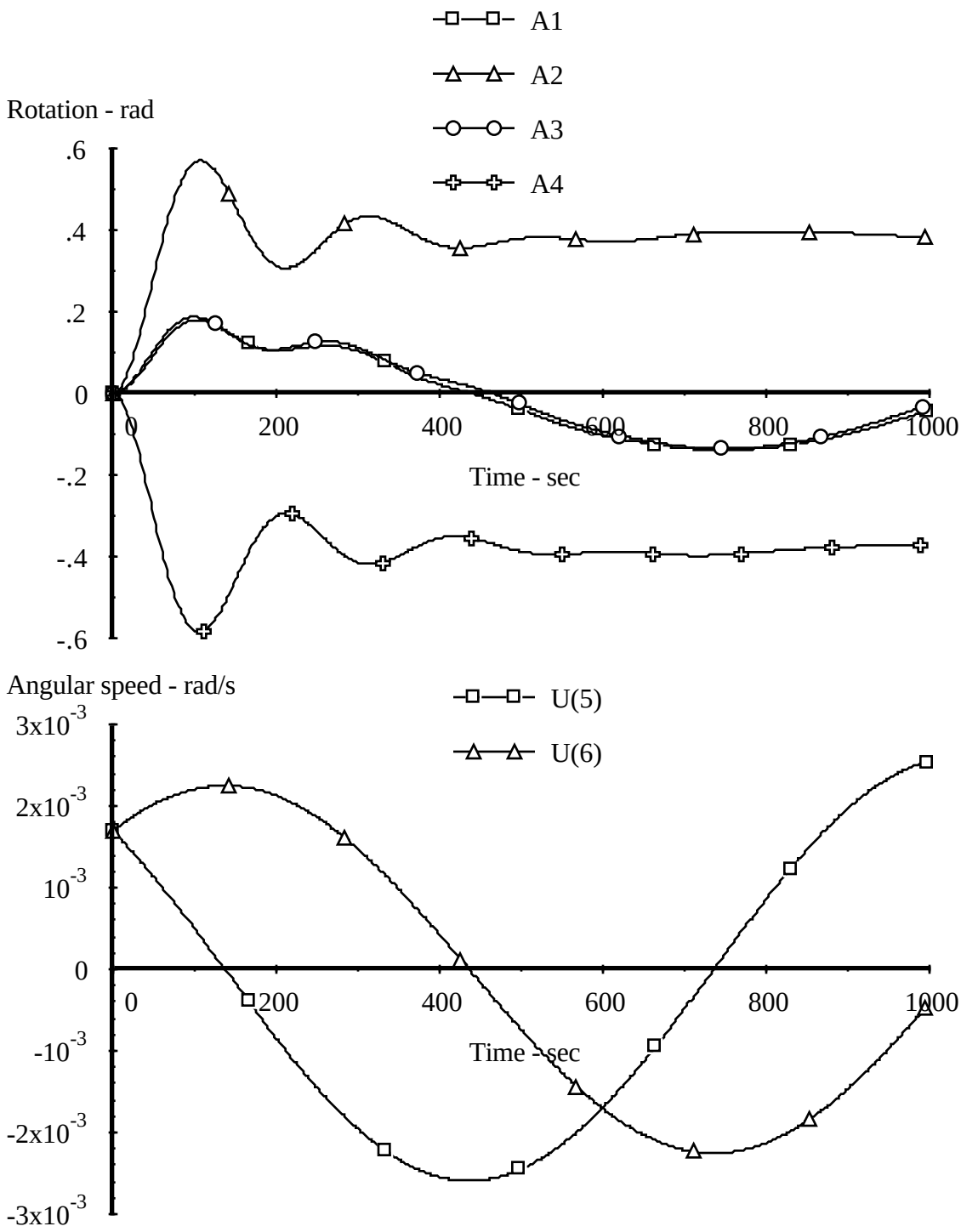

Figure 9.5.3. Time histories for Spacecraft #2.

The numerical efficiencies of the simulation codes generated by AUTOSIM and SYMBA are compared in Table 9.5.2.

Table 9.5.2. Performance comparisons for Spacecraft #2.

Source	adds and subtracts	multiplies, divides, and function calls
SYMBA [83]	514	760
AUTOSIM	338	455

9.6. The "Stanford Arm" Manipulator

The Stanford Arm is a robot with six degrees of freedom that is of interest here because it has been used to benchmark various methods for forming equations of motion. In contrast to most vehicle systems, the topology is a "chain," in which each body except one has one and only one child. It is included here to compare the efficiency of the methods developed in this dissertation with formulations for manipulators that have been published.

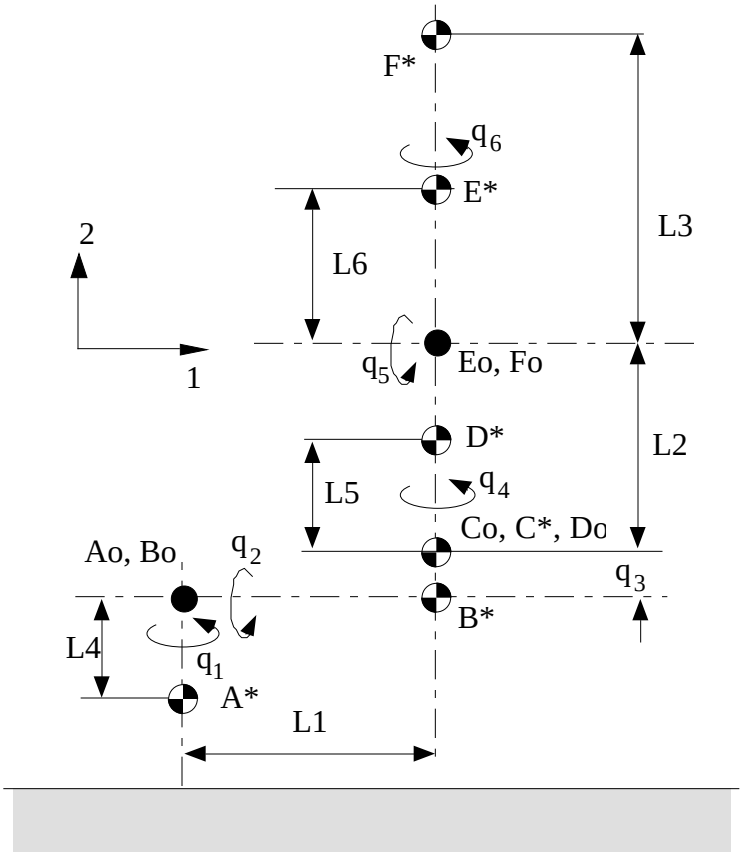

Figure 9.6.1. Sketch of "Stanford Arm" points, dimensions, and coordinates.

Model Description

The robot is composed of six rigid bodies labeled simply bodies A,B,C,D,E, and F for the AUTOSIM input. The geometry is sketched in Figure 9.6.1. Bodies A, B, D, E, and F each have a single rotational degree of freedom shown in the figure. Body C has a single translational degree of freedom. The generalized coordinates introduced by AUTOSIM, q_1, … q_6, are also shown in the figure. Each joint has an associated servo-motor. The torques produced for the five rotational degrees of freedom are designated τ_1, … τ_5 and the force produced for the translational degree of freedom is designated σ. Controller torques and forces have been defined as follows [57]:

$$\tau_1 = -[k_1 (q_1 - q_1^*) + k_2 \dot{q}_1] \tag{9.6.1}$$

$$\tau_2 = -\left[k_3(q_2 - q_2^*) + k_4\dot{q}_2 + g\left\{ [m_C q_3 + m_D(q_3 + L_5) + (m_E + m_F)(q_3 + L_2)]s_2 + (m_E L_6 + m_F L_3)(c_5 s_2 + c_4 s_5 c_2) \right\} \right] \tag{9.6.2}$$

$$\tau_3 = -[k_5 (q_4 - q_4^*) + k_6 \dot{q}_4 - g(m_E L_6 + m_F L_3) s_2 s_4 s_5] \tag{9.6.3}$$

$$\tau_4 = -[k_7 (q_5 - q_5^*) + k_8 \dot{q}_5 + g(m_E L_6 + m_F L_3)(c_2 s_5 + s_2 c_4 c_5)] \tag{9.6.4}$$

$$\tau_5 = -[k_9 (q_6 - q_6^*) + k_{10} \dot{q}_6] \tag{9.6.5}$$

$$\sigma = -[k_{11} (q_3 - q_3^*) + k_{12} \dot{q}_3 - g(m_C + m_D + m_E + m_F) c_2] \tag{9.6.5}$$

where c_2, c_4, c_5, s_2, s_4, and s_5 represent cosine and sine functions of the generalized coordinates q_2, q_4, and q_5; k_1, … k_{12} are feedback controller gains; and q_1^*, … q_6^* are the desired final values of the coordinates.

The initial conditions are that all speeds are zero, five of the generalized coordinates are zero, and $q_2 = \pi/2$. The final values for the rotational coordinates are $q_1^* = q_2^* = q_4^* = q_5^* = q_6^* = \pi/3$, and the final value for q_3 is 0.1 m. Values of the simulation parameters are taken from [57], and are shown in Table 9.6.1 for the names appearing in the AUTOSIM input. (Note that the dimension L4 is not in the table, because it does not appear in the equations of motion. Also, moments of inertia A1 and A2 do not appear.) The target rotations are designated AROT, BROT, DROT, EROT, and FROT, while the target displacement is CDISP.

Table 9.6.1. Parameters and values for Stanford Arm.

A2	0.200000E-01	moment of inertia of A (kg-m2)
AROT	1.04720	target rotation of A (rad)
B1	0.600000E-01	moment of inertia of B (kg-m2)
B2	0.100000E-01	moment of inertia of B (kg-m2)
B3	0.500000E-01	moment of inertia of B (kg-m2)
BROT	1.04720	target rotation of B (rad)
C1	.400000	moment of inertia of C (kg-m2)
C2	0.100000E-01	moment of inertia of C (kg-m2)
C3	.400000	moment of inertia of C (kg-m2)
CDISP	.100000	target displacement of C (m)
D1	0.500000E-03	moment of inertia of D (kg-m2)
D2	0.100000E-02	moment of inertia of D (kg-m2)
D3	0.100000E-02	moment of inertia of D (kg-m2)
DROT	1.04720	target rotation of D (rad)
E1	0.500000E-03	moment of inertia of E (kg-m2)
E2	0.200000E-03	moment of inertia of E (kg-m2)
E3	0.500000E-03	moment of inertia of E (kg-m2)
EROT	1.04720	target rotation of E (rad)
F1	0.100000E-02	moment of inertia of F (kg-m2)
F2	0.200000E-02	moment of inertia of F (kg-m2)
F3	0.300000E-02	moment of inertia of F (kg-m2)
FROT	1.04720	target rotation of F (rad)
K1	3.00000	stiffness coefficient (N-m)
K2	5.00000	damping coefficient (N-m-s)
K3	1.00000	stiffness coefficient (N-m)
K4	3.00000	damping coefficient (N-m-s)
K5	.300000	stiffness coefficient (N-m)
K6	.600000	damping coefficient (N-m-s)
K7	.300000	stiffness coefficient (N-m)
K8	.600000	damping coefficient (N-m-s)
K9	.250000	stiffness coefficient (N-m)
K10	.250000	damping coefficient (N-m-s)
K11	30.0000	stiffness coefficient (N/m)
K12	41.0000	damping coefficient (N-s/m)
L1	.100000	coordinate of center of mass of B in dir 1 (m)
L2	.600000	coord. of attachment pt. for E in dir 2 (m)
L3	.200000	coordinate of center of mass of F in dir 2 (m)
L5	.700000	coordinate of center of mass of D in dir 2 (m)
L6	0.600000E-01	coordinate of center of mass of E in dir 2 (m)
MA	9.00000	mass of A (kg)
MB	6.00000	mass of B (kg)
MC	4.00000	mass of C (kg)
MD	1.00000	mass of D (kg)
ME	.600000	mass of E (kg)
MF	.500000	mass of F (kg)
STOPT	10.0000	simulation stop time (sec)

AUTOSIM Description

The description to AUTOSIM for the uncontolled system is shown in the listing of Figure 9.6.2. Note that the direction for the gravitational field is changed from the default ([n3]) to –[n2] for compatibility with the coordinate systems shown in Figure 9.6.1.

```
(add-body a :body-rotation-axes 2  :mass ma
          :inertia-matrix #(a1 a2 a3) :cm-coordinates #(0 !"-L4" 0))

(add-body b :parent a :body-rotation-axes 1  :mass mb
          :inertia-matrix #(b1 b2 b3) :cm-coordinates #(L1 0 0))

(add-body c :parent b :inertia-matrix #(c1 c2 c3) :mass mc
          :translate 2 :joint-coordinates #(L1 0 0))

(add-body d :parent c :body-rotation-axes 2  :mass md
          :inertia-matrix #(d1 d2 d3) :cm-coordinates #(0 L5 0))

(add-body e :parent d :body-rotation-axes 1  :mass me
          :inertia-matrix #(e1 e2 e3)
          :joint-coordinates #(0 L2 0) :cm-coordinates #(0 L6 0))

(add-body f :parent e :body-rotation-axes 2  :mass mf
          :inertia-matrix #(f1 f2 f3) :cm-coordinates #(0 L3 0))

(add-gravity :direction !"-[n2]")
```

Figure 9.6.2. Description of uncontrolled Stanford Arm.

The controller torques and force (from eqs. 9.6.1 through 9.6.6) are described with the inputs shown in Figure 9.6.3. The controller rules from eqs. 9.6.1 through 9.6.6 are entered directly as expressions for the `:magnitude` argument of the `add-moment` and `add-line-force` macros.

Results

The time history plots obtained by the simulation code are shown in Figure 9.6.4 and agree with results published earlier [57].

The efficiency of the simulation code generated by AUTOSIM is compared with other formulations in Table 9.6.2. The operation counts in this table have been published from a variety of sources, and were summarized by Neilen and Kane previously. In their summaries, the computation needed to uncouple the dynamical equations was not included and therefore more operations are shown here: 65 adds and 86 multiply/divides. (A small savings is obtained here, as the symbolic solution involves only 74 multiplications).

Because none of the other counts include the controller equations (eqs. 9.6.1 through 9.6.6), the AUTOSIM results are for the uncontrolled system. (The additional arithmetic operations generated by AUTOSIM when the control equations are included are: 28 add/subtracts and 25 multiply/divides. These are fewer than appear in eqs. 9.6.1 through

9.6.6, due to the use of intermediate expressions that arise elsewhere in the equations of motion.)

```
(add-moment tau1 :name "torque applied to A"
    :direction [n2] :body1 a
    :magnitude !"-K1*(Q(1) - AROT) - K2*QP(1)")

(add-moment tau2 :name "torque applied to B"
    :direction [a1] :body1 b :body2 a
    :magnitude
      !"-(K3*(Q(2) - BROT) + K4*QP(2) + GEES*
           (((MC+MD)*Q(3) + MD*L5)*SIN(Q(2))
              + (ME*L6 + MF*L3)
                *(COS(Q(5))*SIN(Q(2)) + COS(Q(4))*SIN(Q(5))*COS(Q(2)))
              + (ME + MF)*(Q(3) + L2)*SIN(Q(2))))")

(add-moment tau3 :name "torque applied to D"
    :direction [d2]  :body1 d :body2 c
    :magnitude
      !"-(K5*(Q(4)-DROT) + K6*QP(4)
           - GEES*(ME*L6 + MF*L3)*SIN(Q(2))*SIN(Q(4))*SIN(Q(5)))")

(add-moment tau4 :name "torque applied to E"
    :direction [e1] :body1 e :body2 d
    :magnitude
      !"-(K7*(Q(5) - EROT) + K8*QP(5) + GEES*(ME*L6 + MF*L3)
           *(COS(Q(2))*SIN(Q(5)) + SIN(Q(2))*COS(Q(4))*COS(Q(5))))")

(add-moment tau5 :name "torque applied to F"
    :direction [f2] :body1 f :body2 e
    :magnitude !"-(K9*(Q(6) - FROT) + K10*QP(6))")

(add-line-force sigma :name "force applied to C"
    :direction [c2] :point1 c0 :point2 b0
    :magnitude
      !"-(K11*(Q(3) - CDISP) + K12*QP(3)
              - GEES*(MC+MD+ME+MF)*COS(Q(2)))")
```

Figure 9.6.3. Description of control torques and force for Stanford Arm.

In viewing Table 9.6.1, keep in mind that all but one of the formulations are based on Kane's equations. The formulation obtained by AUTOSIM is the most efficient known, being almost twice as efficient as the first formulation of this sort, obtained manually by Kane and Levinson [57].

The equations of motion produced by AUTOSIM are included in Appendix E.

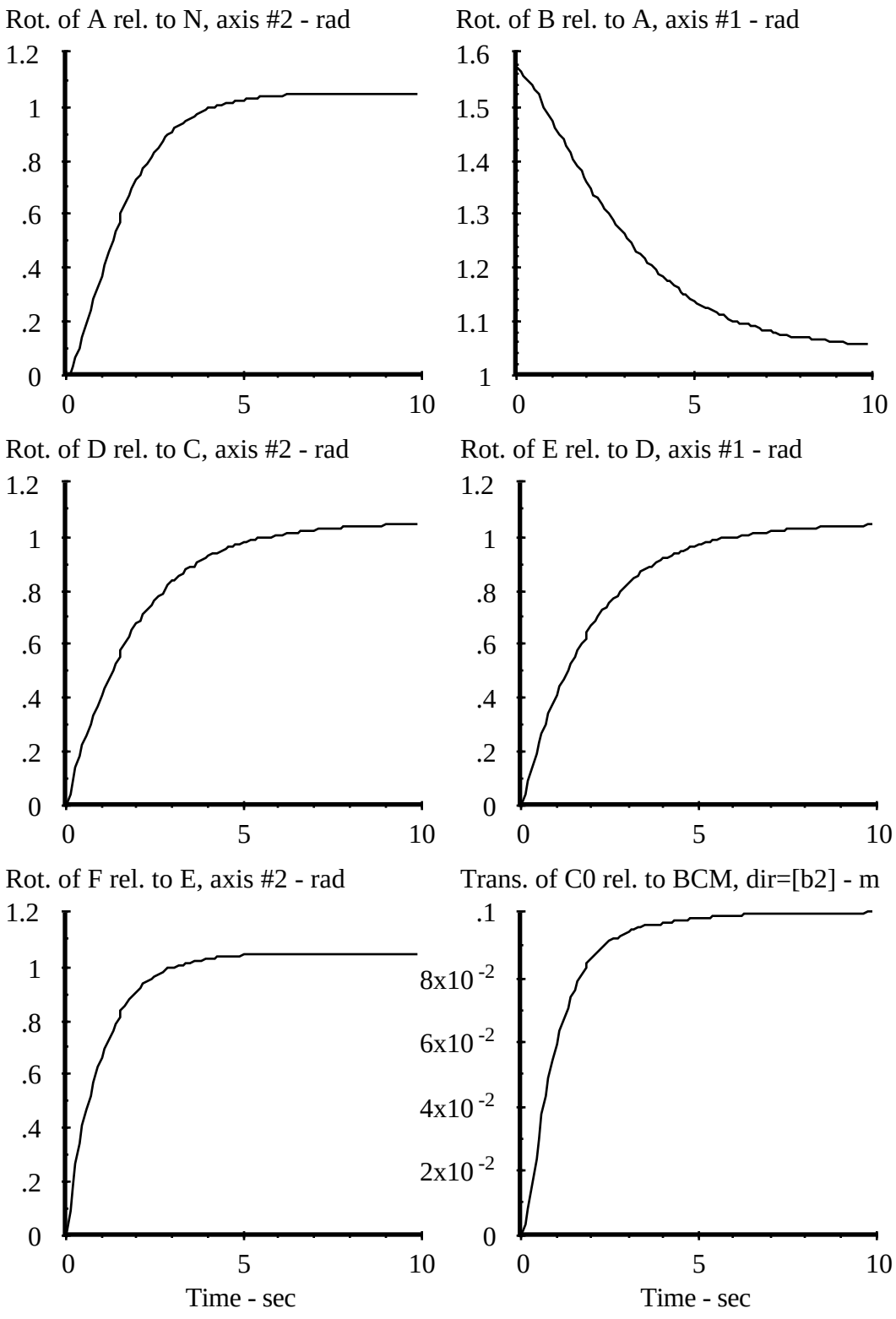

Figure 9.6.4. Time history plots of generalized coordinates.

Table 9.6.1. Performance comparisons between four simulation codes.

Formulation for uncontrolled system*	**adds and subtracts**	**multiplies and divides**
Symbolic using Macsyma (Hussain and Noble) [44]	1902	5406
Numerical Newton-Euler (Walker and Orin) [124]	1255	1627
Symbolic using Macsyma (Kane and Nielan) [82]	521	858
Symbolic, by hand (Kane and Levinson) [57]	459	732
SD/EXACT (Rosenthal and Sherman) [83]	465	718
SD/FAST (Rosenthal and Sherman) [99]	390	576
Symbolic, by hand (Wampler) [125]	318	448
SYMBA (Nielan) [83]	268	384
AUTOSIM	240	353

* for AUTOSIM, the control equations add 28 add/subtracts and 25 multiply/divides. The control equations also add to the other formulations, but the amount is not known.

10. SUMMARY AND CONCLUSIONS

This chapter summarizes the preceding material and then presents conclusions. Possible directions for continuing the research are noted.

10.1 Summary

Simulation codes for ground vehicles and other multibody systems assembled from rigid bodies, joints, and massless force- and moment-producing components have previously been programmed by hand for specific systems. Also, general-purpose programs have been developed to simulate classes of multibody systems. In the latter approach, the equations of motion are developed for each system according to a multibody formalism. The multibody formalisms used to automatically formulate equations, whether numerically or symbolically, have not been representative of how human analysts formulate the equations. They have represented analysis strategies that can be programmed easily, whereas the human analyst usually applies modeling and engineering knowledge to simplify the work. Simulation codes for ground vehicles that are developed by hand can run orders of magnitude faster than popular general-purpose codes simulating the same model. Hence, for applications in which computation time is critical, such as real-time hardware-in-the-loop simulations, or simulations run on desktop computers, simulations are coded by hand because there is no alternative.

In this dissertation, a multibody formalism was developed that includes methods and concepts employed by human analysts. The formalism is built on Kane's method (written for students of dynamics), and then extended with specific tactics for (1) choosing state variables, (2) defining appropriate forms of vector representations, (3) grouping rigid bodies together and choosing coordinate systems so as to simplify expressions that later arise, and (4) obtaining equations of motion for numerical solution. This formalism is made possible by developing a new computer language called AUTOSIM to represent a multibody system and the vector and dyadic expressions involved in its description. Computer data objects are defined in Chapter 5 for representing (1) symbolic algebraic expressions for vector/dyadic analyses, (2) physical components in a multibody system,

and (3) program structures needed in a simulation code. With these representations, the multibody formalism is programmed almost exactly as it is presented in Chapter 8.

AUTOSIM is capable of automatically producing equations of motion in symbolic form for multibody systems that cannot be represented with most symbolic multibody analysis methods. The analyst using AUTOSIM can handle systems with arbitrarily oriented forces and moments, nonholonomic constraints, and closed kinematical loops. The forces, moments, and output variables can involve external subroutines linked to experimental measurements, unconventional models, interfaces with hardware in the loop, etc. Constraints are described using high-level representations that relieve the analyst of the task of manually forming constraint equations or obtaining matrix coefficients.

Closed kinematical loops are handled by a blend of analytical and computational methods. Constraint equations that are too complicated to yield closed-form analytical solutions are used to create recursive numerical procedures that compute certain dependent variables called “computed coordinates.”

The equations formed by AUTOSIM are simplified when possible to remove terms that are negligible when one or more variables or parameters are known to be “small.” Parameters and constants are handled symbolically, and are factored out of the equations when possible so that they can be “precomputed.” As a result, the equations have the same efficiency as would be obtained if numerical values were specified for all parameters (if the numerical values are not zero or one). However, since the parameters are represented by symbols, the same simulation code can be used for any set of valid parameter values.

Six example multibody systems were analyzed. The first three examples could not have been analyzed symbolically with the automation level demonstrated here using previously available methods. The other three examples were systems that have been analyzed with other symbolic multibody computer codes. In the three latter cases, the equations derived by AUTOSIM were more efficient than the most efficient formulation previously published by amounts ranging from 6% (the robot manipulator) to a factor of 2 (the spacecraft in Section 9.4).

10.2 Conclusions

The above results lead to the following conclusions:

1. Kane’s method of analyzing dynamic systems of constrained rigid bodies is easily extended to a form required for computer solution.

2. Methods and concepts used by human analysts can be programmed into a multibody formalism.

3. The derivation of constraint equations and the coefficients needed to form the dynamical equations of a constrained system can be automated.

4. Advances in computer software and hardware permit the above methods to be programmed on desktop computers. This diminishes the need for multibody formalisms designed to fit within the limits of traditional computer languages at the expense of versatility and efficiency.

5. Extremely high efficiency in simulation codes is obtained when the above methods are applied.

6. Regarding the attention to the three areas of (1) the rigid-body dynamics formalism, (2) the algebraic manipulation methods, and (3) the generation of numerical computation code: the general strategy of spreading the analysis methods over these areas permits the use of relatively simple methods within each area.

7. A symbolic analysis method that closely parallels the approach taken by a human analyst may be easier to use than other types of analysis languages, because the automated part of the analysis begins from a description of the problem in terms that are most familiar to the analyst.

Several limitations of the symbolic analysis approach should also be mentioned. First, there is an underlying assumption that for a given system, the correct equations of motion can be derived for once and for all. With some models the system gains or loses degrees of freedom. For example, a man walking has fewer degrees of freedom when both feet are on the ground than when one or two feet are in the air. For some mechanisms, a set of equations obtained to best describe motions about a nominal configuration becomes singular for other configurations. In these cases, generalized numerical codes that change equations “on the fly” may be preferable.

Symbolic analysis methods have had a reputation for being unsuitable for large systems. When using general-purpose computer algebra languages (e.g., Macsyma, Reduce), this is indeed a serious limitation because the languages maintain complete expressions that grow rapidly unless the analyst makes an effort to break up the analysis into segments. With symbolic analyses programmed especially for multibody systems (e.g., AUTOSIM, SD/FAST) this is much less of a problem because expressions are kept

to a manageable size as the analysis proceeds. (In AUTOSIM this is done through the introduction of intermediate variables. In other codes it is done by printing equations as they are derived, and immediately recovering the computer memory that they occupied.) Also, with the widespread use of virtual memory on workstations, memory requirements of several hundred megabytes can be accommodated if necessary. (As a point of reference, the examples in Chapter nine were all performed with a memory limit of 3 MB, and took one to ten minutes of computation time on an Apple Macintosh.)

Although much larger systems can be analyzed symbolically with programs such as AUTOSIM than was possible ten years ago, the performance of AUTOSIM has not yet been investigated for systems that involve more than seven thousand arithmetic operations. (Note that the limiting factor is not the number of bodies, nor the number of degrees of freedom. It is the complexity of the equations, measured approximately by the number of multiply/divide arithmetic operations contained in the equations, that most directly determines the memory and time needed to analyze a system.)

Equations for some multibody systems can be formulated in a numerical code using iteration loops, so that all of the equations are not explicitly formed as they are in AUTOSIM. (For example, a pure chain topology.) For these systems, the explicit formulation developed according to the methods presented in this dissertation might be too lengthy to be derived in a reasonable time.

Since 1980, there have been a number of dynamics formalisms developed for chain topologies that yield explicit equations of motion without forming a mass matrix. As the number of bodies in the chain becomes large, the computation needed to obtain the derivatives of the state variables grows in proportion to the number of links. Hence, these algorithms are called “Order(n) formulations.” For chains of length six and less, these formulations typically require more computation than methods leading to implicit equations (such as presented here). However, for large systems, the decomposition of the mass matrix requires an ever greater effort, that grows in proportion to n^3. Therefore, the method presented here is probably not the most efficient when dealing with long chains of bodies.

10.3 Further Research Opportunities

There are many possibilities for extending the methods developed in this work to other forms of analysis. Two general areas of application are in (1) analytical methods, and (2) numerical methods. In the area of analytical methods, virtually any analysis involving moving reference frames can be programmed directly using the algebra functions defined in Chapter 5. Although this work was limited to the development of efficient simulation codes, the computer algebra methods can be applied to such applications as:

1. Equations for the inverse dynamic problem. (That is, forces and torques in a multibody system are computed as needed to cause known movements.)
2. Symbolic solution of nonlinear statics problems. A Jacobian matrix is formed to solve for forces and torques and equilibrium position, given known values of independent coordinates.
3. Derivation of matrices needed as inputs for other software (e.g., mass, stiffness, and damping matrices for the linearized equations of motion are inputs for popular controller design software).
4. Analysis of constrained systems to define alternative formulations and code for switching between the formulations to avoid singularities.

Applications involving numerical analysis include the following:

1. The simulation codes generated by AUTOSIM can be produced in languages other than Fortran, such as simulation languages (ACSL, ADSIM, etc.) or other programming languages (C, Pascal, Lisp, etc.).
2. The output code can be tailored for a particular computer architecture. For example, the method used to remove unused code (described in Section 5.3) can be modified slightly to isolate sections of code that can be computed independently of each other, in support of parallel processors.
3. The equations can be tailored to novel numerical integration methods. The most critical parameter in determining simulation efficiency is the step size of the numerical integration. Methods that allow larger time steps to be used for the equations of motion, or even portions of the equations, provide a great potential for improving computation speed. Combinations of analytical and computational methods, such as the one used for updating “computed coordinates” in this work,

offer great promise for using symbolically generated numerical algorithms to improve the accuracy and efficiency of the equations.

4. "Interface software" needed to link related analyses can be written automatically. One possibility is to generate subroutines to link with finite element codes to combine rigid-body motions with deformations of flexible bodies. Another possibility is the handling of systems that change degrees of freedom, but which have a small number of possible combinations (e.g., a walking man). Equations could be formulated for each configuration, and the appropriate equations set would be selected "on the fly" during the simulation. A subroutine would be needed to map the values of the state variables for one equation set onto a proper set of initial conditions for another set of equations.

APPENDICES

APPENDIX A — AUTOSIM REFERENCE

The AUTOSIM software runs under Common Lisp. The user interface is that of the Lisp system on which AUTOSIM is installed, and all AUTOSIM commands are Lisp forms. Lisp syntax is detailed fully in many introductory textbooks, and usually in the reference material provided with the Lisp software. However, is not necessary to be fluent in Lisp to use AUTOSIM. The syntax of Lisp is so simple that the basics should be fairly apparent from the examples in Chapter 9. Additionally, selected reference information about the the Lisp environment is presented in this appendix for the reader who is unfamiliar with Lisp and desires "just enough" information to fully understand the examples in Chapter 9. The summary of Lisp is followed by descriptions of the AUTOSIM functions and macros used to describe and analyze a multibody system.

A.1. Brief Summary of Lisp Syntax and Data Types

Working in a Lisp environment is very simple: the analyst types in a "Lisp object" and the machine prints the value of that object. Lisp objects used in AUTOSIM are the following:

- number — numbers are entered and printed as might be expected. Although Lisp has separate representations for different types of numbers (floating point, integer, ratio, etc.) it is not necessary that the analyst using AUTOSIM be concerned with this.
- symbol — a symbol is a fundamental data object in Lisp that has a name and possibly a value. It is described in more detail below.
- list — a list is a sequence of zero or more objects enclosed by parentheses and separated by spaces. Many forms of data are well represented by lists, such as the terms in a sum, the factors in a product, etc. A list by itself is treated as a "Lisp form" and is "evaluated," as described below.
- string — a string is a sequence of alphanumeric characters enclosed with double quote characters, e.g., `"This is a string"`.

- 1-D array — a sequence of zero or more objects that are normally referenced with an index. An array is enclosed by parentheses and preceded by a '#' character, e.g., `#(1 2 3)`. Superficially, a 1-D array appears similar to a list, but it is not represented the same internally on the computer. It is not essential here to understand how and why arrays and lists differ, only that they are not always interchangeable.
- 2-D matrix — a matrix is written as a list of lists, preceded by "#2a" (without the quotes) to indicate that the data are in a matrix of rank 2. For example, the inertia matrix for the example in section 9.1 is written "`#2a((Ixx 0 Ixz) (0 Iyy 0) (Ixz 0 Izzr))`."
- Comments — anything following a semi-colon character is ignored by Lisp.
- F-strings — a string preceded by an exclamation mark is parsed to obtain an algebraic expression, e.g., `!"dot([n1], vel(b))"`. This is not a part of Lisp, but is an addition made by AUTOSIM. F-strings are described in more detail in the next subsection.

Lisp Symbols

The symbol is a basic element in Lisp. A symbol has a printed representation, which usually has the appearance of a word written in capital letters. Some examples are: `RESET`, `M2`, `T`, `NIL`, `ADD-BODY`, and `B*`. Symbol names may include numbers and many non-alphabetic characters, such as '*' and '-'. Names read as input are automatically converted to capital letters, and therefore most alphabet characters in symbol names are upper-case, regardless of how they appear as input.

A symbol may also have an associated value. In fact, the Lisp symbol is a data structure with several forms of associated data beyond a print name and a value. However, in AUTOSIM, symbols are used by the analyst mostly for their printed representation or for functions that the symbols represent. The value of a Lisp symbol is viewed by typing the name of the symbol. A new value is assigned using the `setf` macro, as described later.

Two reserved symbols widely used in Lisp are `T` and `NIL`. `NIL` means empty or false (depending on context). An empty list (e.g., "()") is represented as `NIL`. All conditional forms (IF-THEN, etc.) base the decision on whether a Boolian form is `NIL` or not `NIL`. The symbol `T` is commonly used to indicate unconditionally not `NIL` (i.e., True).

Lists and Lisp Forms

Most of the inputs from the analyst using AUTOSIM are *Lisp forms*, entered as lists. Each form begins and ends with a parenthesis, and includes at least one element. For example, the form used to reset the AUTOSIM environment is

```
(reset)
```

The above form is a list with one item, namely, the symbol `reset`. In general, items in a list are separated by one or more spaces, tabs, or *newline* characters. The following two lists are equivalent:

```
; Version 1 (note the use of ";" to insert comments)

(add-body b :name "new body" :joint-coordinates #(L 0 0)
:cm-coordinates #(R1 0 R2) :inertia-matrix  #(ixx iyy izz))

; Version 2

(add-body b
  :name                "new body"
  :joint-coordinates   #(L   0    0 )
  :cm-coordinates      #(R1  0    R2)
  :inertia-matrix      #(ixx iyy izz))
```

The second version, entered with multiple lines and spaces for readability, is interpreted exactly the same as the first version.

If a list is entered directly into the system, it is "evaluated" and the result of the evaluation is then printed. For example, if one types `(add 3 4)`, the following display would be seen.[1]

```
? (add 3 4)
7
?
```

The forms used in AUTOSIM are technically classified in Lisp as either *functions* or *macros*. They have three types of items in the list: (1) the name of the function or macro (e.g., `reset`, `add-body`, `add`), (2) zero or more *required* arguments, and (3) zero or more pairs of *optional* keyword arguments. The pairs of keyword arguments consist of a keyword and an argument. Keywords always begin with the colon character, as in the above example. In the example (`add-body`), there is one required argument, (the symbol

---

[1] Allegro CL, the Lisp package sold by Apple computer, shows user entries in boldface and the machine responses in plain type. The question mark is a prompt issued by Allegro to indicate it is ready for the next input. Examples in this dissertation follow the same convention.

b), and four optional keyword arguments. The keywords are `:name`, `:joint-coordinates`, `:cm-coordinates`, and `:inertia-matrix.` Keyword arguments are always optional, and can appear in any order *after* the mandatory arguments. If a keyword argument is not provided, then the argument associated with that keyword defaults to a value specified in the Lisp program. Numerous examples of keyword arguments appear in Chapter 9.

Assigning Values to Lisp Symbols with SETF

The language Common Lisp includes hundreds of predefined functions, macros, and "special forms." Only one of these forms appears in the examples in Chapter 9. This is the macro `setf`, used to modify Lisp data. With AUTOSIM, it is mainly used to assign values to Lisp symbols. In Fortran, assignment is performed with the '=' symbol. A symbolic equivalent of the Fortran statement

```
X = A + B
```

is performed with the Lisp form

```
(setf x !"a + b")
```

Later, if the expression (A + B) is required, the Lisp symbol X can be used if we specify that the *value* of x [the expression `(A + B)`] is of interest and not the *name* (the symbol `X`). (To specify this in an F-string, the name of the symbol is preceeded with the '#' character. See Figure 9.1.4 for an example.)

AUTOSIM maintains about a hundred global Lisp variables. Most of these must not be changed by the analyst. One however, is intended to be set by the analyst. It is the symbol `*multibody-system-name*`. This symbol can be assigned to a string that gives a descriptive name for the multibody system. That name is used to generate some of the documentation for the Fortran simulation code generated by AUTOSIM. Also, it appear on the screen when the simulation code is run by the end user.

Table A.2.1. Mathematical functions that can be used in F-strings.

F-String	*Lisp*	*Argument(s)*	*Description*
–	neg	*x*	$-x$
–	sub	*x, y*	$x - y$
*	mul	*x, y*	$x\,y$ (either *x* or *y* must be a scalar)
**	power	*x, y*	x^y (*x* is scalar, *y* is a number)
+	add	*x, y*	$x + y$
/	div	*x, y*	x / y (*y* is scalar)
#	N/A	*symbol*	use value of Lisp symbol named *symbol*
atan	make-atan	*x*	$y = \tan^{-1} x$ $(-\pi/2 \le y \le +\pi/2)$
atan2	make-atan	*s1, s2*	$y = \tan^{-1}(s1 / s2)$ $(-\pi \le y \le +\pi)$
cos	make-cos	*x*	$\cos x$ (*x* is scalar)
func	make-func	*fname,* {*arg*}*	arbitrary Fortran function:
nominal	nominal	*x*	expression when all variables are zero
sin	make-sin	*x*	$\sin x$ (*x* is scalar)
angle	angle	*v1, v2,* {*v3*}	angle between two vectors, sign determined by optional third vector (right-handed rule)
cross	cross	*v1, v2*	$\vec{\mathbf{v}}_1 \times \vec{\mathbf{v}}_2$
dir	dir	*v*	$\vec{\mathbf{v}} / \lvert\vec{\mathbf{v}}\rvert$
dot	dot	*v1, v2*	$\vec{\mathbf{v}}_1 \bullet \vec{\mathbf{v}}_2$
dplane	dot-plane	*v1, v2*	project vector onto plane
dxdt	dxdt	*x*	$\dot{x}$
mag	mag	*v*	$\lvert\vec{\mathbf{v}}\rvert$
partial	partial	*y, x*	$\partial y / \partial x$ (*x* is scalar)
pos	pos*	*p1,* {*p2*}	position vector connecting two points (default for *p2* is the fixed origin)
rot	rot*	*B*	absolute rotational velocity of body *B*
vel	vel*	*p1,* {*p2*}	difference between absolute velocities of 2 points (default for *p2* is the fixed origin)

NOTE: arguments enclosed with braces {} are optional. Those followed with a '*' are repeated zero or more times.

A.2 AUTOSIM Algebraic Expressions

In Chapter 5, a number of Lisp functions were defined to perform algebraic operations. Algebraic expressions containing these operations are printed in Fortran syntax, e.g., `3.0*A*B**2/SIN(Q(3))`. The Lisp functions are used to automate all operations that are performed in a formal procedure. In addition to predefined analyses, these functions are also needed by the analyst to describe forces, moments, output variables, and constraint equations. To simplify their use by the analyst, AUTOSIM includes a *parser* that reads a Fortran-style expression and converts it to a Lisp equivalent. The parser is invoked by putting the expression in a string, and preceding the string with an exclamation mark. Expressions entered in this way are called F-strings. Examples of F-strings appear frequently in the examples of Chapter 9.

Briefly, the parser reads a string in two steps. It first removes any spaces and linefeeds from the string and converts all of the text to upper-case. Then, through a sequence of "find and replace" operations, it replaces known functions with Lisp equivalents. Arithmetic operators (+, –, /, *, **) are also replaced. Table A.2.1 summarizes the functions recognized by the parser.

Normally, any symbols that appear in an F-string are assumed to represent parameters or variables in the multibody system. However, it is sometimes convenient to assign an expression to a Lisp symbol via the `setf` macro, and then include that expression in an F-string. To do this, the name of the Lisp symbol should be preceded with the character '#' (without the quotes). Examples of this are seen in Section 9.1.

A.3 AUTOSIM Functions and Macros

The basic procedure for analyzing a system is as follows:

1. Invoke the function `reset` to initialize AUTOSIM and clear any old data.
2. Describe the multibody system. Include all of the bodies, additional points of interest, constraint equations, forces, moments, and external subroutines.
3. Define all output variables of interest.
4. Invoke the function `dynamics` to derive efficient equations for computing state variables and output variables.

5. (this step is optional.) Check the descriptions of the input parameters identified by AUTOSIM. Set names and units in the paramater definitions (replacing names and units chosen automatically) if the existing descriptions are not clear enough.

6. Generate the Fortran simulation code by invoking the function `write-sim`.

The analysis functions are listed in Table A.3.1. The macros and functions used to describe the multibody system and the output variables are described below.

Table A.3.1. AUTOSIM functions for analyzing the multibody system.

Lisp Function	Action
`reset`	clear all AUTOSIM data to start new analysis
`dynamics`	analyze system to derive equations of motion
`write-sim`	write simulation code in the Fortran language

Describing the Multibody System

Table A.3.2 lists the macros used to build a description of the multibody system. The order in which these macros are entered as inputs by the analyst is not critical, other than the obvious restriction that it is not possible to reference an object until it has been entered. For example, if body B is entered with body A listed as the parent, it is necessary to add body A before adding body B. The macros in the table are described in more detail below.

`add-body` *symbol*

This macro creates a `body` object and sets most of its slots. The conventions for representing system topology and joint kinematics that were presented in Chapter 8 are put into use with this macro. In addition to creating the `body` object, this macro creates `uv`s as needed for the coordinate system of the new body, a number of `sym` objects for mass and inertia parameters, and three `point` objects representing (1) the origin of the new coordinate system, (2) the joint position (fixed in the coordinate system of the parent), and (3) the the center of mass. The macro has one required argument, *symbol*, which is a unique symbol used to identify the body, e.g., `B`. The keyword arguments are defined below:

Table A.3.2. AUTOSIM macros for describing a multibody system.

Lisp form	Arguments	
	Required	**Optional Keywords**
`add-body`	*symbol*	`:parent :coordinate-system :name :mass :inertia-matrix :joint-coordinates :cm-coordinates :translate :parent-rotation-axis :body-rotation-axes :reference-axis :small-angles :small-translations`
`add-constraint`	*expression*	`:variable`
`add-gravity`		`:direction :gees`
`add-line-force`	*symbol*	`:name :direction :magnitude :no-forcem :point1 :point2 :x :x0 :v`
`add-moment`	*symbol*	`:name :direction :magnitude :body1 :body2 :no-forcem`
`add-point`	*symbol*	`:name :body :coordinates :coordinate-system`
`add-strut`	*symbol*	`:magnitude :point1 :point2 :no-forcem :x :x0 :v :name`
`add-subroutine`	*where name* {*symbol*}*	
`add-variables`	*where type* {*symbol*}*	
`large`	{*symbol*}*	
`no-movement`	*point1 point2 direction*	`:confirm`
`set-defaults`	{*symbol number*}*	
`set-name`	{*symbol name*}*	
`set-units`	{*symbol units*}*	
`small`	{*symbol*}*	

NOTE: arguments enclosed with braces {}* are repeated zero or more times.

parent — the parent body. The default is `N`.

mass — an expression for the mass of the new body. If this argument is not provided, a symbol is created by appending the letter M to *symbol* . (e.g., `BM`)

name — a string that describes the body. (e.g., `"left-front wheel"`) If the argument is not provided, *symbol* is used as the name.

inertia-matrix — the inertia matrix of the body, with respect to the local coordinate system. Three forms of input are allowed: (1) a 3 × 3 matrix, containing all of the terms; (2) a three-element array, containing the moments of inertia (the products are set to zero); and (3) zero (all moments and products are set to zero). If this argument is not provided, a full 3 × 3 matrix is used. AUTOSIM generates symbols by appending the letter `I` and two digits to *symbol*, e.g., `BI11`, `BI12`, `BI22`, `BI13`, `BI23`, and `BI33`.

coordinate-system — the body whose coordinate system is used to specify point locations and movement directions. The default is the parent body for all coordinates except the center of mass. The default for the center of mass is the coordinate system of the new body.

joint-coordinates — a 3-element array containing the coordinates of the joint, using the specified coordinate system. In Figure 8.1.1, the vector defined by these coordinates is designated $\vec{\mathbf{r}}^{A_0B}$. The default is `#(0 0 0)`.

cm-coordinates — a 3-element array containing the coordinates of the center of mass, using the specified coordinate system. In Figure 8.1.1, the vector defined by these coordinates is designated $\vec{\mathbf{r}}^{B_0B^*}$. The default is `#(0 0 0)`.

translate — a list of consecutive translations allowed by the joint. Translations are allowed in any direction, with the direction(s) fixed in the parent body. This argument is normally a list. The length of the list provides the number of translational generalized coordinates for the body. Each item on the list must be either (1) a number specifying an axis parallel to the direction of translation, or (2) a 3-element array giving the coordinates of the translational direction. Directions are defined in the specified coordinate system. The joint shown in Figure 8.1.1 has a single translational degree of freedom in the direction $\vec{\mathbf{r}}_T^B$. Assuming this direction is not aligned with an axis in the parent, an array of 3 coordinates would be used to describe the direction.

When there is one translational generalized coordinate, it can be provided in lieu of a list with one element. The default is `NIL` (a joint with no translational degrees of freedom).

parent-rotation-axis — the direction of the first rotation (if the joint has one or three rotational degrees of freedom) in the specified coordinate system. In the figure, this direction is designated $\vec{\mathbf{r}}^{B}_{rot}$. The direction is described either by (1) a 3-element array containing its coordinates, or (2) the number of an axis parallel to the rotation. The default is that the *parent-rotation-axis* is the same as the first element of the *body-rotation-axes* list. If neither the *parent-rotation-axis* nor the *body-rotation-axes* arguments are provided, the joint has no rotational degrees of freedom.

body-rotation-axes — a list of consecutive rotation axes in the new body. The axes of the coordinate system of the new body are defined by this argument. The number of rotations is obtained from the length of this list (0, 1, or 3). Each element of the list must be an axis number (1, 2, or 3) indicating about which axis the rotation occurs. If the joint has one degree of freedom, a single number is valid as a value. (It need not be enclosed in parentheses.) If this argument is not provided and the *parent-rotation-axis* is also not provided, then the joint has no rotational degrees of freedom. If the *body-rotation-axes* list is not provided and the *parent-rotation-axis* is provided, then the *body-rotation-axes* list is set to a single rotation about an axis included in direction of *parent-rotation-axis*. (Unless the *parent-rotation-axis* is an axis number, the analyst should not depend on the default *body-rotation-axes* being set as intended.)

reference-axis — the coordinates (or axis number) of the reference axis, in the coordinate system of the parent. The reference axis determines the orientation of the new body relative to the parent in the nominal state. (The nominal state exists when all generalized coordinates are zero.) In Figure 8.1.1, this direction is designated $\vec{\mathbf{r}}^{B}_{ref}$. The default is determined by the right-handed convention from the *parent-rotation-axis*.

small-angles — a list whose length matches the number of rotational degrees of freedom, and which identifies the rotation angles as “small” or “not small.” If this argument is provided, it must have the same length as the list provided for the argument *body-rotation-axes*. Small angles are identified as `T`, large angles are identified as `NIL`. If the joint has one rotational degree of freedom, the value can

be provided without enclosing it in parentheses. The default is that all generalized coordinates and speeds introduced for rotation in this body are not small.

small-translations — a list whose length matches the number of translational degrees of freedom, and which identifies the translational displacements as "small" or "not small." If this argument is provided, it must have the same length as the list provided for the argument *translate*. Small displacements are identified as `T`, large displacements are identified as `NIL`. If the joint has one translational degree of freedom, the value can be provided without enclosing it in parentheses. The default is that all generalized coordinates and speeds introduced for translation of this body are not small.

`add-constraint` *zero-exp*

This function is used to apply a constraint equation. The required argument *zero-exp* is an expression that is identically zero when the constraint is satisfied. The optional argument *variable* is a state variable (generalized speed or generalized coordinate) that is eliminated by the constraint. The expression *zero-exp* is used to "solve-for" *variable*. If *variable* is not provided, then the units of *zero-exp* are used to determine if the variable to be eliminated should be a coordinate or a speed. The criteria for selecting the variable to eliminate was described in Chapter 8.

`add-gravity`

This macro applies a force to each body at its mass center, in the direction *direction*, with magnitude *m*•*gees* where *m* is the mass of the body. The default direction is [N3] and the default symbol for *gees* is `GEES`.

`add-line-force` *symbol*

This macro creates a `force` object that represents a force in the system that passes through a known point with a known line of action. The required argument, *symbol*, is a symbol used to identify the force. The optional arguments are defined below. Note that at least one of the optional arguments *point1* or *point2* must be included in order for the force to actually affect the system. Typically, most of the additional arguments are used.

name — string that describes the force, used by AUTOSIM when creating documentation and for labeling output variables, e.g., `"tire force"`. The default is *symbol*.

direction — expression for the direction in which the force acts. The default is `[N3]`.

magnitude — expression for the force magnitude. (E.g., `!"-K*(x - x0) - D*v"`). This expression must be scalar. Because springs, dampers, and controllers involve deflections, dummy variables are provided to simplify the specification of simple elements. (See descriptions for the keywords `x`, `x0`, and `v`.) The default is a constant that prints the same as *symbol*.

point1 — point upon which the force acts. The line of action for the force passes through this point to affect the body containing *point1*. The default is the fixed origin (i.e., the origin of the inertial reference).

point2 — second point. This point serves two functions: (1) the second body influenced by the force is the body containing *point2*, and (2) relative deflection and velocity are define between *point1* and a plane perpendicular to *direction* that contains *point2*. The line of action for the force does not necessarily pass through this point. The default is the fixed origin.

x — a dummy variable for the distance between *point1* and a plane perpendicular to *direction* that passes through *point2*. If the expression for *magnitude* contains this symbol, the symbol is replaced with an expression for the distance. The default symbol is `X`. This need only be changed if the analyst has provided an expression for *magnitude* that uses the symbol `X` to represent an existing parameter or external variable in the system.

x0 — a dummy variable for the nominal value of *x* when all generalized coordinates are zero. This expression is the "static" distance between *point1* and the plane containing *point2* that is perpendicular to *direction*. The default symbol is `X0`. This need only be changed if the analyst has provided an expression for *magnitude* that uses the symbol `X0` to represent an existing parameter or external variable in the system.

v — a dummy variable for the speed between *point1* and *point2*, in the direction *direction*. If the expression for *magnitude* contains this symbol, the symbol is replaced with an expression for the speed. The default symbol is `V`. This need only be changed if the analyst has provided an expression for *magnitude* that uses the symbol `V` to represent an existing parameter or external variable in the system.

no-forcem — AUTOSIM normally introduces a new symbol for each force magnitude (an element of the Fortran array FORCEM) to make the simulation code easier to read. To disable this behavior, set *no-forcem* to any value except `NIL` (e.g., `T`). The default is `NIL`.

`add-moment` *symbol*

This macro creates a `moment` object that represents the moment of a couple between two bodies. The required argument, *symbol*, is a symbol used to identify the moment. The optional arguments are defined below. Note that at least one of the optional arguments *body1* or *body2* must be included in order for the moment to actually affect the system.

name — string that describes the moment. The default is *symbol*.

direction — expression for the direction or the moment. The default is `[N3]`.

magnitude — expression for the moment magnitude. This expression must be scalar. The default is a constant that prints the same as *symbol*.

body1 — symbol for body upon which the moment acts. The default is `N`.

body2 — symbol for body from which the moment acts. The default is `N`.

no-forcem — AUTOSIM normally introduces a new symbol for each force magnitude (an element of the Fortran array FORCEM) to make the simulation code easier to read. To disable this behavior, set *no-forcem* `T`. The default is `NIL`.

`add-point` *symbol*

This macro defines a point on a body for later reference in describing forces, output variables, or constraint equations. The required argument *symbol* is used to reference the point later. The optional arguments are the following:

name — a string that describes the point. The default is *symbol*.

body — the body/coordinate system in which the point is located. The default is `N`.

coordinates—a 3-element array containing the coordinates of the point in the specified coordinate system. The default is `#(0 0 0)`.

coordinate-system — a coordinate system to use for specifying the *coordinates* of the point. If *coordinate-system* is not the same as *body*, the coordinates are converted by assuming the system is in its nominal state (all generalized coordinates are zero). The default is *body*.

`add-strut` *symbol*

This macro defines a force that connects two known points. The direction of the force is derived by AUTOSIM. This macro is similar to `add-line-force`, and most of the arguments have the same meanings. The difference is that a force defined with `add-line-force` has a known direction and a single point through which the line of action passes, whereas a force defined with `add-strut` has an unknown direction that passes through two known points. Thus, `add-strut` does not include an argument for the direction. Also, the argument *point2* is more simply defined as the second point upon which the force acts. The dummy variables *x*, *x0*, and *v* involve displacement and velocity along the line connecting *point1* and *point2*.

`add-subroutine` *where name* {*variable*}*

This macro is used to specify that an external subroutine should be included in the code generated by AUTOSIM. *name* is the name of the subroutine, and the arguments are listed as zero or more *variable* arguments. The variables can be any scalar expressions. An example use of this macro appear in section 9.4.

The argument *where* specifies where in the simulation code the subroutine is needed. The valid symbols that can be provided are the following:

`difeqn` — the subroutine call should be made in the DIFEQN subroutine, to contribute to the equations of motion. For example, an external tire model might be implemented as a subroutine that has several input variables and several output forces and moments that appear in the equations of motion.

`echo` — the subroutine call should be made in the ECHO subroutine, to print data into an echo file. For example, if an external tire model is used, a subroutine might be included to print all of the tire parameters that are hidden from AUTOSIM.

`input` — the subroutine call should be made in the INPUT subroutine, to parse lines of input for keywords related to external subroutines.

`init` — the subroutine call should be made when initializations are performed.

`output` — the subroutine call should be made in the OUTPUT subroutine, to compute values needed for one or more output variables.

`update` — the subroutine call should be made once per time step, so that local variables in the subroutine can be "updated." This is necessary with many models that involve hysteresis.

`add-variables` *where type* {*variable*}*

This macro is similar to `add-subroutine`, and exists mainly to define variables that appear in external subroutines (added with `add-subroutine`) that receive values from the subroutines. The argument *where* has exactly the same meaning and accepted values as described above for `add-subroutine`. The argument *type* is a Fortran variable type, such as `REAL`, `INTEGER`, `CHARACTER*20`, `REAL*8`, etc. The rest of the arguments are the variables being added.

`large` {*symbol*}*

This macro is used to declare that one or more *symbols* are "large." This is commonly applied to large stiffness values that are multiplied with "small" deflections to obtain forces or moments that are not small. For example, see section 9.4. The macro assigns the `small-order` slot of each argument a value of –1.

`no-movement` *point1 point2 direction*

This macro adds two constraint equations, one for speed and one for position, that declare that there is no movement between *point1* and *point2* in the direction *direction.* The macro `no-movement` works by invoking the `add-constraint` macro twice. The first time, it takes the difference in velocity of the two points and dots the result with *direction* to obtain a scalar constraint equation. The second time, it takes the difference in position between the two points and again dots the result with *direction* to obtain a scalar expression. It verifies that it can select two variables to eliminate (one speed and one coordinate) before invoking `add-constraint`, and will do nothing if it cannot find both variables. The keyword `:confirm` can be set to `T` to allow the analyst a chance to cancel if he or she does not approve the choice of variables to eliminate. (If the analyst does not approve of the variable selected by the macro, no action is taken.) The default is `NIL`. That is, the macro does not offer the analyst a chance to cancel.

`set-defaults` {*symbol number*}*

All parameters in the simulation code generated by AUTOSIM have default values. If the analyst provides no information, all default values are 1.0. This macro is used to assign different values. The numerical values assigned here are used in the simulation code only if

the end user does not provide values as inputs. That is, all parameters can be modified by the user, whether or not this macro was used by the analyst.

`set-name` {*symbol name*}*

All parameters and variables in the simulation code generated by AUTOSIM have names. However, the names generated automatically by AUTOSIM may not be as meaningful to the end user as names that the analyst might have in mind. This macro is used to override names of parameters and variables that appear in documentation and output files of the simulation code.

`set-units` {*symbol units*}*

All parameters and variables in the simulation code generated by AUTOSIM have units. However, there is not always enough information to deduce the units of some parameters and external variables. This macro is used to override units of parameters and variables that appear in documentation and output files of the simulation code.

`small` {*symbol*}*

This macro is used to declare that one or more *symbols* are "small." This is commonly applied to speeds that are small, but which apply to coordinates that are not small. For example, see section 9.1. The macro assigns the `small-order` slot of each argument a value of +1.

Specifying Output Variables

Although most of the material in this dissertation involves the derivation of equations of motion, the actual purpose of a simulation code is to generate time histories of variables of interest. Thus, it is essential to specify exactly which variables are of interest and should be written as output by the simulation code. The macro `add-out` is used to specify virtually any output variable that might be of interest. Additionally, a few functions have been prepared to automatically specify that the simulation code generated by AUTOSIM include groups of "standard" variables as output variables. These functions are listed in Table A.3.

Table A.3.3. AUTOSIM functions for specifying outputs.

Lisp Function			Action
`add-accelerations-to-output`			Add all derivatives of generalized speeds to the list of output variables.
`add-coordinates-to-output`			Add all generalized coordinates to the list of output variables.
`add-forces-to-output`			Add all force magnitudes to the list of output variables.
`add-moments-to-output`			Add all moment magnitudes to the list of output variables.
`add-speeds-to-output`			Add all generalized speeds to the list of output variables.
Function	**Required**	**Optional**	
`add-out`	*expression* *short-name*	*long-name* *gen-name* *body units*	Add one variable to list of outputs. Specify labels with keyword arguments.

`add-out` *expression short-name*

This macro defines a variable that will be computed in the simulation code and written into an output file. The required argument *expression* is a scalar expression. The second required argument *short-name* is a string with up to 8-characters that describes the variable. The optional keyword arguments provide additional labeling information. The program generated by AUTOSIM will put this information into the header of the output file to facilitate automated post-processing of file generated by the simulation code. The keyword arguments are defined as follows:

long-name — a string with up to 32 characters that provides a more detailed name for the variable. The default is *short-name*.

gen-name — a string with up to 32 characters that provides generic name for the variable. The default is determined by the units of the variable.

body — the body most closely associated with the variable. The default is N.

units — the units of the variable. The default is the expression for units that is obtained using the AUTOSIM function `get-units`.

APPENDIX B — PASSENGER CAR HANDLING MODEL

This appendix contains the complete source code for the passenger car handling model described in Section 9.1. This version is based on statements that some of the variables are small.

```
C  Passenger car handling model simulation program.
C  Version created December 13, 1989 by AUTOSIM
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
C  This program simulates the passenger car handling model by
C  numerically integrating the 7 ordinary differential equations that
C  describe the kinematics and dynamics of the system.  The passenger
C  car handling model is composed of 2 bodies and has 3 degrees of
C  freedom.
C
C  Each derivative evaluation requires 34 multiply/divides, 24
C  add/subtracts, and 2 function/subroutine calls.
C
C  Bodies:
C  =======
C  Non-rolling body (NRB); parent=N; 3 coords: Q(1) Q(2) Q(3)
C  Rolling body (RB); parent=NRB; 1 coord: Q(4)
C
C  Generalized Coordinates:
C  ========================
C    Q(1): Translation of NRB0 relative to the fixed origin along [n1].
C          (in)
C    Q(2): Translation of NRB0 relative to the fixed origin along [n2].
C          (in)
C    Q(3): Rotation of the non-rolling body relative to the inertial
C          reference about axis #3. (deg)
C    Q(4): Rotation of the rolling body relative to the non-rolling
C          body about axis #1. (deg)
C
C  Independent Speeds:
C  ===================
C    U(1): Abs. trans. speed of NRB* along axis 2. (in/s)
C    U(2): Abs. rot. of NRB, axis 3. (deg/s)
C    U(3): Rot. of RB relative to NRB, axis 1. (deg/s)
C
C  Nonholonomic Constraints:
C  =========================
C  Abs. trans. speed of NRB* along axis 1.: SPEED
C
C  Active Forces:
C  ==============
C  FORCEM(1): (negative) Side force, front axle
C  FORCEM(2): (negative) Side force, rear axle
```

```
C
C  Active Moments:
C  ===============
C  FORCEM(3): (negative) Aligning moment, front axle
C  FORCEM(4): (negative) Aligning moment, rear axle
C  FORCEM(5): (negative) roll moment from suspension
C
C  Program Sections:
C  =================
C  MAIN -- Control flow of program and perform numerical integration
C
C  BLOCK DATA -- initialize variables in COMMON blocks
C  DIFEQN (T, Q, QP, U, UP) -- compute QP and UP given T, Q, and U
C  ECHO (IFILE, Q, U) -- create output file with echo of input
C            parameters
C  INPUT (Q, U) -- read parameters and initial conditions
C  Function LENSTR (STRING) -- count characters in left-justified
C            string
C  Function NORMA(A) -- Normalize angle
C  Function OPNFIL(PROMPT, STAT, IUNIT) -- let user open file
C  OPNOUT(IFILE) -- create output file and write header
C  OUTPUT(IFILE, T, Q, QP, U, UP) -- write variables at time T
C  PRECMP -- pre-compute constants used in simulation
C  TIMDAT(TIMEDT) -- get time and date from computer
C
      IMPLICIT NONE
      CHARACTER*80 INFILE, TITLE
      REAL         NORMA, PARS, STEP, STEP2, STOPT, T, Y, YM, YP
      INTEGER      I, IECHO, IFILE, ILOOP1, ILOOP2, IPRNT2, ISEC1,
     &             ISEC2, NCOORD, NLOOP, NPARS, NSPEED, NTOT
C
      PARAMETER    (NCOORD = 4, NSPEED = 3, IFILE = 1, NTOT = 7)
      DIMENSION    Y(NTOT), YM(NTOT), YP(NTOT)
      PARAMETER    (NPARS = 24)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE (PARS(22), STEP), (PARS(23), STOPT)
C
      WRITE(*, '(5A)')
     & ' Passenger car handling model simulation program.'
      WRITE(*, '(5A)')
     & ' Version created December 13, 1989 by AUTOSIM'
      WRITE(*, '(5A)')
     & '  '
C
C  Read input data
C
      CALL INPUT(Y, Y(NCOORD + 1))
      IPRNT2 = PARS(10)
C
C   Compute constants in common block /PRCMP/ before starting.
C
      CALL PRECMP
C
C   Option to echo data to output file
```

```
C
      CALL ECHO(IFILE, Y, Y(NCOORD + 1))
C
C   Set up output file with simulated time histories
C
      CALL OPNOUT (IFILE)
      CALL TIME (ISEC1)
C
C Start by evaluating derivatives and printing variables at t=0
C
      T = 0.
      CALL DIFEQN(T, Y, YP, Y(NCOORD + 1), YP(NCOORD + 1))
      CALL OUTPUT(IFILE, T, Y, YP, Y(NCOORD + 1), YP(NCOORD + 1))
C
C Integration loop. Continue until printout time reaches final time.
C Use two evaluations of the derivatives to integrate over the step.
C
      NLOOP = STOPT / STEP / IPRNT2 + 1
      STEP2 = STEP / 2.
      DO 60 ILOOP1 = 1, NLOOP
        DO 50 ILOOP2 = 1, IPRNT2
          DO 10 I = 1, NTOT
            YM(I) = Y(I) + STEP2 * YP(I)
   10     CONTINUE
          CALL DIFEQN (T + STEP2, YM, YP, YM(NCOORD + 1),
     &                 YP(NCOORD + 1))
C
          DO 20 I = 1, NTOT
            Y(I) = Y(I) + STEP * YP(I)
   20     CONTINUE
C
          T = T + STEP
          CALL DIFEQN (T, Y, YP,  Y(NCOORD + 1), YP(NCOORD + 1))
   50   CONTINUE
        CALL OUTPUT (IFILE, T, Y, YP, Y(NCOORD + 1), YP(NCOORD + 1))
        IF (T .GE. STOPT) GO TO 70
   60 CONTINUE
   70 CONTINUE
C
      CALL TIME (ISEC2)
C
C End of integration loop.  Print final status of run
C
      WRITE (*, *) ' Termination at time =', T, ' sec.'
      WRITE (*,*) ' Computation efficiency: ', (ISEC2 - ISEC1) / T,
     &            ' sec/sim. sec'
      WRITE (*,*) ' '

      CLOSE(IFILE)
      PAUSE ' Done'
      END
```

```
C=======================================================================
      BLOCK DATA
C=======================================================================
      CHARACTER*80 INFILE, TITLE
      REAL         PARS
      INTEGER      NPARS
C
      PARAMETER    (NPARS = 24)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      DATA         PARS /-444.0, -428.0, 1080.0, 1000.0, 0.82, 63.4,
     &             78.0, 212.0, 15.48, 2.0, 5580.0, 0.0, 37080.0,
     &             6211.0, -0.016, 125.5, 1.0, 704.0, 3831.0, 968.0,
     &             1.0, 0.025, 2.0, 5.1/
      DATA         INFILE /' '/
      DATA         TITLE  /'Default parameter values'/
      END
C=======================================================================
      SUBROUTINE DIFEQN(T, Q, QP, U, UP)
C=======================================================================
C  This subroutine defines the equations of motion for the Passenger
C  car handling model, which includes 3 degrees of freedom.
C
C  --> T  real  time
C  --> Q  real  array of 4 generalized coordinates
C  <-- QP real  array of derivitives of Q
C  --> U  real  array of 3 generalized speeds
C  <-- UP real  array of derivatives of U
C
C  Each derivative evaluation requires 34 multiply/divides, 24
C  add/subtracts, and 2 function/subroutine calls.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      CHARACTER*80 INFILE, TITLE
      REAL         C, CA1, CA2, CAM1, CAM2, CCOEF1, CE, CG1, CROLL,
     &             DEGREES, FORCEM, GEES, H, IPRINT, IXX, IXZ, IZZR,
     &             KROLL, KRS2, L, NRBI33, NRBM, PARS, PC, Q, QP, RBM,
     &             S, SPEED, STEER, STEP, STOPT, T, THETAR, U, UP, Z
      INTEGER      NCOORD, NPARS, NSPEED
C
      PARAMETER    (NCOORD = 4, NSPEED = 3)
      DIMENSION    Q(NCOORD), QP(NCOORD), U(NSPEED), UP(NSPEED)
      DIMENSION    C(4), FORCEM(5), S(4), Z(30)
      COMMON       /DYVARS/ C, FORCEM, S, Z
      SAVE         /DYVARS/
C
      PARAMETER    (NPARS = 24)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE  (PARS(1), CA1), (PARS(2), CA2), (PARS(3), CAM1),
```

```
     &             (PARS(4), CAM2), (PARS(5), CCOEF1), (PARS(6), CE),
     &             (PARS(7), CG1), (PARS(8), CROLL), (PARS(9), H),
     &             (PARS(10), IPRINT), (PARS(11), IXX), (PARS(12),
     &             IXZ), (PARS(13), IZZR), (PARS(14), KROLL),
     &             (PARS(15), KRS2), (PARS(16), L), (PARS(17), NRBI33),
     &             (PARS(18), NRBM), (PARS(19), RBM), (PARS(20),
     &             SPEED), (PARS(21), STEER), (PARS(22), STEP),
     &             (PARS(23), STOPT), (PARS(24), THETAR)
      DIMENSION    PC(43)
      COMMON       /PRCMP/ PC
      SAVE         /PRCMP/
C
      PARAMETER    (GEES = 386.2, DEGREES = 57.29577951308232)
      S(3) = SIN(Q(3))
      C(3) = COS(Q(3))
C
C
C  Kinematical equations
C
      QP(1) = (SPEED*C(3) -U(1)*S(3))
      QP(2) = (U(1)*C(3) + SPEED*S(3))
      QP(3) = U(2)
      QP(4) = U(3)
C
C  define expression for Side force, front axle
C
      Z(1) = (STEER -PC(43)*U(2) -PC(2)*U(1))
      FORCEM(1) = (-PC(1)*Q(4) + CA1*Z(1))
C
C  define expression for Side force, rear axle
C
      Z(2) = (KRS2*Q(4) -PC(2)*U(1))
      FORCEM(2) = CA2*Z(2)
C
C  define expression for Aligning moment, front axle
C
      FORCEM(3) = CAM1*Z(1)
C
C  define expression for Aligning moment, rear axle
C
      FORCEM(4) = CAM2*Z(2)
C
C  define expression for roll moment from suspension
C
      FORCEM(5) = (KROLL*Q(4) + PC(4)*U(3))
C
C  Dynamical equations
C
      Z(3) = PC(3)*Q(4)
      Z(4) = SPEED*U(2)
      Z(5) = (PC(8)*Q(4) -Z(4))
      Z(6) = (PC(32)*Q(4) + NRBM*Z(4) -RBM*Z(5) + FORCEM(1) +
     &        FORCEM(2))
      Z(7) = (PC(36)*Z(3) -PC(10)*Z(5) + L*FORCEM(1) + FORCEM(3) +
     &        FORCEM(4))
      Z(8) = PC(16)*Z(6)
      Z(9) = (PC(37)*Z(5) + PC(38)*Z(6) + PC(39)*(Z(7) -Z(8))
```

```
     &        -PC(28)*FORCEM(5))
      Z(10) = (PC(29)*(Z(7) -Z(8)) + PC(40)*Z(9))
      UP(3) = Z(9)
      UP(2) = -Z(10)
      UP(1) = -(PC(30)*Z(6) + PC(41)*Z(9) -PC(42)*Z(10))
      RETURN
      END
C=======================================================================
      SUBROUTINE ECHO(IFILE, Q, U)
C=======================================================================
C  This subroutine prompts the user for the name of an optional echo
C  file for the passenger car handling model.  If a file is selected,
C  all of the parameter values and initial conditions are written to
C  confirm that the intended values were used in the simulation.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      CHARACTER*24 TIMEDT
      CHARACTER*80 INFILE, OPNFIL, TITLE
      REAL         CA1, CA2, CAM1, CAM2, CCOEF1, CE, CG1, CROLL,
     &             DEGREES, GEES, H, IPRINT, IXX, IXZ, IZZR, KROLL,
     &             KRS2, L, NRBI33, NRBM, PARS, Q, RBM, SPEED, STEER,
     &             STEP, STOPT, T, THETAR, U
      INTEGER      IFILE, NCOORD, NPARS, NSPEED
C
      PARAMETER    (NPARS = 24)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE  (PARS(1), CA1), (PARS(2), CA2), (PARS(3), CAM1),
     &             (PARS(4), CAM2), (PARS(5), CCOEF1), (PARS(6), CE),
     &             (PARS(7), CG1), (PARS(8), CROLL), (PARS(9), H),
     &             (PARS(10), IPRINT), (PARS(11), IXX), (PARS(12),
     &             IXZ), (PARS(13), IZZR), (PARS(14), KROLL),
     &             (PARS(15), KRS2), (PARS(16), L), (PARS(17), NRBI33),
     &             (PARS(18), NRBM), (PARS(19), RBM), (PARS(20),
     &             SPEED), (PARS(21), STEER), (PARS(22), STEP),
     &             (PARS(23), STOPT), (PARS(24), THETAR)
      PARAMETER    (NCOORD = 4, NSPEED = 3)
      DIMENSION    Q(NCOORD), U(NSPEED)
      PARAMETER    (GEES = 386.2, DEGREES = 57.29577951308232)
C
      IF (OPNFIL('Name of (optional) file to echo the input data',
     &           'OPTOUT', IFILE) .EQ. ' ')  RETURN
C
      CALL TIMDAT(TIMEDT)
C
      WRITE(IFILE, '(A)') 'PARSFILE'
      WRITE(IFILE, '(5A)')
     & 'Echo file created by:'
      WRITE(IFILE, '(5A)')
     & 'Passenger car handling model simulation program.'
      WRITE(IFILE, '(5A)')
     & 'Version created December 13, 1989 by AUTOSIM'
```

```
 WRITE(IFILE, '(5A)')
&' '
 WRITE(IFILE, '(A,T8,A)') 'TITLE', TITLE
 WRITE(IFILE, '(/A,A)') '* Input File: ', INFILE
 WRITE(IFILE, '(A, A)') '* Run was made ', TIMEDT
              WRITE(IFILE, '(/A/)') '* PARAMETER VALUES'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'CA1', CA1/DEGREES,
& 'front cornering stiffness (lb/deg)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'CA2', CA2/DEGREES,
& 'rear cornering stiffness (lb/deg)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'CAM1', CAM1/DEGREES,
& 'front aligning moment coefficient (in-lb/deg)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'CAM2', CAM2/DEGREES,
& 'rear aligning moment coefficient (in-lb/deg)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'CCOEF1', CCOEF1,
& 'prop. of body roll resulting in front wheel camber (-)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'CE', CE,
& 'distance from rear axle to sprung mass c.g. (in)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'CG1', CG1/DEGREES,
& 'front camber stiffness (lb/deg)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'CROLL', CROLL/DEGREES,
& 'torsional damping rate for the vehicle body in roll'
&,' (in-lb-s/d)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'H', H,
& 'height of sprung mass c.g. above roll axis (in)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'IPRINT', IPRINT,
& 'number of time steps between output printing (counts)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'IXX', IXX,
& 'moment of inertia of RB (in-lb-s2)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'IXZ', IXZ,
& 'product of inertia of RB (in-lb-s2)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'IZZR', IZZR,
& 'moment of inertia of RB (in-lb-s2)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'KROLL', KROLL/DEGREES,
& 'torsional spring rate for the vehicle body in roll'
&,' (in-lb/deg)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'KRS2', KRS2,
& 'roll-steer coefficient for rear axle (-)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'L', L,
& 'wheelbase (in)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'NRBI33', NRBI33,
& 'moment of inertia of NRB (in-lb-s2)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'NRBM', GEES*NRBM,
& 'mass of NRB (lbm)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'RBM', RBM*GEES,
& 'mass of RB (lbm)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'SPEED', SPEED,
& 'forward speed (in/s)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'STEER', STEER*DEGREES,
& 'Steer angle at road (deg)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'STEP', STEP,
& 'simulation time step (sec)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'STOPT', STOPT,
& 'simulation stop time (sec)'
 WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'THETAR', THETAR*DEGREES,
& 'inclination angle of roll axis (deg)'
 WRITE(IFILE, '(/A/)') '* INITIAL CONDITIONS'
```

```
      WRITE(IFILE, '(A, T8, G13.6, T24, 5A)') 'Q(1)', Q(1),
     & 'Translation of NRB0 relative to the fixed origin along [n1].'
     &,' (in)'
      WRITE(IFILE, '(A, T8, G13.6, T24, 5A)') 'Q(2)', Q(2),
     & 'Translation of NRB0 relative to the fixed origin along [n2].'
     &,' (in)'
      WRITE(IFILE, '(A, T8, G13.6, T24, 5A)') 'Q(3)', DEGREES*Q(3),
     & 'Rotation of the non-rolling body relative to the inertial'
     &,' reference about axis #3. (deg)'
      WRITE(IFILE, '(A, T8, G13.6, T24, 5A)') 'Q(4)', DEGREES*Q(4),
     & 'Rotation of the rolling body relative to the non-rolling body'
     &,' about axis #1. (deg)'
      WRITE(IFILE, '(A, T8, G13.6, T24, 5A)') 'U(1)', U(1),
     & 'Abs. trans. speed of NRB* along axis 2. (in/s)'
      WRITE(IFILE, '(A, T8, G13.6, T24, 5A)') 'U(2)', DEGREES*U(2),
     & 'Abs. rot. of NRB, axis 3. (deg/s)'
      WRITE(IFILE, '(A, T8, G13.6, T24, 5A)') 'U(3)', DEGREES*U(3),
     & 'Rot. of RB relative to NRB, axis 1. (deg/s)'
      WRITE(IFILE, '(/A)') 'END'
      CLOSE(IFILE)
      RETURN
      END
C=======================================================================
      SUBROUTINE INPUT(Q, U)
C=======================================================================
C  This subroutine prompts the user for the name of an optional
C  parameter file for the Passenger car handling model.  If a file is
C  selected, parameter values are read to override the default values.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      LOGICAL      ISIT
      CHARACTER*80 BUFFER, ECHFIL, INFILE, OPNFIL, QUEUE, TITLE
      CHARACTER*8  CHAR8, NAMES, QC, UC
      REAL         CA1, CA2, CAM1, CAM2, CCOEF1, CE, CG1, CROLL,
     &             DEGREES, GEES, H, IPRINT, IXX, IXZ, IZZR, KROLL,
     &             KRS2, L, NRBI33, NRBM, PARS, PSCALE, Q, QINIT,
     &             QSCALE, RBM, SPEED, STEER, STEP, STOPT, T, THETAR,
     &             U, UINIT, USCALE
      INTEGER      IFILE, ILOOP, IQUEUE, LENSTR, LSTRNG, MAXQ, NCOORD,
     &             NPARS, NQUEUE, NSPEED
C
      PARAMETER    (NPARS = 24)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE  (PARS(1), CA1), (PARS(2), CA2), (PARS(3), CAM1),
     &             (PARS(4), CAM2), (PARS(5), CCOEF1), (PARS(6), CE),
     &             (PARS(7), CG1), (PARS(8), CROLL), (PARS(9), H),
     &             (PARS(10), IPRINT), (PARS(11), IXX), (PARS(12),
     &             IXZ), (PARS(13), IZZR), (PARS(14), KROLL),
     &             (PARS(15), KRS2), (PARS(16), L), (PARS(17), NRBI33),
     &             (PARS(18), NRBM), (PARS(19), RBM), (PARS(20),
     &             SPEED), (PARS(21), STEER), (PARS(22), STEP),
```

```
     &             (PARS(23), STOPT), (PARS(24), THETAR)
      PARAMETER    (GEES = 386.2, DEGREES = 57.29577951308232)
      PARAMETER    (NCOORD = 4, NSPEED = 3, MAXQ = 20, IFILE = 1)
      DIMENSION    NAMES(NPARS), Q(NCOORD), QC(NCOORD), QINIT(NCOORD),
     &             QUEUE(MAXQ), QSCALE(NCOORD), U(NSPEED), UC(NSPEED),
     &             UINIT(NSPEED), USCALE(NSPEED)
      DATA         QINIT /NCOORD*0./, UINIT /NSPEED*0./
      DATA         QC /'Q(1)', 'Q(2)', 'Q(3)', 'Q(4)'/
      DATA         UC /'U(1)', 'U(2)', 'U(3)'/
      DATA         QSCALE /1, 1, DEGREES, DEGREES/
      DATA         USCALE /1, DEGREES, DEGREES/
      DATA         NAMES /'CA1', 'CA2', 'CAM1', 'CAM2', 'CCOEF1', 'CE',
     &             'CG1', 'CROLL', 'H', 'IPRINT', 'IXX', 'IXZ', 'IZZR',
     &             'KROLL', 'KRS2', 'L', 'NRBI33', 'NRBM', 'RBM',
     &             'SPEED', 'STEER', 'STEP', 'STOPT', 'THETAR'/
      NQUEUE = 0
      IQUEUE = 0
C
C  Open file with parameter values and initial conditions
C
    5 INFILE = OPNFIL ('Name of (optional) file with parameter values',
     &         'OPTIN', IFILE)
    6 IF (INFILE .NE. '') THEN
        READ(IFILE, '(A)') CHAR8
C
        IF (CHAR8 .EQ. 'END') GO TO 100
        IF (CHAR8 .NE. 'PARSFILE') THEN
          CLOSE(IFILE)
          WRITE (*, '(A)') ' Error--File did not begin with "PARSFILE"'
          IF (IQUEUE .EQ. 0) THEN
            GO TO 5
          ELSE
            GO TO 100
          END IF
        END IF
C
C  Read line from file.  CHAR8 is the keyword, checked for:
C   o  TITLE keyword,
C   o  parameter keyword (from NAMES array),
C   o  initial value of generalized coordinate (keyword from QC array),
C   o  initial value of generalized speed (keyword from UC array),
C   o  (possibly) keyword for other input subroutine, or
C   o  END keyword.
C  All other lines are ignored.  Also, all lines are ignored after END
C  is found, and any line  with a '*' in column 1 is ignored.
C
   10   READ(IFILE, '(A8,A80)', END=100, ERR=100) CHAR8, BUFFER
        IF (CHAR8 .EQ. 'TITLE') THEN
          TITLE = BUFFER
          GO TO 10
        ELSE IF (CHAR8 .EQ. 'END') THEN
          GO TO 100
        ELSE IF (CHAR8(1:1) .EQ. '*') THEN
          GO TO 10
        ELSE IF (CHAR8 .EQ. 'PARSFILE') THEN
          INQUIRE (FILE=BUFFER, EXIST=ISIT)
          IF (ISIT) THEN
```

```
            NQUEUE = NQUEUE + 1
            QUEUE (NQUEUE) = BUFFER
          ELSE
            LSTRNG = LENSTR (BUFFER)
            WRITE (*,'(A,A,A)') 'Error--PARSFILE "', BUFFER(:LSTRNG),
     &                          '"not found (skipped).'
          END IF
          GO TO 10
        END IF
C
C  Check for names of parameters
C
        DO 20 ILOOP = 1, NPARS
          IF (CHAR8 .EQ. NAMES(ILOOP)) THEN
            READ(BUFFER, '(G13.0)') PARS(ILOOP)
            GO TO 10
          END IF
   20   CONTINUE
C
C  Check for names of generalized coordinates (initial conditions)
C
        DO 30 ILOOP = 1, NCOORD
          IF (CHAR8 .EQ. QC(ILOOP)) THEN
            READ(BUFFER, '(G13.0)') QINIT(ILOOP)
            GO TO 10
          END IF
   30   CONTINUE
C
C  Check for names of generalized speeds (initial conditions)
C
        DO 40 ILOOP = 1, NSPEED
          IF (CHAR8 .EQ. UC(ILOOP)) THEN
            READ(BUFFER, '(G13.0)') UINIT(ILOOP)
            GO TO 10
          END IF
   40   CONTINUE
C
        GO TO 10
      END IF
C
C  Close this file and process other PARS files that were referenced.
C
  100 CLOSE (IFILE)
      IF (IQUEUE .LT. NQUEUE) THEN
        IQUEUE = IQUEUE + 1
        INFILE = QUEUE (IQUEUE)
        OPEN (IFILE, STATUS='OLD', FILE=INFILE)
        WRITE (*, '(A,A)') ' Reading from PARSFILE ', INFILE
        GO TO 6
      END IF
C
C  Set initial conditions
C
      DO 110 ILOOP = 1, NCOORD
  110 Q(ILOOP) = QINIT(ILOOP) / QSCALE(ILOOP)
C
      DO 120 ILOOP = 1, NSPEED
```

```
  120 U(ILOOP) = UINIT(ILOOP) / USCALE(ILOOP)
C
C  Convert units as needed.
C
      CA1 = CA1*DEGREES
      CA2 = CA2*DEGREES
      CAM1 = CAM1*DEGREES
      CAM2 = CAM2*DEGREES
      CG1 = CG1*DEGREES
      CROLL = CROLL*DEGREES
      KROLL = KROLL*DEGREES
      NRBM = NRBM/GEES
      RBM = RBM/GEES
      STEER = STEER/DEGREES
      THETAR = THETAR/DEGREES
C
      RETURN
      END
C=======================================================================
      FUNCTION LENSTR (STRING)
C=======================================================================
C  count characters in left-justified string.   M. Sayers, 8-9-87
C
      CHARACTER*(*) STRING
      N = LEN (STRING)
      DO 10 L = N, 1, -1
        IF (STRING(L:L) .NE. ' ') THEN
          LENSTR = L
          RETURN
        END IF
   10 CONTINUE
      LENSTR = 1
      RETURN
      END
C=======================================================================
      FUNCTION NORMA(A)
C=======================================================================
C  normalize angle
C
      REAL A, NORMA, PI
      PARAMETER (PI=3.141592653589793)
      IF (A .GE. PI) THEN
        NORMA = A - PI
      ELSE IF (A .LE. -PI) THEN
        NORMA = A + PI
      ELSE
        NORMA = A
      END IF
      RETURN
      END
C=======================================================================
      FUNCTION OPNFIL (PROMPT, STAT, IFILE)
C=======================================================================
C  This function tries to get a file name from the user and open the
C  file.
C
C  --> PROMPT string   Message to prompt user
```

```
C  --> STAT   string   Status of file ("NEW"    = mandatory output,
C                                     "OLD"    = mandatory input,
C                                     "OPTIN"  = optional input,
C                                     "OPTOUT" = optional output)
C  --> IFILE  integer  Fortran I/O unit for file
C  <-- OPNFIL string   name of file opened or " " if no file selected
C
C M. Sayers January 30, 1989
C
      LOGICAL       ISIT
      CHARACTER*(*) PROMPT, STAT, OPNFIL
      CHARACTER*3   STAT2
      INTEGER       IFILE, L, LENSTR
C
C  Set Fortran STATUS type
C
      IF (STAT .EQ. 'NEW' .OR. STAT .EQ. 'OPTOUT') THEN
        STAT2 = 'NEW'
      ELSE
        STAT2 = 'OLD'
      END IF
C
C  Ask user for file name, and check for no response (blank line)
C
  100 WRITE(*, '(A, A, A\)') ' ', PROMPT, ': '
      READ(*, '(A)') OPNFIL
      IF (OPNFIL .EQ. ' ') THEN
        IF (STAT .EQ. 'OPTIN' .OR. STAT .EQ. 'OPTOUT') THEN
          RETURN
        ELSE IF (STAT .EQ. 'NEW') THEN
          WRITE (*,'(A)') ' Output file is required!'
          GO TO 100
        ELSE
          WRITE (*,'(A)') ' Input file is required!'
          GO TO 100
        END IF
      END IF
C
C  Deal with existance of file (or lack thereof)
C
      INQUIRE (FILE=OPNFIL, EXIST=ISIT)
      IF ((.NOT. ISIT) .AND. (STAT2 .EQ. 'OLD')) THEN
        L = LENSTR(OPNFIL)
        WRITE (*, '(A, A, A)') ' File "', OPNFIL(:L),
     &                '" does not exist.  Try again.'
        GO TO 100
      ELSE IF (ISIT .AND. STAT2 .EQ. 'NEW') THEN
        OPEN (IFILE, FILE=OPNFIL)
        CLOSE (IFILE, STATUS='DELETE')
      END IF
C
C  Open file and write blank line on screen
C
      OPEN(IFILE, STATUS=STAT2, FILE=OPNFIL)
      WRITE (*,'(A)') ' '
      RETURN
      END
```

```
C=====================================================================
      SUBROUTINE OPNOUT(IFILE)
C=====================================================================
C  This subroutine prompts the user for the name of a file set that
C  will be created to store time histories of the 9 output variables
C  computed by the Passenger car handling model simulation program.
C
C  A text file is created and opened, and labeling information is
C  written to facilitate post-processing of the data.  Then, the text
C  file is closed and a corresponding binary file is created and opened
C  to store the numerical values of the output variables.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      CHARACTER*80 FNOUT, INFILE, OPNFIL, TITLE
      LOGICAL      ISIT
      REAL         CA1, CA2, CAM1, CAM2, CCOEF1, CE, CG1, CROLL,
     &             DEGREES, GEES, H, IPRINT, IXX, IXZ, IZZR, KROLL,
     &             KRS2, L, NRBI33, NRBM, PARS, RBM, SPEED, STEER,
     &             STEP, STOPT, T, THETAR
      INTEGER      IFILE, ILOOP, IPRNT2, LENSTR, LSTRNG, MAXBUF,
     &             NBYTES, NCHAN, NCOORD, NPARS, NRECS, NSAMP, NSCAN,
     &             NSPEED, NUMKEY, NVARS
      CHARACTER*32 GENNAM, LONGNM, RIGBOD
      CHARACTER*24 TIMEDT
      CHARACTER*8  CHAR8, SHORTN, UNITSN
C
      PARAMETER    (NPARS = 24)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE (PARS(22), STEP), (PARS(23), STOPT)
C
      PARAMETER    (NVARS = 9, NUMKEY = 1)
      DIMENSION    LONGNM(NVARS), GENNAM(NVARS), RIGBOD(NVARS),
     &             SHORTN(NVARS), UNITSN(NVARS)
C
C  Prompt user to provide name of output file.  File is opened and
C  attached to Fortran unit IFILE.
C
      FNOUT = OPNFIL('Name of (required) file for time history outputs',
     &    'NEW', IFILE)
      NCHAN = 0
      IPRNT2 = PARS(10)
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Lateral Acceleration'
      SHORTN (NCHAN) = 'Ay'
      GENNAM (NCHAN) = 'Translational Acceleration'
      UNITSN (NCHAN) = 'g''s'
      RIGBOD (NCHAN) = 'Rolling Body'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Yaw Rate'
```

```
      SHORTN (NCHAN) = 'r'
      GENNAM (NCHAN) = 'Angular Speed'
      UNITSN (NCHAN) = 'deg/s'
      RIGBOD (NCHAN) = 'Non-Rolling Body'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Front slip angle'
      SHORTN (NCHAN) = 'alpha f'
      GENNAM (NCHAN) = 'Slip Angle'
      UNITSN (NCHAN) = 'deg'
      RIGBOD (NCHAN) = 'Non-Rolling Body'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Rear slip angle'
      SHORTN (NCHAN) = 'alpha r'
      GENNAM (NCHAN) = 'Slip Angle'
      UNITSN (NCHAN) = 'deg'
      RIGBOD (NCHAN) = 'Non-Rolling Body'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Side force, front axle'
      SHORTN (NCHAN) = 'FY1'
      GENNAM (NCHAN) = 'Force'
      UNITSN (NCHAN) = 'lb'
      RIGBOD (NCHAN) = 'Non-Rolling Body'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Side force, rear axle'
      SHORTN (NCHAN) = 'FY2'
      GENNAM (NCHAN) = 'Force'
      UNITSN (NCHAN) = 'lb'
      RIGBOD (NCHAN) = 'Non-Rolling Body'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Aligning moment, front axle'
      SHORTN (NCHAN) = 'MZ1'
      GENNAM (NCHAN) = 'Moment'
      UNITSN (NCHAN) = 'in-lb'
      RIGBOD (NCHAN) = 'Non-Rolling Body'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Aligning moment, rear axle'
      SHORTN (NCHAN) = 'MZ2'
      GENNAM (NCHAN) = 'Moment'
      UNITSN (NCHAN) = 'in-lb'
      RIGBOD (NCHAN) = 'Non-Rolling Body'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'roll moment from suspension'
      SHORTN (NCHAN) = 'ROLLM'
      GENNAM (NCHAN) = 'Moment'
      UNITSN (NCHAN) = 'in-lb'
      RIGBOD (NCHAN) = 'Rolling Body'
C
C   Write Header Info for ERD file
C
C  Set parameters needed to write header for ERD format file
```

```
C  NUMKEY = 1 for 32-bit floating-point binary
C  NSAMP  = number of samples
C  NRECS  = number of "records" in output file
C  NBYTES = number of bytes/record
C
      NSAMP = STOPT / STEP / IPRNT2 + 1
      NBYTES = 4 * NCHAN
      NRECS = NSAMP
C
C  Write standard ERD file heading.
C
      WRITE(IFILE, '(A)') 'ERDFILEV2.00'
      WRITE(IFILE, 100) NCHAN, NSAMP, NRECS, NBYTES, NUMKEY, STEP*IPRNT2
      WRITE(IFILE, '(A,A)') 'TITLE   ', TITLE
      WRITE(IFILE, 110) 'SHORTNAM', (SHORTN(ILOOP), ILOOP=1, NCHAN)
      WRITE(IFILE, 120) 'LONGNAME', (LONGNM(ILOOP), ILOOP=1, NCHAN)
      WRITE(IFILE, 110) 'UNITSNAM', (UNITSN(ILOOP), ILOOP=1, NCHAN)
      WRITE(IFILE, 120) 'GENNAME ', (GENNAM(ILOOP), ILOOP=1, NCHAN)
      WRITE(IFILE, 120) 'RIGIBODY', (RIGBOD(ILOOP), ILOOP=1, NCHAN)
      WRITE(IFILE, '(A)') 'XLABEL  Time'
      WRITE(IFILE, '(A)') 'XUNITS  sec'
C
      IF (INFILE .EQ. ' ') THEN
        WRITE(IFILE, '(A)') 'HISTORY No input file (used defaults)'
      ELSE
        WRITE(IFILE, '(A, A)') 'HISTORY Input parameter file was ',
     &        INFILE
      END IF
      CALL TIMDAT(TIMEDT)
      WRITE(IFILE, '(A,A)')
     & 'HISTORY Data generated with Passenger car handling model at '
     &, TIMEDT
      WRITE(IFILE, '(A)') 'END'
C
C  Close (text) header and create binary file.
C
      CLOSE(IFILE)
      LSTRNG = LENSTR(FNOUT)
      FNOUT = FNOUT (:LSTRNG) // '.BIN'
      INQUIRE(FILE=FNOUT, EXIST=ISIT)
      IF (ISIT) THEN
        OPEN (IFILE, FILE=FNOUT)
        CLOSE (IFILE, STATUS='DELETE')
      END IF
C
      OPEN(IFILE, FILE=FNOUT, STATUS='NEW', ACCESS='SEQUENTIAL',
     &      FORM='UNFORMATTED')
C
  100 FORMAT (5(I6,','),G13.6)
  110 FORMAT (A8, 9A8)
  120 FORMAT (A8, 9A32)
      RETURN
      END
```

```
C=======================================================================
      SUBROUTINE OUTPUT(IFILE, T, Q, QP, U, UP)
C=======================================================================
C  --> IFILE integer Fortran i/o unit for output
C  --> T     real    time
C  --> Q     real    array of 4 generalized coordinates
C  --> QP    real    array of derivitives of Q
C  --> U     real    array of 3 generalized speeds
C  --> UP    real    array of derivatives of U
C
C  This subroutine writes the values of the 9 output variables computed
C  by the Passenger car handling model simulation program into an
C  output file, using the values at time T.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
       IMPLICIT NONE
       CHARACTER*80 INFILE, TITLE
       REAL         C, CA1, CA2, CAM1, CAM2, CCOEF1, CE, CG1, CROLL,
     &              DEGREES, FORCEM, GEES, H, IPRINT, IXX, IXZ, IZZR,
     &              KROLL, KRS2, L, NRBI33, NRBM, OUTBUF, PARS, PC, Q,
     &              QP, RBM, S, SPEED, STEER, STEP, STOPT, T, THETAR, U,
     &              UP, Z
       INTEGER      IFILE, ILOOP, NCOORD, NPARS, NSPEED, NVARS
C
       PARAMETER    (NCOORD = 4, NSPEED = 3, NVARS = 9)
       DIMENSION    Q(NCOORD), QP(NCOORD), U(NSPEED), UP(NSPEED),
     &              OUTBUF(NVARS)
       DIMENSION    PC(43)
       COMMON       /PRCMP/ PC
       SAVE         /PRCMP/
C
       DIMENSION    C(4), FORCEM(5), S(4), Z(30)
       COMMON       /DYVARS/ C, FORCEM, S, Z
       SAVE         /DYVARS/
C
       PARAMETER    (NPARS = 24)
       DIMENSION    PARS(NPARS)
       COMMON       /INPARS/ PARS, TITLE, INFILE
       SAVE         /INPARS/
C
       EQUIVALENCE  (PARS(1), CA1), (PARS(2), CA2), (PARS(3), CAM1),
     &              (PARS(4), CAM2), (PARS(5), CCOEF1), (PARS(6), CE),
     &              (PARS(7), CG1), (PARS(8), CROLL), (PARS(9), H),
     &              (PARS(10), IPRINT), (PARS(11), IXX), (PARS(12),
     &              IXZ), (PARS(13), IZZR), (PARS(14), KROLL),
     &              (PARS(15), KRS2), (PARS(16), L), (PARS(17), NRBI33),
     &              (PARS(18), NRBM), (PARS(19), RBM), (PARS(20),
     &              SPEED), (PARS(21), STEER), (PARS(22), STEP),
     &              (PARS(23), STOPT), (PARS(24), THETAR)
       PARAMETER    (GEES = 386.2, DEGREES = 57.29577951308232)
       Z(11) = Q(4)*S(3)
       Z(12) = PC(5)*Z(11)
       Z(13) = (-Z(12) + C(3))
       Z(14) = Z(13)*C(3)
       Z(15) = Q(4)*C(3)
```

```
      Z(16) = PC(5)*Z(15)
      Z(17) = (Z(16) + S(3))
      Z(18) = Z(17)*S(3)
      Z(19) = (Z(14) + Z(18))
      Z(20) = Z(19)*UP(1)
      Z(21) = PC(31)*UP(2)
      Z(22) = H*UP(3)
      Z(23) = (Z(21) + Z(22))
      Z(24) = Z(17)**2
      Z(25) = Z(13)**2
      Z(26) = (Z(24) + Z(25))
      Z(27) = Z(23)*Z(26)
      Z(28) = (Z(4) + Z(20) + Z(27))
      Z(29) = 1.0/SQRT(Z(26))
      Z(30) = Z(28)*Z(29)
C
C  fill buffer with output variables.
C
      OUTBUF(1) = Z(30)/GEES
      OUTBUF(2) = DEGREES*U(2)
      OUTBUF(3) = -DEGREES*Z(1)
      OUTBUF(4) = -DEGREES*Z(2)
      OUTBUF(5) = -FORCEM(1)
      OUTBUF(6) = -FORCEM(2)
      OUTBUF(7) = -FORCEM(3)
      OUTBUF(8) = -FORCEM(4)
      OUTBUF(9) = -FORCEM(5)
C
C  The following line writes to an unformatted binary file
C
      WRITE (IFILE) (OUTBUF(ILOOP), ILOOP=1, NVARS)
C
C--The next 3 lines are for the Macintosh
C
      IF (T .EQ. 0.) WRITE (*, '(/A/7X,A)') ' Progress:','sec'
      CALL TOOLBX (Z'89409000', 0, -11)
      WRITE (*, '(F6.2)') T
      RETURN
      END
C=======================================================================
      SUBROUTINE PRECMP
C=======================================================================
C  This subroutine defines all constants that can be pre-computed for
C  the Passenger car handling model.  The constants are put into the
C  COMMON block /PRECMP/
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      CHARACTER*80 INFILE, TITLE
      REAL         CA1, CA2, CAM1, CAM2, CCOEF1, CE, CG1, CROLL,
     &             DEGREES, GEES, H, IPRINT, IXX, IXZ, IZZR, KROLL,
     &             KRS2, L, NRBI33, NRBM, PARS, PC, RBM, SPEED, STEER,
     &             STEP, STOPT, THETAR
      INTEGER      NPARS
C
```

```
      DIMENSION    PC(43)
      COMMON       /PRCMP/ PC
      SAVE         /PRCMP/
C
      PARAMETER    (NPARS = 24)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE  (PARS(1), CA1), (PARS(2), CA2), (PARS(3), CAM1),
     &             (PARS(4), CAM2), (PARS(5), CCOEF1), (PARS(6), CE),
     &             (PARS(7), CG1), (PARS(8), CROLL), (PARS(9), H),
     &             (PARS(10), IPRINT), (PARS(11), IXX), (PARS(12),
     &             IXZ), (PARS(13), IZZR), (PARS(14), KROLL),
     &             (PARS(15), KRS2), (PARS(16), L), (PARS(17), NRBI33),
     &             (PARS(18), NRBM), (PARS(19), RBM), (PARS(20),
     &             SPEED), (PARS(21), STEER), (PARS(22), STEP),
     &             (PARS(23), STOPT), (PARS(24), THETAR)
      PARAMETER    (GEES = 386.2, DEGREES = 57.29577951308232)
C
      PC(1) = CG1*CCOEF1
      PC(2) = 1.0/SPEED
      PC(3) = COS(THETAR)
      PC(4) = CROLL*COS(THETAR)
      PC(5) = SIN(THETAR)
      PC(6) = CE*COS(THETAR)
      PC(7) = H*SIN(THETAR)
      PC(8) = GEES*COS(THETAR)
      PC(9) = GEES*SIN(THETAR)
      PC(10) = RBM*(CE*COS(THETAR) + H*SIN(THETAR))
      PC(11) = H*RBM
      PC(12) = (RBM + NRBM)
      PC(13) = (NRBI33 + COS(THETAR)*(IZZR*COS(THETAR) +
     &          IXZ*SIN(THETAR)) + RBM*(CE*COS(THETAR) +
     &          H*SIN(THETAR))**2 + (IXZ*COS(THETAR) +
     &          IXX*SIN(THETAR))*SIN(THETAR))
      PC(14) = (IXZ*COS(THETAR) + IXX*SIN(THETAR) + H*RBM
     &         *(CE*COS(THETAR) + H*SIN(THETAR)))
      PC(15) = (RBM*H**2 + IXX)
      PC(16) = PC(10)/PC(12)
      PC(17) = PC(11)/PC(12)
      PC(18) = PC(10)*PC(16)
      PC(19) = (PC(13) -PC(18))
      PC(20) = PC(10)*PC(17)
      PC(21) = (PC(14) -PC(20))
      PC(22) = PC(21)/PC(19)
      PC(23) = PC(11)*PC(16)
      PC(24) = (PC(14) -PC(23))
      PC(25) = PC(11)*PC(17)
      PC(26) = PC(22)*PC(24)
      PC(27) = (PC(15) -PC(25) -PC(26))
      PC(28) = 1.0/PC(27)
      PC(29) = 1.0/PC(19)
      PC(30) = 1.0/PC(12)
      PC(31) = (PC(6) + PC(7))
      PC(32) = RBM*PC(8)
      PC(33) = PC(9)*PC(11)
```

```
      PC(34) = CE*RBM
      PC(35) = PC(8)*PC(34)
      PC(36) = (PC(33) + PC(35))
      PC(37) = PC(11)*PC(28)
      PC(38) = PC(17)*PC(28)
      PC(39) = PC(22)*PC(28)
      PC(40) = PC(24)*PC(29)
      PC(41) = PC(11)*PC(30)
      PC(42) = PC(10)*PC(30)
      PC(43) = L*PC(2)
      RETURN
      END
*****************************************************************
      SUBROUTINE TIMDAT (TIMEDT)
*****************************************************************
C  Get date and time.  On the Mac, this requires the TIME and DATE
C  subroutines from Absoft.
C
C  by M. Sayers, 1986.
C
C  <-- TIMEDT char*24   string containing time & date.
C
      CHARACTER*24 TIMEDT
      CHARACTER*36 MONTHS
      INTEGER*4 M, IDAY, IYEAR, ISEC
      INTEGER*2 YEAR, MONTH, DAY, HOUR, MIN, SEC, I100
      MONTHS = 'JanFebMarAprMayJunJulAugSepOctNovDec'

C--The following 4 lines are for the IBM PC (using Microsoft
C--time and date functions)
*      CALL GETDAT (YEAR, MONTH, DAY)
*      CALL GETTIM (IHOUR, MIN, SEC, I100)
*      WRITE (TIMEDT, 100) IHOUR, MIN, MONTHS (MONTH*3-2:MONTH*3),
*     &       DAY, YEAR

C--get time for MTS version
C      CALL TIME(22, 0, TIMEDT)

C--The following 5 lines are for the Apple Mac
C--(using Absoft time & date functions)
      CALL DATE (M, IDAY, IYEAR)
      CALL TIME (ISEC)
      WRITE (TIMEDT, 100)
     &       ISEC/3600, MOD (ISEC, 3600) / 60, MONTHS (M*3-2:M*3),
     &       IDAY, 1900 + IYEAR

  100 FORMAT (I2,':',I2.2,' on ',A3,I3,',',I5)
      RETURN
      END
```

APPENDIX C — FOUR-BAR LINKAGE

This appendix contains the complete source code for the four-bar linkage described in Section 9.3.

```
C  4-Bar linkage simulation program.
C  Version created December 11, 1989 by AUTOSIM
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
C  This program simulates the 4-bar linkage by numerically integrating
C  the 4 ordinary differential equations that describe the kinematics
C  and dynamics of the system.  The 4-bar linkage is composed of 3
C  bodies and has 1 degree of freedom.
C
C  Each derivative evaluation requires 141 multiply/divides, 81
C  add/subtracts, and 6 function/subroutine calls.
C
C  Bodies:
C  =======
C  A; parent=N; 1 DOF: Q(1)
C  B; parent=A; 1 DOF: Q(2)
C  C; parent=N; 1 DOF: Q(3)
C
C  Generalized Coordinates:
C  ========================
C    Q(1): Rotation of A relative to the inertial reference about axis
C          #3. (rad)
C    Q(2): Rotation of B relative to A about axis #3. (rad)
C    Q(3): Rotation of C relative to the inertial reference about axis
C          #3. (rad)
C
C  Independent Speeds:
C  ===================
C    U(1): Abs. rot. of A, axis 3. (rad/s)
C
C  Nonholonomic Constraints:
C  =========================
C  Rot. of B relative to A, axis 3.: -U(1)*(1 -L1*(S(2) -(L1 -L5)
C         *(C(3)*(C(1)*C(2)**2 -C(2)*S(1)*S(2)) + C(2)*(C(2)*S(1) +
C          C(1)*S(2))*S(3))*(S(2)*(C(1)*C(3) + S(1)*S(3)) + C(2)
C         *(C(3)*S(1) -C(1)*S(3)))/(L1 -L5 -(L1 -L5)*(S(2)*(C(1)*C(3) +
C          S(1)*S(3)) + C(2)*(C(3)*S(1) -C(1)*S(3)))**2))/L4)
C  Abs. rot. of C, axis 3.: L1*U(1)*(C(3)*(C(1)*C(2)**2
C          -C(2)*S(1)*S(2)) + C(2)*(C(2)*S(1) + C(1)*S(2))*S(3))/(L1
C          -L5 -(L1 -L5)*(S(2)*(C(1)*C(3) + S(1)*S(3)) + C(2)
C         *(C(3)*S(1) -C(1)*S(3)))**2)
C
```

```
C  Active Forces:
C  ==============
C  FORCEM(1): strut
C
C  Program Sections:
C  =================
C  MAIN -- Control flow of program and perform numerical integration
C
C  BLOCK DATA -- initialize variables in COMMON blocks
C  DIFEQN (T, Q, QP, U, UP) -- compute QP and UP given T, Q, and U
C  ECHO (IFILE, Q, U) -- create output file with echo of input
C            parameters
C  INITNR(X, ALPHA, BETA, Q) -- compute ALPHA and BETA coeficients for
C            MNEWT
C  INPUT (Q, U) -- read parameters and initial conditions
C  Function LENSTR (STRING) -- count characters in left-justified
C            string
C  LUDCMP(A, N, NP, INDX, D) -- decompose matrix into LUD form [1]
C  LUBKSB(A, N, NP, INDX, B) -- solve simultaneous equations [1]
C  MNEWT(NTRIAL,X,N,TOLX,TOLF,Q) -- solve for initial conditions [1]
C  Function NORMA(A) -- Normalize angle
C  Function OPNFIL(PROMPT, STAT, IUNIT) -- let user open file
C  OPNOUT(IFILE) -- create output file and write header
C  OUTPUT(IFILE, T, Q, QP, U, UP) -- write variables at time T
C  PRECMP -- pre-compute constants used in simulation
C  TIMDAT(TIMEDT) -- get time and date from computer
C
C  [1] Press et. al., Numerical Recipes, The Art of Scientific
C      Computing. Cambridge Press, 1986.
C
      IMPLICIT NONE
      CHARACTER*80 INFILE, TITLE
      REAL         NORMA, PARS, STEP, STEP2, STOPT, T, Y, YM, YP
      INTEGER      I, IECHO, IFILE, ILOOP1, ILOOP2, IPRNT2, ISEC1,
     &             ISEC2, NCOORD, NLOOP, NPARS, NSPEED, NTOT
C
      PARAMETER    (NCOORD = 3, NSPEED = 1, IFILE = 1, NTOT = 4)
      DIMENSION    Y(NTOT), YM(NTOT), YP(NTOT)
      PARAMETER    (NPARS = 14)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE (PARS(13), STEP), (PARS(14), STOPT)
C
      WRITE(*, '(5A)')
     & ' 4-Bar linkage simulation program.'
      WRITE(*, '(5A)')
     & ' Version created December 11, 1989 by AUTOSIM'
      WRITE(*, '(5A)')
     & '  '
C
C  Read input data
C
      CALL INPUT(Y, Y(NCOORD + 1))
      IPRNT2 = PARS(4)
C
```

```
C   Compute constants in common block /PRCMP/ before starting.
C
      CALL PRECMP
C
C  compute initial values of dependent coordinates.
C
      CALL MNEWT(20, Y(2), 2, 1.E-06, 1.E-06, Y)
C
C   Option to echo data to output file
C
      CALL ECHO(IFILE, Y, Y(NCOORD + 1))
C
C   Set up output file with simulated time histories
C
      CALL OPNOUT (IFILE)
      CALL TIME (ISEC1)
C
C Start by evaluating derivatives and printing variables at t=0
C
      T = 0.
      CALL DIFEQN(T, Y, YP, Y(NCOORD + 1), YP(NCOORD + 1))
      CALL OUTPUT(IFILE, T, Y, YP, Y(NCOORD + 1), YP(NCOORD + 1))
C
C Integration loop. Continue until printout time reaches final time.
C Use two evaluations of the derivatives to integrate over the step.
C
      NLOOP = STOPT / STEP / IPRNT2 + 1
      STEP2 = STEP / 2.
      DO 60 ILOOP1 = 1, NLOOP
        DO 50 ILOOP2 = 1, IPRNT2
          DO 10 I = 1, NTOT
            YM(I) = Y(I) + STEP2 * YP(I)
   10     CONTINUE
          CALL DIFEQN (T + STEP2, YM, YP, YM(NCOORD + 1),
     &                 YP(NCOORD + 1))
C
          DO 20 I = 1, NTOT
            Y(I) = Y(I) + STEP * YP(I)
   20     CONTINUE
C
          T = T + STEP
          CALL DIFEQN (T, Y, YP,  Y(NCOORD + 1), YP(NCOORD + 1))
   50   CONTINUE
        CALL OUTPUT (IFILE, T, Y, YP, Y(NCOORD + 1), YP(NCOORD + 1))
        IF (T .GE. STOPT) GO TO 70
   60 CONTINUE
   70 CONTINUE
C
      CALL TIME (ISEC2)
C
C End of integration loop.  Print final status of run
C
      WRITE (*, *) ' Termination at time =', T, ' sec.'
      WRITE (*,*) ' Computation efficiency: ', (ISEC2 - ISEC1) / T,
     &            ' sec/sim. sec'
      WRITE (*,*) ' '
```

```
      CLOSE(IFILE)
      PAUSE ' Done'
      END
C=======================================================================
      BLOCK DATA
C=======================================================================
      CHARACTER*80 INFILE, TITLE
      REAL         PARS
      INTEGER      NPARS
C
      PARAMETER    (NPARS = 14)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      DATA         PARS /1.0, 10.0, 100.0, 1.0, 10000.0, 0.5, 0.1, 0.2,
     &             0.3, 0.1, 0.3, 0.5, 0.005, 1.0/
      DATA         INFILE /' '/
      DATA         TITLE  /'Default parameter values'/
      END
C=======================================================================
      SUBROUTINE DIFEQN(T, Q, QP, U, UP)
C=======================================================================
C  This subroutine defines the equations of motion for the 4-bar
C  linkage, which includes 1 degree of freedom.
C
C  --> T  real  time
C  --> Q  real  array of 3 generalized coordinates
C  <-- QP real  array of derivitives of Q
C  --> U  real  array of 1 generalized speed
C  <-- UP real  array of derivative of U
C
C  Each derivative evaluation requires 141 multiply/divides, 81
C  add/subtracts, and 6 function/subroutine calls.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      CHARACTER*80 INFILE, TITLE
      REAL         BI33, BM, C, D, DEGREES, FORCEM, GEES, IPRINT, K,
     &             L1, L2, L3, L4, L5, L6, L7, PARS, PC, Q, QP, S,
     &             STEP, STOPT, T, U, UP, Z
      INTEGER      NCOORD, NPARS, NSPEED
C
      PARAMETER    (NCOORD = 3, NSPEED = 1)
      DIMENSION    Q(NCOORD), QP(NCOORD), U(NSPEED), UP(NSPEED)
      DIMENSION    C(3), FORCEM(1), S(3), Z(71)
      COMMON       /DYVARS/ C, FORCEM, S, Z
      SAVE         /DYVARS/
C
      PARAMETER    (NPARS = 14)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE  (PARS(1), BI33), (PARS(2), BM), (PARS(3), D),
```

```
     &             (PARS(4), IPRINT), (PARS(5), K), (PARS(6), L1),
     &             (PARS(7), L2), (PARS(8), L3), (PARS(9), L4),
     &             (PARS(10), L5), (PARS(11), L6), (PARS(12), L7),
     &             (PARS(13), STEP), (PARS(14), STOPT)
      DIMENSION    PC(30)
      COMMON       /PRCMP/ PC
      SAVE         /PRCMP/
C
      PARAMETER    (GEES = 9.80665)
      S(1) = SIN(Q(1))
      S(2) = SIN(Q(2))
      S(3) = SIN(Q(3))
      C(1) = COS(Q(1))
      C(2) = COS(Q(2))
      C(3) = COS(Q(3))
C
C
C  Kinematical equations
C
      Z(1) = L5*S(3)
      Z(2) = C(3)*S(1)
      Z(3) = C(1)*S(3)
      Z(4) = (Z(2) -Z(3))
      Z(5) = Z(4)*S(2)
      Z(6) = C(1)*C(3)
      Z(7) = S(1)*S(3)
      Z(8) = (Z(6) + Z(7))
      Z(9) = Z(8)*C(2)
      Z(10) = (Z(5) -Z(9) + C(3))
      Z(11) = L4*Z(10)
      Z(12) = L1*Z(4)
      Z(13) = (Z(1) -Z(11) + Z(12))
      Z(14) = L5*C(3)
      Z(15) = Z(8)*S(2)
      Z(16) = Z(4)*C(2)
      Z(17) = (Z(15) + Z(16) + S(3))
      Z(18) = L4*Z(17)
      Z(19) = L1*Z(8)
      Z(20) = (Z(14) + Z(18) -Z(19))
      Z(21) = Z(13)/Z(20)
      Z(22) = (-Q(3) + Z(21))
      Q(3) = -Z(22)
      Z(23) = L1*C(2)
      Z(24) = C(1)*C(2)
      Z(25) = S(1)*S(2)
      Z(26) = (Z(24) -Z(25))
      Z(27) = L5*Z(26)
      Z(28) = C(2)*S(1)
      Z(29) = C(1)*S(2)
      Z(30) = (Z(28) + Z(29))
      Z(31) = L4*Z(30)
      Z(32) = (-Z(5) + Z(9))
      Z(33) = PC(1)*Z(32)
      Z(34) = (Z(23) -Z(27) -Z(31) -Z(33))
      Z(35) = L4*Z(26)
      Z(36) = L5*Z(30)
      Z(37) = L1*S(2)
```

```
      Z(38) = (Z(15) + Z(16))
      Z(39) = PC(1)*Z(38)
      Z(40) = (Z(35) -Z(36) + Z(37) -Z(39))
      Z(41) = Z(34)/Z(40)
      Q(2) = (Q(2) + Z(41))
      QP(1) = U(1)
      Z(42) = -(Z(25) -C(1)*C(2))*C(2)
      Z(43) = (Z(42)*C(3) + Z(30)*C(2)*S(3))
      Z(44) = PC(1)*(1 -Z(38)**2)
      Z(45) = Z(43)/Z(44)
      Z(46) = (-PC(1)*Z(38)*Z(45) + S(2))
      Z(47) = (1 -PC(2)*Z(46))
      QP(2) = -U(1)*Z(47)
      QP(3) = L1*U(1)*Z(45)
C
C  define expression for strut
C
      Z(48) = L1*C(1)
      Z(49) = L1*S(1)
      Z(50) = (PC(21) -PC(19)*Z(26) + PC(20)*Z(30) -L2*Z(37) -L6*Z(48)
     &         -L7*Z(49) -PC(15)*C(1) -PC(16)*S(1) -PC(17)*S(2))
      Z(51) = L7*Z(30)
      Z(52) = L6*Z(26)
      Z(53) = 1.0/SQRT(Z(50))
      FORCEM(1) = -(-PC(25) + K*SQRT(Z(50)) + U(1)*Z(53)
     &     *(-PC(27)*Z(46)*(Z(23) -Z(51) -Z(52)) -PC(28)*C(1) +
     &      PC(29)*C(2) + PC(30)*S(1)))
C
C  Dynamical equations
C
      Z(54) = (Z(37) -L3*(1 -Z(47)))
      Z(55) = GEES*C(1)
      Z(56) = (-L1*U(1)**2 + GEES*S(1))
      Z(57) = Z(53)*FORCEM(1)
      Z(58) = L7*Z(57)
      Z(59) = L6*Z(57)
      Z(60) = L1*Z(57)
      Z(61) = (QP(1) -QP(3))
      Z(62) = (((Z(6) + Z(7))*Z(61) + Z(8)*QP(2))*C(2) -((Z(2)
     &         -Z(3))*Z(61) + Z(4)*QP(2))*S(2))
      Z(63) = (QP(1) + QP(2))
      Z(64) = U(1)*(QP(2)*C(2) -(2.0*Z(43)*Z(62)*Z(39)**2 + Z(44)
     &        *(PC(1)*Z(43)*Z(62) -Z(38)*((PC(22)*Z(29)*QP(2)
     &         -PC(1)*Z(30)*QP(3))*C(2)*C(3) + PC(1)*(C(3)
     &        *(Z(28)*Z(63)*C(2) + (Z(24)*QP(1) -Z(25)*QP(2))*S(2)) +
     &         (Z(42)*QP(3) -(Z(24) -Z(25))*Z(63)*C(2) +
     &         Z(30)*QP(2)*S(2))*S(3)))))/Z(44)**2)
      Z(65) = BM*Z(54)
      Z(66) = BM*Z(23)
      UP(1) = (-Z(46)*(PC(23)*Z(64) + PC(24)*(Z(23) -Z(51)
     &         -Z(52))*Z(53)*FORCEM(1)) + Z(54)*(-Z(30)*Z(58)
     &         -Z(26)*Z(59) + Z(60)*C(2)) -Z(65)*(-PC(8)*Z(64) +
     &         Z(56)*C(2) + Z(55)*S(2)) + Z(23)*(L2*Z(57) -Z(26)*Z(58)
     &         + Z(30)*Z(59) -Z(60)*S(2)) + Z(66)*(PC(9)
     &        *(U(1)*Z(46))**2 -Z(55)*C(2) +
     &         Z(56)*S(2)))/(PC(23)*Z(46)**2 + Z(54)*Z(65) +
     &         Z(23)*Z(66))
```

```
      RETURN
      END
C=======================================================================
      SUBROUTINE ECHO(IFILE, Q, U)
C=======================================================================
C  This subroutine prompts the user for the name of an optional echo
C  file for the 4-bar linkage.  If a file is selected, all of the
C  parameter values and initial conditions are written to confirm that
C  the intended values were used in the simulation.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      CHARACTER*24 TIMEDT
      CHARACTER*80 INFILE, OPNFIL, TITLE
      REAL         BI33, BM, D, DEGREES, GEES, IPRINT, K, L1, L2, L3,
     &             L4, L5, L6, L7, PARS, Q, STEP, STOPT, T, U
      INTEGER      IFILE, NCOORD, NPARS, NSPEED
C
      PARAMETER    (NPARS = 14)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE  (PARS(1), BI33), (PARS(2), BM), (PARS(3), D),
     &             (PARS(4), IPRINT), (PARS(5), K), (PARS(6), L1),
     &             (PARS(7), L2), (PARS(8), L3), (PARS(9), L4),
     &             (PARS(10), L5), (PARS(11), L6), (PARS(12), L7),
     &             (PARS(13), STEP), (PARS(14), STOPT)
      PARAMETER    (NCOORD = 3, NSPEED = 1)
      DIMENSION    Q(NCOORD), U(NSPEED)
      PARAMETER    (GEES = 9.80665)
C
      IF (OPNFIL('Name of (optional) file to echo the input data',
     &           'OPTOUT', IFILE) .EQ. ' ')  RETURN
C
      CALL TIMDAT(TIMEDT)
C
      WRITE(IFILE, '(A)') 'PARSFILE'
      WRITE(IFILE, '(5A)')
     & 'Echo file created by:'
      WRITE(IFILE, '(5A)')
     & '4-Bar linkage simulation program.'
      WRITE(IFILE, '(5A)')
     & 'Version created December 11, 1989 by AUTOSIM'
      WRITE(IFILE, '(5A)')
     & ' '
      WRITE(IFILE, '(A,T8,A)') 'TITLE', TITLE
      WRITE(IFILE, '(/A,A)') '* Input File: ', INFILE
      WRITE(IFILE, '(A, A)') '* Run was made ', TIMEDT
                 WRITE(IFILE, '(/A/)') '* PARAMETER VALUES'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'BI33', BI33,
     & 'moment of inertia of B (kg-m2)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'BM', BM,
     & 'mass of B (kg)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'D', D,
```

```
     & 'coefficient in term in strut (N-sec/rad/m)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'IPRINT', IPRINT,
     & 'number of time steps between output printing (counts)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'K', K,
     & 'stiffness coefficient in term in strut (N/m)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'L1', L1,
     & 'coordinate of attachment point for B in dir 1 (m)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'L2', L2,
     & 'coordinate of strut pt 1 in dir 2 (m)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'L3', L3,
     & 'coordinate of center of mass of B in dir 2 (m)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'L4', L4,
     & 'coordinate of b-point in dir 2 (m)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'L5', L5,
     & 'coordinate of attachment point for C in dir 1 (m)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'L6', L6,
     & 'coordinate of strut pt 2 in dir 1 (m)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'L7', L7,
     & 'coordinate of strut pt 2 in dir 2 (m)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'STEP', STEP,
     & 'simulation time step (sec)'
      WRITE(IFILE, '(A,T8,G13.6,T24,5A)') 'STOPT', STOPT,
     & 'simulation stop time (sec)'
      WRITE(IFILE, '(/A/)') '* INITIAL CONDITIONS'
      WRITE(IFILE, '(A, T8, G13.6, T24, 5A)') 'Q(1)', Q(1),
     & 'Rotation of A relative to the inertial reference about axis'
     &,' #3. (rad)'
      WRITE(IFILE, '(A, T8, G13.6, T24, 5A)') 'Q(2)', Q(2),
     & 'Rotation of B relative to A about axis #3. (rad)'
      WRITE(IFILE, '(A, T8, G13.6, T24, 5A)') 'Q(3)', Q(3),
     & 'Rotation of C relative to the inertial reference about axis'
     &,' #3. (rad)'
      WRITE(IFILE, '(A, T8, G13.6, T24, 5A)') 'U(1)', U(1),
     & 'Abs. rot. of A, axis 3. (rad/s)'
      WRITE(IFILE, '(/A)') 'END'
      CLOSE(IFILE)
      RETURN
      END
C=======================================================================
      SUBROUTINE INITNR(X, ALPHA, BETA, Q)
C=======================================================================
C  This subroutine computes coefficients for a Newton-Raphson iteration
C  needed to establish the initial values of 2 computed coordinates in
C  the 4-bar linkage.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      REAL         ALPHA, BETA, BI33, BM, D, DEGREES, GEES, IPRINT, K,
     &             L1, L2, L3, L4, L5, L6, L7, PARS, PC, Q, STEP,
     &             STOPT, T, X
      INTEGER      NPARS
      CHARACTER*80 INFILE, TITLE
C
      DIMENSION    PC(30)
      COMMON       /PRCMP/ PC
```

```
      SAVE         /PRCMP/
C
      PARAMETER    (NPARS = 14)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE  (PARS(1), BI33), (PARS(2), BM), (PARS(3), D),
     &             (PARS(4), IPRINT), (PARS(5), K), (PARS(6), L1),
     &             (PARS(7), L2), (PARS(8), L3), (PARS(9), L4),
     &             (PARS(10), L5), (PARS(11), L6), (PARS(12), L7),
     &             (PARS(13), STEP), (PARS(14), STOPT)
      PARAMETER    (GEES = 9.80665)
      DIMENSION    X(2), Q(*), BETA(*), ALPHA(2, 2)
C
      BETA(1) = (-L1*COS(X(1)) + L4*(COS(X(1))*SIN(Q(1)) +
     &      COS(Q(1))*SIN(X(1))) + L5*(COS(Q(1))*COS(X(1))
     &      -SIN(Q(1))*SIN(X(1))) + (L1 -L5)*(-SIN(X(1))
     &     *(COS(X(2))*SIN(Q(1)) -COS(Q(1))*SIN(X(2))) + COS(X(1))
     &     *(COS(Q(1))*COS(X(2)) + SIN(Q(1))*SIN(X(2)))))
      ALPHA(1,1) = -(-L5*(COS(X(1))*SIN(Q(1)) + COS(Q(1))*SIN(X(1))) +
     &      L4*(COS(Q(1))*COS(X(1)) -SIN(Q(1))*SIN(X(1))) +
     &      L1*SIN(X(1)) -(L1 -L5)*(SIN(X(1))*(COS(Q(1))*COS(X(2)) +
     &      SIN(Q(1))*SIN(X(2))) + COS(X(1))*(COS(X(2))*SIN(Q(1))
     &      -COS(Q(1))*SIN(X(2)))))
      ALPHA(1,2) = -(L1 -L5)*(COS(X(1))*(COS(X(2))*SIN(Q(1))
     &      -COS(Q(1))*SIN(X(2))) + SIN(X(1))*(COS(Q(1))*COS(X(2)) +
     &      SIN(Q(1))*SIN(X(2))))
      BETA(2) = -(L5*SIN(X(2)) -L4*(COS(X(2)) + SIN(X(1))
     &     *(COS(X(2))*SIN(Q(1)) -COS(Q(1))*SIN(X(2))) -COS(X(1))
     &     *(COS(Q(1))*COS(X(2)) + SIN(Q(1))*SIN(X(2)))) + L1
     &     *(COS(X(2))*SIN(Q(1)) -COS(Q(1))*SIN(X(2))))
      ALPHA(2,1) = -L4*(SIN(X(1))*(COS(Q(1))*COS(X(2)) +
     &      SIN(Q(1))*SIN(X(2))) + COS(X(1))*(COS(X(2))*SIN(Q(1))
     &      -COS(Q(1))*SIN(X(2))))
      ALPHA(2,2) = (L5*COS(X(2)) + L4*(SIN(X(2)) + SIN(X(1))
     &     *(COS(Q(1))*COS(X(2)) + SIN(Q(1))*SIN(X(2))) + COS(X(1))
     &     *(COS(X(2))*SIN(Q(1)) -COS(Q(1))*SIN(X(2)))) -L1
     &     *(COS(Q(1))*COS(X(2)) + SIN(Q(1))*SIN(X(2))))
      RETURN
      END
C=======================================================================
      SUBROUTINE INPUT(Q, U)
C=======================================================================
C  This subroutine prompts the user for the name of an optional
C  parameter file for the 4-bar linkage.  If a file is selected,
C  parameter values are read to override the default values.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      LOGICAL      ISIT
      CHARACTER*80 BUFFER, ECHFIL, INFILE, OPNFIL, QUEUE, TITLE
      CHARACTER*8  CHAR8, NAMES, QC, UC
      REAL         BI33, BM, D, DEGREES, GEES, IPRINT, K, L1, L2, L3,
     &             L4, L5, L6, L7, PARS, PSCALE, Q, QINIT, QSCALE,
```

```
     &             STEP, STOPT, T, U, UINIT, USCALE
      INTEGER      IFILE, ILOOP, IQUEUE, LENSTR, LSTRNG, MAXQ, NCOORD,
     &             NPARS, NQUEUE, NSPEED
C
      PARAMETER    (NPARS = 14)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE  (PARS(1), BI33), (PARS(2), BM), (PARS(3), D),
     &             (PARS(4), IPRINT), (PARS(5), K), (PARS(6), L1),
     &             (PARS(7), L2), (PARS(8), L3), (PARS(9), L4),
     &             (PARS(10), L5), (PARS(11), L6), (PARS(12), L7),
     &             (PARS(13), STEP), (PARS(14), STOPT)
      PARAMETER    (GEES = 9.80665)
      PARAMETER    (NCOORD = 3, NSPEED = 1, MAXQ = 20, IFILE = 1)
      DIMENSION    NAMES(NPARS), Q(NCOORD), QC(NCOORD), QINIT(NCOORD),
     &             QUEUE(MAXQ), QSCALE(NCOORD), U(NSPEED), UC(NSPEED),
     &             UINIT(NSPEED), USCALE(NSPEED)
      DATA         QINIT /NCOORD*0./, UINIT /NSPEED*0./
      DATA         QC /'Q(1)', 'Q(2)', 'Q(3)'/
      DATA         UC /'U(1)'/
      DATA         QSCALE /1, 1, 1/
      DATA         USCALE /1/
      DATA         NAMES /'BI33', 'BM', 'D', 'IPRINT', 'K', 'L1', 'L2',
     &             'L3', 'L4', 'L5', 'L6', 'L7', 'STEP', 'STOPT'/
      NQUEUE = 0
      IQUEUE = 0
C
C  Open file with parameter values and initial conditions
C
    5 INFILE = OPNFIL ('Name of (optional) file with parameter values',
     &         'OPTIN', IFILE)
    6 IF (INFILE .NE. '') THEN
        READ(IFILE, '(A)') CHAR8
C
        IF (CHAR8 .EQ. 'END') GO TO 100
        IF (CHAR8 .NE. 'PARSFILE') THEN
          CLOSE(IFILE)
          WRITE (*, '(A)') ' Error--File did not begin with "PARSFILE"'
          IF (IQUEUE .EQ. 0) THEN
            GO TO 5
          ELSE
            GO TO 100
          END IF
        END IF
C
C  Read line from file.  CHAR8 is the keyword, checked for:
C   o  TITLE keyword,
C   o  parameter keyword (from NAMES array),
C   o  initial value of generalized coordinate (keyword from QC array),
C   o  initial value of generalized speed (keyword from UC array),
C   o  (possibly) keyword for other input subroutine, or
C   o  END keyword.
C  All other lines are ignored.  Also, all lines are ignored after END
C  is found, and any line  with a '*' in column 1 is ignored.
C
```

```
   10   READ(IFILE, '(A8,A80)', END=100, ERR=100) CHAR8, BUFFER
        IF (CHAR8 .EQ. 'TITLE') THEN
          TITLE = BUFFER
          GO TO 10
        ELSE IF (CHAR8 .EQ. 'END') THEN
          GO TO 100
        ELSE IF (CHAR8(1:1) .EQ. '*') THEN
          GO TO 10
        ELSE IF (CHAR8 .EQ. 'PARSFILE') THEN
          INQUIRE (FILE=BUFFER, EXIST=ISIT)
          IF (ISIT) THEN
            NQUEUE = NQUEUE + 1
            QUEUE (NQUEUE) = BUFFER
          ELSE
            LSTRNG = LENSTR (BUFFER)
            WRITE (*,'(A,A,A)') 'Error--PARSFILE "', BUFFER(:LSTRNG),
     &                          '"not found (skipped).'
          END IF
          GO TO 10
        END IF
C
C  Check for names of parameters
C
        DO 20 ILOOP = 1, NPARS
          IF (CHAR8 .EQ. NAMES(ILOOP)) THEN
            READ(BUFFER, '(G13.0)') PARS(ILOOP)
            GO TO 10
          END IF
   20   CONTINUE
C
C  Check for names of generalized coordinates (initial conditions)
C
        DO 30 ILOOP = 1, NCOORD
          IF (CHAR8 .EQ. QC(ILOOP)) THEN
            READ(BUFFER, '(G13.0)') QINIT(ILOOP)
            GO TO 10
          END IF
   30   CONTINUE
C
C  Check for names of generalized speeds (initial conditions)
C
        DO 40 ILOOP = 1, NSPEED
          IF (CHAR8 .EQ. UC(ILOOP)) THEN
            READ(BUFFER, '(G13.0)') UINIT(ILOOP)
            GO TO 10
          END IF
   40   CONTINUE
C
        GO TO 10
      END IF
C
C  Close this file and process other PARS files that were referenced.
C
  100 CLOSE (IFILE)
      IF (IQUEUE .LT. NQUEUE) THEN
        IQUEUE = IQUEUE + 1
        INFILE = QUEUE (IQUEUE)
```

```
        OPEN (IFILE, STATUS='OLD', FILE=INFILE)
        WRITE (*, '(A,A)') ' Reading from PARSFILE ', INFILE
        GO TO 6
      END IF
C
C  Set initial conditions
C
      DO 110 ILOOP = 1, NCOORD
  110 Q(ILOOP) = QINIT(ILOOP) / QSCALE(ILOOP)
C
      DO 120 ILOOP = 1, NSPEED
  120 U(ILOOP) = UINIT(ILOOP) / USCALE(ILOOP)
C
C  Convert units as needed.
C
C
      RETURN
      END
C=======================================================================
      FUNCTION LENSTR (STRING)
C=======================================================================
C  count characters in left-justified string.   M. Sayers, 8-9-87
C
      CHARACTER*(*) STRING
      N = LEN (STRING)
      DO 10 L = N, 1, -1
        IF (STRING(L:L) .NE. ' ') THEN
          LENSTR = L
          RETURN
        END IF
   10 CONTINUE
      LENSTR = 1
      RETURN
      END
C=======================================================================
      SUBROUTINE LUDCMP(A,N,NP,INDX,D)
C=======================================================================
C This subroutine is from Numerical Recipes.  It decomposes a square
C matrix A into LU form.
C
      PARAMETER (NMAX=100,TINY=1.0E-20)
      DIMENSION A(NP,NP),INDX(N),VV(NMAX)
      D=1.
      DO 12 I=1,N
        AAMAX=0.
        DO 11 J=1,N
          IF (ABS(A(I,J)).GT.AAMAX) AAMAX=ABS(A(I,J))
11      CONTINUE
        IF (AAMAX.EQ.0.) PAUSE 'Singular matrix.'
        VV(I)=1./AAMAX
12    CONTINUE
      DO 19 J=1,N
        IF (J.GT.1) THEN
          DO 14 I=1,J-1
            SUM=A(I,J)
            IF (I.GT.1)THEN
              DO 13 K=1,I-1
```

```
                SUM=SUM-A(I,K)*A(K,J)
13            CONTINUE
              A(I,J)=SUM
            ENDIF
14        CONTINUE
        ENDIF
        AAMAX=0.
        DO 16 I=J,N
          SUM=A(I,J)
          IF (J.GT.1)THEN
            DO 15 K=1,J-1
              SUM=SUM-A(I,K)*A(K,J)
15          CONTINUE
            A(I,J)=SUM
          ENDIF
          DUM=VV(I)*ABS(SUM)
          IF (DUM.GE.AAMAX) THEN
            IMAX=I
            AAMAX=DUM
          ENDIF
16      CONTINUE
        IF (J.NE.IMAX)THEN
          DO 17 K=1,N
            DUM=A(IMAX,K)
            A(IMAX,K)=A(J,K)
            A(J,K)=DUM
17        CONTINUE
          D=-D
          VV(IMAX)=VV(J)
        ENDIF
        INDX(J)=IMAX
        IF(J.NE.N)THEN
          IF(A(J,J).EQ.0.)A(J,J)=TINY
          DUM=1./A(J,J)
          DO 18 I=J+1,N
            A(I,J)=A(I,J)*DUM
18        CONTINUE
        ENDIF
19    CONTINUE
      IF(A(N,N).EQ.0.)A(N,N)=TINY
      RETURN
      END
C=====================================================================
      SUBROUTINE LUBKSB(A,N,NP,INDX,B)
C=====================================================================
C This subroutine is from Numerical Recipes.  It solves a set of
C simultaneous linear equations by back-substitution after an LU
C decomposition.  It is used here for performing a Newton-Raphson
C solution for nonlinear equations.
C
      DIMENSION A(NP,NP),INDX(N),B(N)
      II=0
      DO 12 I=1,N
        LL=INDX(I)
        SUM=B(LL)
        B(LL)=B(I)
        IF (II.NE.0)THEN
```

```
          DO 11 J=II,I-1
            SUM=SUM-A(I,J)*B(J)
11        CONTINUE
        ELSE IF (SUM.NE.0.) THEN
          II=I
        ENDIF
        B(I)=SUM
12    CONTINUE
      DO 14 I=N,1,-1
        SUM=B(I)
        IF(I.LT.N)THEN
          DO 13 J=I+1,N
            SUM=SUM-A(I,J)*B(J)
13        CONTINUE
        ENDIF
        B(I)=SUM/A(I,I)
14    CONTINUE
      RETURN
      END
C=======================================================================
      SUBROUTINE MNEWT(NTRIAL,X,N,TOLX,TOLF,Q)
C=======================================================================
C This subroutine is from Numerical Recipes, modified to take the
C additional argument Q needed by user function INITNR.

      PARAMETER (NP=10)
      DIMENSION X(*),ALPHA(NP,NP),BETA(NP),INDX(NP),Q(*)
      IF (N .GT. NP) THEN
        WRITE(*,*) ' Oops! Dimension NP in MNEWT is too small.  Change '
        WRITE(*,*) ' to ', N, ' and recompile.'
        PAUSE
        STOP
      END IF
      DO 13  K=1,NTRIAL
        CALL INITNR(X,ALPHA,BETA,Q)
        ERRF=0.
        DO 11 I=1,N
          ERRF=ERRF+ABS(BETA(I))
11      CONTINUE
        IF(ERRF.LE.TOLF)RETURN
        CALL LUDCMP(ALPHA,N,N,INDX,D)
        CALL LUBKSB(ALPHA,N,N,INDX,BETA)
        ERRX=0.
        DO 12 I=1,N
          ERRX=ERRX+ABS(BETA(I))
          X(I)=X(I)+BETA(I)
12      CONTINUE
        IF(ERRX.LE.TOLX)RETURN
13    CONTINUE
      RETURN
      END
C=======================================================================
      FUNCTION NORMA(A)
C=======================================================================
C  normalize angle
C
      REAL A, NORMA, PI
```

```
      PARAMETER (PI=3.141592653589793)
      IF (A .GE. PI) THEN
        NORMA = A - PI
      ELSE IF (A .LE. -PI) THEN
        NORMA = A + PI
      ELSE
        NORMA = A
      END IF
      RETURN
      END
C=====================================================================
      FUNCTION OPNFIL (PROMPT, STAT, IFILE)
C=====================================================================
C  This function tries to get a file name from the user and open the
C  file.
C
C  --> PROMPT string   Message to prompt user
C  --> STAT   string   Status of file ("NEW"    = mandatory output,
C                                      "OLD"    = mandatory input,
C                                      "OPTIN"  = optional input,
C                                      "OPTOUT" = optional output)
C  --> IFILE  integer  Fortran I/O unit for file
C  <-- OPNFIL string   name of file opened or " " if no file selected
C
C M. Sayers January 30, 1989
C
      LOGICAL       ISIT
      CHARACTER*(*) PROMPT, STAT, OPNFIL
      CHARACTER*3   STAT2
      INTEGER       IFILE, L, LENSTR
C
C  Set Fortran STATUS type
C
      IF (STAT .EQ. 'NEW' .OR. STAT .EQ. 'OPTOUT') THEN
        STAT2 = 'NEW'
      ELSE
        STAT2 = 'OLD'
      END IF
C
C  Ask user for file name, and check for no response (blank line)
C
  100 WRITE(*, '(A, A, A\)') ' ', PROMPT, ': '
      READ(*, '(A)') OPNFIL
      IF (OPNFIL .EQ. ' ') THEN
        IF (STAT .EQ. 'OPTIN' .OR. STAT .EQ. 'OPTOUT') THEN
          RETURN
        ELSE IF (STAT .EQ. 'NEW') THEN
          WRITE (*,'(A)') ' Output file is required!'
          GO TO 100
        ELSE
          WRITE (*,'(A)') ' Input file is required!'
          GO TO 100
        END IF
      END IF
C
C  Deal with existance of file (or lack thereof)
C
```

```
      INQUIRE (FILE=OPNFIL, EXIST=ISIT)
      IF ((.NOT. ISIT) .AND. (STAT2 .EQ. 'OLD')) THEN
        L = LENSTR(OPNFIL)
        WRITE (*, '(A, A, A)') ' File "', OPNFIL(:L),
     &               '" does not exist.  Try again.'
        GO TO 100
      ELSE IF (ISIT .AND. STAT2 .EQ. 'NEW') THEN
        OPEN (IFILE, FILE=OPNFIL)
        CLOSE (IFILE, STATUS='DELETE')
      END IF
C
C  Open file and write blank line on screen
C
      OPEN(IFILE, STATUS=STAT2, FILE=OPNFIL)
      WRITE (*,'(A)') ' '
      RETURN
      END
C=======================================================================
      SUBROUTINE OPNOUT(IFILE)
C=======================================================================
C  This subroutine prompts the user for the name of a file set that
C  will be created to store time histories of the 7 output variables
C  computed by the 4-bar linkage simulation program.
C
C  A text file is created and opened, and labeling information is
C  written to facilitate post-processing of the data.  Then, the text
C  file is closed and a corresponding binary file is created and opened
C  to store the numerical values of the output variables.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      CHARACTER*80 FNOUT, INFILE, OPNFIL, TITLE
      LOGICAL      ISIT
      REAL         BI33, BM, D, DEGREES, GEES, IPRINT, K, L1, L2, L3,
     &             L4, L5, L6, L7, PARS, STEP, STOPT, T
      INTEGER      IFILE, ILOOP, IPRNT2, LENSTR, LSTRNG, MAXBUF,
     &             NBYTES, NCHAN, NCOORD, NPARS, NRECS, NSAMP, NSCAN,
     &             NSPEED, NUMKEY, NVARS
      CHARACTER*32 GENNAM, LONGNM, RIGBOD
      CHARACTER*24 TIMEDT
      CHARACTER*8  CHAR8, SHORTN, UNITSN
C
      PARAMETER    (NPARS = 14)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE (PARS(13), STEP), (PARS(14), STOPT)
C
      PARAMETER    (NVARS = 7, NUMKEY = 1)
      DIMENSION    LONGNM(NVARS), GENNAM(NVARS), RIGBOD(NVARS),
     &             SHORTN(NVARS), UNITSN(NVARS)
C
C  Prompt user to provide name of output file.  File is opened and
C  attached to Fortran unit IFILE.
```

```
C
      FNOUT = OPNFIL('Name of (required) file for time history outputs',
     &    'NEW', IFILE)
      NCHAN = 0
      IPRNT2 = PARS(4)
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'strut force'
      SHORTN (NCHAN) = 'F'
      GENNAM (NCHAN) = 'Force'
      UNITSN (NCHAN) = 'N'
      RIGBOD (NCHAN) = 'B'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'X coordinate of B*'
      SHORTN (NCHAN) = 'B* X'
      GENNAM (NCHAN) = 'Translation'
      UNITSN (NCHAN) = 'm'
      RIGBOD (NCHAN) = 'B'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Y coordinate of B*'
      SHORTN (NCHAN) = 'B* Y'
      GENNAM (NCHAN) = 'Translation'
      UNITSN (NCHAN) = 'm'
      RIGBOD (NCHAN) = 'B'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Rot. of A rel. to N, axis #3'
      SHORTN (NCHAN) = 'Q(1)'
      GENNAM (NCHAN) = 'Rotation'
      UNITSN (NCHAN) = 'rad'
      RIGBOD (NCHAN) = 'A'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Rot. of B rel. to A, axis #3'
      SHORTN (NCHAN) = 'Q(2)'
      GENNAM (NCHAN) = 'Rotation'
      UNITSN (NCHAN) = 'rad'
      RIGBOD (NCHAN) = 'B'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'Rot. of C rel. to N, axis #3'
      SHORTN (NCHAN) = 'Q(3)'
      GENNAM (NCHAN) = 'Rotation'
      UNITSN (NCHAN) = 'rad'
      RIGBOD (NCHAN) = 'C'
C
      NCHAN = NCHAN + 1
      LONGNM (NCHAN) = 'angle of B rel. to N'
      SHORTN (NCHAN) = 'B-angle'
      GENNAM (NCHAN) = 'Rotation Angle'
      UNITSN (NCHAN) = 'rad'
      RIGBOD (NCHAN) = 'B'
C
C   Write Header Info for ERD file
C
```

```
C  Set parameters needed to write header for ERD format file
C  NUMKEY = 1 for 32-bit floating-point binary
C  NSAMP  = number of samples
C  NRECS  = number of "records" in output file
C  NBYTES = number of bytes/record
C
      NSAMP = STOPT / STEP / IPRNT2 + 1
      NBYTES = 4 * NCHAN
      NRECS = NSAMP
C
C  Write standard ERD file heading.
C
      WRITE(IFILE, '(A)') 'ERDFILEV2.00'
      WRITE(IFILE, 100) NCHAN, NSAMP, NRECS, NBYTES, NUMKEY, STEP*IPRNT2
      WRITE(IFILE, '(A,A)') 'TITLE   ', TITLE
      WRITE(IFILE, 110) 'SHORTNAM', (SHORTN(ILOOP), ILOOP=1, NCHAN)
      WRITE(IFILE, 120) 'LONGNAME', (LONGNM(ILOOP), ILOOP=1, NCHAN)
      WRITE(IFILE, 110) 'UNITSNAM', (UNITSN(ILOOP), ILOOP=1, NCHAN)
      WRITE(IFILE, 120) 'GENNAME ', (GENNAM(ILOOP), ILOOP=1, NCHAN)
      WRITE(IFILE, 120) 'RIGIBODY', (RIGBOD(ILOOP), ILOOP=1, NCHAN)
      WRITE(IFILE, '(A)') 'XLABEL  Time'
      WRITE(IFILE, '(A)') 'XUNITS  sec'
C
      IF (INFILE .EQ. ' ') THEN
        WRITE(IFILE, '(A)') 'HISTORY No input file (used defaults)'
      ELSE
        WRITE(IFILE, '(A, A)') 'HISTORY Input parameter file was ',
     &        INFILE
      END IF
      CALL TIMDAT(TIMEDT)
      WRITE(IFILE, '(A,A)')
     & 'HISTORY Data generated with 4-bar linkage at '
     &, TIMEDT
      WRITE(IFILE, '(A)') 'END'
C
C  Close (text) header and create binary file.
C
      CLOSE(IFILE)
      LSTRNG = LENSTR(FNOUT)
      FNOUT = FNOUT (:LSTRNG) // '.BIN'
      INQUIRE(FILE=FNOUT, EXIST=ISIT)
      IF (ISIT) THEN
        OPEN (IFILE, FILE=FNOUT)
        CLOSE (IFILE, STATUS='DELETE')
      END IF
C
      OPEN(IFILE, FILE=FNOUT, STATUS='NEW', ACCESS='SEQUENTIAL',
     &      FORM='UNFORMATTED')
C
  100 FORMAT (5(I6,','),G13.6)
  110 FORMAT (A8, 7A8)
  120 FORMAT (A8, 7A32)
      RETURN
      END
C=======================================================================
      SUBROUTINE OUTPUT(IFILE, T, Q, QP, U, UP)
C=======================================================================
```

```
C  --> IFILE integer Fortran i/o unit for output
C  --> T     real    time
C  --> Q     real    array of 3 generalized coordinates
C  --> QP    real    array of derivitives of Q
C  --> U     real    array of 1 generalized speed
C  --> UP    real    array of derivatives of U
C
C  This subroutine writes the values of the 7 output variables computed
C  by the 4-bar linkage simulation program into an output file, using
C  the values at time T.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      CHARACTER*80 INFILE, TITLE
      REAL         BI33, BM, C, D, DEGREES, FORCEM, GEES, IPRINT, K,
     &             L1, L2, L3, L4, L5, L6, L7, OUTBUF, PARS, PC, Q, QP,
     &             S, STEP, STOPT, T, U, UP, Z
      INTEGER      IFILE, ILOOP, NCOORD, NPARS, NSPEED, NVARS
C
      PARAMETER    (NCOORD = 3, NSPEED = 1, NVARS = 7)
      DIMENSION    Q(NCOORD), QP(NCOORD), U(NSPEED), UP(NSPEED),
     &             OUTBUF(NVARS)
      DIMENSION    PC(30)
      COMMON       /PRCMP/ PC
      SAVE         /PRCMP/
C
      DIMENSION    C(3), FORCEM(1), S(3), Z(71)
      COMMON       /DYVARS/ C, FORCEM, S, Z
      SAVE         /DYVARS/
C
      PARAMETER    (NPARS = 14)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE  (PARS(1), BI33), (PARS(2), BM), (PARS(3), D),
     &             (PARS(4), IPRINT), (PARS(5), K), (PARS(6), L1),
     &             (PARS(7), L2), (PARS(8), L3), (PARS(9), L4),
     &             (PARS(10), L5), (PARS(11), L6), (PARS(12), L7),
     &             (PARS(13), STEP), (PARS(14), STOPT)
      PARAMETER    (GEES = 9.80665)
      Z(67) = (Q(1) + Q(2))
      Z(68) = L3*Z(26)
      Z(69) = (Z(49) + Z(68))
      Z(70) = L3*Z(30)
      Z(71) = (Z(48) -Z(70))
C
C  fill buffer with output variables.
C
      OUTBUF(1) = -FORCEM(1)
      OUTBUF(2) = Z(71)
      OUTBUF(3) = Z(69)
      OUTBUF(4) = Q(1)
      OUTBUF(5) = Q(2)
      OUTBUF(6) = Q(3)
```

```
      OUTBUF(7) = Z(67)
C
C  The following line writes to an unformatted binary file
C
      WRITE (IFILE) (OUTBUF(ILOOP), ILOOP=1, NVARS)
C
C--The next 3 lines are for the Macintosh
C
      IF (T .EQ. 0.) WRITE (*, '(/A/7X,A)') ' Progress:','sec'
      CALL TOOLBX (Z'89409000', 0, -11)
      WRITE (*, '(F6.2)') T
      RETURN
      END
C=======================================================================
      SUBROUTINE PRECMP
C=======================================================================
C  This subroutine defines all constants that can be pre-computed for
C  the 4-bar linkage.  The constants are put into the COMMON block
C  /PRECMP/
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      IMPLICIT NONE
      CHARACTER*80 INFILE, TITLE
      REAL         BI33, BM, D, DEGREES, GEES, IPRINT, K, L1, L2, L3,
     &             L4, L5, L6, L7, PARS, PC, STEP, STOPT
      INTEGER      NPARS
C
      DIMENSION    PC(30)
      COMMON       /PRCMP/ PC
      SAVE         /PRCMP/
C
      PARAMETER    (NPARS = 14)
      DIMENSION    PARS(NPARS)
      COMMON       /INPARS/ PARS, TITLE, INFILE
      SAVE         /INPARS/
C
      EQUIVALENCE  (PARS(1), BI33), (PARS(2), BM), (PARS(3), D),
     &             (PARS(4), IPRINT), (PARS(5), K), (PARS(6), L1),
     &             (PARS(7), L2), (PARS(8), L3), (PARS(9), L4),
     &             (PARS(10), L5), (PARS(11), L6), (PARS(12), L7),
     &             (PARS(13), STEP), (PARS(14), STOPT)
      PARAMETER    (GEES = 9.80665)
C
      PC(1) = (L1 -L5)
      PC(2) = L1/L4
      PC(3) = SQRT((L1 -L6)**2 + (L2 -L7)**2)
      PC(4) = 1.0/L4
      PC(5) = D*L1
      PC(6) = (L2 -L3)
      PC(7) = L1*BI33/L4
      PC(8) = L1*L3/L4
      PC(9) = L3*PC(2)**2
      PC(10) = L6**2
      PC(11) = L2*L6
      PC(12) = L2**2
```

```
      PC(13) = L2*L7
      PC(14) = L1**2
      PC(15) = L1*L6
      PC(16) = L1*L7
      PC(17) = L1*L2
      PC(18) = L7**2
      PC(19) = 2.0*PC(13)
      PC(20) = 2.0*PC(11)
      PC(21) = (PC(10) + PC(12) + PC(14) + PC(18))
      PC(22) = 2.0*PC(1)
      PC(23) = PC(2)*PC(7)
      PC(24) = PC(2)*PC(6)
      PC(25) = K*PC(3)
      PC(26) = L2*PC(4)
      PC(27) = PC(5)*PC(26)
      PC(28) = L7*PC(5)
      PC(29) = L2*PC(5)
      PC(30) = L6*PC(5)
      RETURN
      END
*****************************************************************
      SUBROUTINE TIMDAT (TIMEDT)
*****************************************************************
C  Get date and time.  On the Mac, this requires the TIME and DATE
C  subroutines from Absoft.
C
C  by M. Sayers, 1986.
C
C  <-- TIMEDT char*24   string containing time & date.
C
      CHARACTER*24 TIMEDT
      CHARACTER*36 MONTHS
      INTEGER*4 M, IDAY, IYEAR, ISEC
      INTEGER*2 YEAR, MONTH, DAY, HOUR, MIN, SEC, I100
      MONTHS = 'JanFebMarAprMayJunJulAugSepOctNovDec'

C--The following 4 lines are for the IBM PC (using Microsoft
C--time and date functions)
*      CALL GETDAT (YEAR, MONTH, DAY)
*      CALL GETTIM (IHOUR, MIN, SEC, I100)
*      WRITE (TIMEDT, 100) IHOUR, MIN, MONTHS (MONTH*3-2:MONTH*3),
*     &       DAY, YEAR

C--get time for MTS version
C      CALL TIME(22, 0, TIMEDT)

C--The following 5 lines are for the Apple Mac
C--(using Absoft time & date functions)
      CALL DATE (M, IDAY, IYEAR)
      CALL TIME (ISEC)
      WRITE (TIMEDT, 100)
     &       ISEC/3600, MOD (ISEC, 3600) / 60, MONTHS (M*3-2:M*3),
     &       IDAY, 1900 + IYEAR

  100 FORMAT (I2,':',I2.2,' on ',A3,I3,',',I5)
      RETURN
      END
```

APPENDIX D — SPACECRAFT #1 EQUATIONS

This appendix contains the equations of motion for the spacecraft described in Section 9.4. These are the full, nonlinear equations, extracted from the subroutines DIFEQN and PRECMP. Definitions of state variables and parameters are found in Section 9.4.

The computations performed "in the loop" are the following:

```
C
C  Each derivative evaluation requires 773 multiply/divides, 628
C  add/subtracts, and 18 function/subroutine calls.
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
      S(6) = SIN(Q(6))
      S(5) = SIN(Q(5))
      S(4) = SIN(Q(4))
      S(7) = SIN(Q(7))
      S(8) = SIN(Q(8))
      S(9) = SIN(Q(9))
      S(10) = SIN(Q(10))
      C(6) = COS(Q(6))
      C(5) = COS(Q(5))
      C(4) = COS(Q(4))
      C(7) = COS(Q(7))
      C(8) = COS(Q(8))
      C(9) = COS(Q(9))
      C(10) = COS(Q(10))
C
C
C  Kinematical equations
C
      Z(1) = (PC(2)*U(5) + U(1))
      Z(2) = (PC(2)*U(4) -U(2))
      QP(1) = (C(5)*(Z(1)*C(6) + Z(2)*S(6)) + U(3)*S(5))
      Z(3) = S(4)*S(5)
      QP(2) = -(Z(2)*(C(6)*C(4) -Z(3)*S(6)) -Z(1)*(Z(3)*C(6) +
     &         C(4)*S(6)) + U(3)*C(5)*S(4))
      QP(3) = (U(3)*C(5)*C(4) + Z(1)*(-C(4)*C(6)*S(5) + S(6)*S(4))
     &         -Z(2)*(C(4)*S(6)*S(5) + C(6)*S(4)))
      QP(4) = (U(4)*C(6) -U(5)*S(6))/C(5)
      QP(5) = (U(5)*C(6) + U(4)*S(6))
      QP(6) = (U(6) -QP(4)*S(5))
      QP(7) = U(7)
      QP(8) = U(8)
      QP(9) = U(9)
      QP(10) = U(10)
```

```
C
C  External subroutines and extra variables
C
      CALL CMD(T, CLKCMD, CAMCMD)
C
C  define expression for boom-torque Z
C
      FORCEM(1) = (KB*Q(9) + BB*U(9))
C
C  define expression for boom-torque X
C
      FORCEM(2) = (KB*Q(10) + BB*U(10))
C
C  define expression for torque from clock motor
C
      FORCEM(3) = (KCLOCK*(CLKCMD -Q(7)) -BCLOCK*U(7))
C
C  define expression for torque from camera motor
C
      FORCEM(4) = (KCLOCK*(CAMCMD -Q(8)) -BCLOCK*U(8))
C
C  define expression for thruster torque #1
C
      FORCEM(5) = LTT1*THRUST(T, 1, (Q(4) + GYRO*U(4)))
C
C  define expression for thruster torque #2
C
      FORCEM(6) = LTT2*THRUST(T, 2, (Q(5) + GYRO*U(5)))
C
C  define expression for thruster torque #3
C
      FORCEM(7) = LTT3*THRUST(T, 3, (Q(6) + GYRO*U(6)))
C
C  Dynamical equations
C
      Z(4) = (U(6) + U(7))
      Z(5) = (U(4)*C(7) + U(5)*S(7))
      Z(6) = (U(5)*C(7) -U(4)*S(7))
      Z(7) = (Z(6)*C(8) -Z(4)*S(8))
      Z(8) = (Z(4)*C(8) + Z(6)*S(8))
      Z(9) = S(7)*S(8)
      Z(10) = C(8)*S(7)
      Z(11) = C(7)*S(8)
      Z(12) = C(7)*C(8)
      Z(13) = (U(8) -Z(5))
      Z(14) = (U(6) + U(9))
      Z(15) = (U(5)*C(9) -U(4)*S(9))
      Z(16) = (Z(14)*C(10) -Z(15)*S(10))
      Z(17) = (Z(15)*C(10) + Z(14)*S(10))
      Z(18) = S(9)*S(10)
      Z(19) = C(10)*S(9)
      Z(20) = C(9)*S(10)
      Z(21) = C(9)*C(10)
      Z(22) = (U(10) + U(4)*C(9) + U(5)*S(9))
      Z(23) = PC(3)*S(7)
      Z(24) = PC(3)*C(7)
      Z(25) = (L5 -L3*C(8))
```

```
 Z(26) = (Z(25)*C(7) + Z(24)*S(8))
 Z(27) = (L6 + L3*S(8))
 Z(28) = (Z(27)*C(7) + Z(24)*C(8))
 Z(29) = (L5*Z(9) + L6*Z(10) + Z(23))
 Z(30) = (Z(25)*S(7) + Z(23)*S(8))
 Z(31) = (Z(23)*C(8) + Z(27)*S(7))
 Z(32) = (L5*Z(11) + L6*Z(12) + Z(24))
 Z(33) = (L3 -L5*C(8) + L6*S(8))
 Z(34) = PC(2)*S(9)
 Z(35) = PC(2)*C(9)
 Z(36) = L7*S(9)
 Z(37) = (Z(35)*C(10) + L7*S(10))
 Z(38) = (L8*C(9) + L7*C(10) -Z(35)*S(10))
 Z(39) = (L8*Z(18) -Z(34))
 Z(40) = (L8*S(9) -Z(34)*S(10))
 Z(41) = Z(34)*C(10)
 Z(42) = (L8*Z(20) -Z(35))
 Z(43) = Z(36)*S(10)
 Z(44) = Z(36)*C(10)
 Z(45) = L8*C(10)
 Z(46) = (Z(45) + L7*C(9))
 Z(47) = U(4)*U(7)
 Z(48) = U(5)*U(7)
 Z(49) = Z(48)*C(7)
 Z(50) = Z(47)*S(7)
 Z(51) = (Z(49) -Z(50))
 Z(52) = (Z(47)*C(7) + Z(48)*S(7))
 Z(53) = (U(8)*Z(8) + Z(52)*C(8))
 Z(54) = -(U(8)*Z(7) -Z(52)*S(8))
 Z(55) = U(4)*U(9)
 Z(56) = U(5)*U(9)
 Z(57) = (Z(56)*C(9) -Z(55)*S(9))
 Z(58) = (Z(55)*C(9) + Z(56)*S(9))
 Z(59) = -(U(10)*Z(17) -Z(58)*S(10))
 Z(60) = U(6)*U(4)
 Z(61) = U(5)*U(6)
 Z(62) = U(4)**2
 Z(63) = U(5)**2
 Z(64) = (Z(62) + Z(63))
 Z(65) = -(U(6)*U(2) -U(5)*U(3))
 Z(66) = -(U(4)*U(3) -U(6)*U(1))
 Z(67) = -(U(5)*U(1) -U(4)*U(2))
 Z(68) = (PC(3)*Z(60) -Z(65))
 Z(69) = (PC(3)*Z(61) -Z(66))
 Z(70) = Z(7)*Z(8)
 Z(71) = Z(13)**2
 Z(72) = -(-L3*(Z(4)*Z(6) + Z(49) -Z(50)) + PC(3)*Z(64) + Z(67))
 Z(73) = (L3*(Z(4)**2 + Z(5)**2) -Z(69)*C(7) + Z(68)*S(7))
 Z(74) = (L5*(Z(51) + Z(70)) + L6*(Z(71) + Z(7)**2) -Z(72)*C(8) +
&         Z(73)*S(8))
 Z(75) = -(L6*(Z(51) -Z(70)) -L5*(Z(71) + Z(8)**2) + Z(73)*C(8) +
&         Z(72)*S(8))
 Z(76) = (L3*Z(5)*Z(6) -L6*(Z(8)*Z(13) + Z(53)) + L5*(Z(7)*Z(13)
&         -Z(54)) + Z(68)*C(7) + Z(69)*S(7))
 Z(77) = (-L7*U(5)*U(4) + PC(2)*Z(60) + Z(65))
 Z(78) = (PC(2)*Z(61) + L7*(Z(62) + U(6)**2) + Z(66))
 Z(79) = (Z(78)*C(9) -Z(77)*S(9))
```

```
 Z(80) = (L7*Z(61) + PC(2)*Z(64) -Z(67))
 Z(81) = Z(17)*Z(22)
 Z(82) = Z(16)*Z(17)
 Z(83) = (L8*(Z(16)**2 + Z(22)**2) + Z(79)*C(10) -Z(80)*S(10))
 Z(84) = (L8*(Z(57) + Z(82)) + Z(80)*C(10) + Z(79)*S(10))
 Z(85) = (L8*(Z(59) -Z(81)) + Z(77)*C(9) + Z(78)*S(9))
 Z(86) = BI13*U(5)
 Z(87) = BI12*U(5)
 Z(88) = DI23*Z(7)
 Z(89) = DI13*Z(13)
 Z(90) = DI12*Z(7)
 Z(91) = DI13*Z(8)
 Z(92) = (DI12*Z(51) + DI22*Z(53) -DI23*Z(54) + Z(13)*(Z(88) +
&         Z(89)) + Z(8)*(PC(7)*Z(13) + Z(90) -Z(91)))
 Z(93) = DI23*Z(8)
 Z(94) = DI12*Z(13)
 Z(95) = (-DI11*Z(51) -DI12*Z(53) + DI13*Z(54) + Z(7)*(-DI33*Z(8)
&         + Z(88) + Z(89)) + Z(8)*(DI22*Z(7) -Z(93) + Z(94)))
 Z(96) = (DI13*Z(51) + DI23*Z(53) -DI33*Z(54) + Z(7)*(PC(8)*Z(13)
&         + Z(90) -Z(91)) + Z(13)*(Z(93) -Z(94)))
 Z(97) = DM*Z(29)
 Z(98) = DM*Z(28)
 Z(99) = DM*Z(26)
 Z(100) = DI13*Z(9)
 Z(101) = DI12*Z(10)
 Z(102) = DI11*C(7)
 Z(103) = DM*Z(32)
 Z(104) = DM*Z(31)
 Z(105) = DM*Z(30)
 Z(106) = (DI23*Z(11) -DI22*Z(12) + DI12*S(7))
 Z(107) = (DI33*Z(11) -DI23*Z(12) + DI13*S(7))
 Z(108) = DI13*Z(11)
 Z(109) = DI12*Z(12)
 Z(110) = DI11*S(7)
 Z(111) = (Z(108) -Z(109) + Z(110))
 Z(112) = DM*Z(33)
 Z(113) = (DI33*C(8) + DI23*S(8))
 Z(114) = (DI23*C(8) + DI22*S(8))
 Z(115) = (DI13*C(8) + DI12*S(8))
 Z(116) = FI2*(U(10)*Z(16) -Z(58)*C(10))
 Z(117) = (FI1*Z(59) -PC(11)*Z(81))
 Z(118) = (FI1*Z(57) + PC(11)*Z(82))
 Z(119) = FM*Z(39)
 Z(120) = FM*Z(38)
 Z(121) = FM*Z(37)
 Z(122) = FI1*C(9)
 Z(123) = FM*Z(42)
 Z(124) = FM*Z(41)
 Z(125) = FM*Z(40)
 Z(126) = FI1*S(9)
 Z(127) = FI2*Z(21)
 Z(128) = FI1*Z(20)
 Z(129) = FM*Z(46)
 Z(130) = FM*Z(44)
 Z(131) = FM*Z(43)
 Z(132) = FI2*S(10)
 Z(133) = FI1*C(10)
```

```
 Z(134) = FM*Z(45)
 Z(135) = -(-PC(1)*Z(65) + DM*(Z(9)*Z(74) -Z(10)*Z(75) +
&          Z(76)*C(7)) + FM*(Z(19)*Z(83) + Z(18)*Z(84)
&          -Z(85)*C(9)))
 Z(136) = (PC(1)*Z(66) -DM*(-Z(11)*Z(74) + Z(12)*Z(75) +
&          Z(76)*S(7)) + FM*(Z(21)*Z(83) + Z(20)*Z(84) +
&          Z(85)*S(9)))
 Z(137) = (PC(1)*Z(67) + DM*(Z(74)*C(8) + Z(75)*S(8)) -FM
&         *(Z(84)*C(10) -Z(83)*S(10)))
 Z(138) = ((PC(4)*U(5) + BI23*U(6))*U(6) + CI*(U(5)*Z(4) -Z(48))
&          -BI23*Z(63) + U(4)*(BI12*U(6) -Z(86)) -Z(10)*Z(92) +
&          Z(9)*Z(96) + Z(76)*Z(97) + Z(75)*Z(98) -Z(74)*Z(99) +
&          Z(19)*Z(116) -Z(18)*Z(117) -Z(85)*Z(119) -Z(84)*Z(120)
&          + Z(83)*Z(121) + FORCEM(5) + Z(95)*C(7) -Z(118)*C(9)
&          -FORCEM(1)*(Z(18)*C(10) -Z(19)*S(10)))
 Z(139) = -(-Z(10)*Z(98) -Z(9)*Z(99) -Z(18)*Z(120) + Z(19)*Z(121)
&          + Z(97)*C(7) + Z(119)*C(9))
 Z(140) = (Z(12)*Z(98) + Z(11)*Z(99) + Z(20)*Z(120) -Z(21)*Z(121)
&          + Z(97)*S(7) + Z(119)*S(9))
 Z(141) = (-Z(99)*C(8) + Z(120)*C(10) + Z(98)*S(8) + Z(121)*S(10))
 Z(142) = ((BI23*U(5) + BI13*U(4) + BI33*U(6))*U(4) -CI*(U(4)*Z(4)
&          -Z(47)) -U(6)*(PC(5)*U(4) + BI13*U(6) + Z(87)) +
&          Z(12)*Z(92) -Z(11)*Z(96) -Z(76)*Z(103) + Z(75)*Z(104)
&          -Z(74)*Z(105) -Z(21)*Z(116) + Z(20)*Z(117) +
&          Z(85)*Z(123) + Z(83)*Z(124) -Z(84)*Z(125) + FORCEM(6) +
&          Z(95)*S(7) -Z(118)*S(9) + FORCEM(1)*(Z(20)*C(10)
&          -Z(21)*S(10)))
 Z(143) = (Z(10)*Z(104) + Z(9)*Z(105) -Z(19)*Z(124) + Z(18)*Z(125)
&          + Z(103)*C(7) + Z(123)*C(9))
 Z(144) = -(-Z(12)*Z(104) -Z(11)*Z(105) + Z(21)*Z(124)
&          -Z(20)*Z(125) + Z(103)*S(7) + Z(123)*S(9))
 Z(145) = (-Z(105)*C(8) + Z(125)*C(10) + Z(104)*S(8) +
&          Z(124)*S(10))
 Z(146) = (BI12 -Z(29)*Z(103) + Z(28)*Z(104) + Z(26)*Z(105) +
&          Z(10)*Z(106) -Z(9)*Z(107) -Z(39)*Z(123) + Z(37)*Z(124)
&          + Z(38)*Z(125) -Z(19)*Z(127) -Z(18)*Z(128) +
&          Z(111)*C(7) + Z(126)*C(9))
 Z(147) = Z(76)*Z(112)
 Z(148) = Z(96)*C(8)
 Z(149) = Z(117)*C(10)
 Z(150) = Z(92)*S(8)
 Z(151) = Z(116)*S(10)
 Z(152) = Z(112)*C(7)
 Z(153) = (Z(19)*Z(130) + Z(18)*Z(131) + Z(152) + Z(129)*C(9))
 Z(154) = Z(112)*S(7)
 Z(155) = (-Z(21)*Z(130) -Z(20)*Z(131) + Z(154) + Z(129)*S(9))
 Z(156) = (Z(131)*C(10) -Z(130)*S(10))
 Z(157) = Z(29)*Z(112)
 Z(158) = Z(9)*Z(113)
 Z(159) = Z(10)*Z(114)
 Z(160) = Z(19)*Z(132)
 Z(161) = Z(18)*Z(133)
 Z(162) = Z(115)*C(7)
 Z(163) = (BI13 + Z(39)*Z(129) + Z(37)*Z(130) -Z(38)*Z(131) +
&          Z(157) -Z(158) + Z(159) -Z(160) + Z(161) + Z(162))
 Z(164) = Z(32)*Z(112)
 Z(165) = Z(11)*Z(113)
```

```
 Z(166) = Z(12)*Z(114)
 Z(167) = Z(21)*Z(132)
 Z(168) = Z(20)*Z(133)
 Z(169) = Z(115)*S(7)
 Z(170) = (BI23 -Z(42)*Z(129) + Z(41)*Z(130) -Z(40)*Z(131) -Z(164)
&          + Z(165) -Z(166) + Z(167) -Z(168) + Z(169))
 Z(171) = Z(33)*Z(112)
 Z(172) = Z(113)*C(8)
 Z(173) = Z(133)*C(10)
 Z(174) = Z(114)*S(8)
 Z(175) = Z(132)*S(10)
 Z(176) = (Z(147) -Z(148) -Z(150) + FORCEM(3))
 Z(177) = -(Z(157) -Z(158) + Z(159) + Z(162))
 Z(178) = -(-Z(164) + Z(165) -Z(166) + Z(169))
 Z(179) = (Z(171) + Z(172) + Z(174))
 Z(180) = (PC(9)*Z(74) -PC(10)*Z(75) -Z(95) + FORCEM(4))
 Z(181) = (PC(9)*Z(9) + PC(10)*Z(10))
 Z(182) = (PC(9)*Z(11) + PC(10)*Z(12))
 Z(183) = -(PC(9)*C(8) -PC(10)*S(8))
 Z(184) = -(PC(9)*Z(26) + PC(10)*Z(28) -Z(100) + Z(101) + Z(102))
 Z(185) = -(PC(9)*Z(30) + PC(10)*Z(31) + Z(108) -Z(109) + Z(110))
 Z(186) = (Z(85)*Z(134) + Z(149) + Z(151) + FORCEM(1))
 Z(187) = Z(134)*C(9)
 Z(188) = Z(134)*S(9)
 Z(189) = (Z(39)*Z(134) -Z(160) + Z(161))
 Z(190) = (Z(42)*Z(134) -Z(167) + Z(168))
 Z(191) = (Z(46)*Z(134) + Z(173) + Z(175))
 Z(192) = (Z(45)*Z(134) + Z(173) + Z(175))
 Z(193) = (PC(12)*Z(84) + Z(118) + FORCEM(2))
 Z(194) = PC(12)*Z(18)
 Z(195) = PC(12)*Z(20)
 Z(196) = PC(12)*C(10)
 Z(197) = (PC(12)*Z(38) + Z(122))
 Z(198) = (PC(12)*Z(40) + Z(126))
 Z(199) = PC(12)*Z(43)
 Z(200) = PC(18)*Z(196)
 Z(201) = PC(18)*Z(183)
 Z(202) = PC(18)*Z(141)
 Z(203) = PC(18)*Z(145)
 Z(204) = PC(18)*Z(156)
 Z(205) = Z(187)/Z(192)
 Z(206) = Z(188)/Z(192)
 Z(207) = Z(189)/Z(192)
 Z(208) = Z(190)/Z(192)
 Z(209) = Z(191)/Z(192)
 Z(210) = Z(152)/Z(179)
 Z(211) = Z(154)/Z(179)
 Z(212) = Z(115)/Z(179)
 Z(213) = Z(177)/Z(179)
 Z(214) = Z(178)/Z(179)
 Z(215) = (PC(17) -Z(196)*Z(200))
 Z(216) = Z(194)/Z(215)
 Z(217) = Z(195)/Z(215)
 Z(218) = Z(196)*Z(201)/Z(215)
 Z(219) = (Z(197) -Z(196)*Z(202))/Z(215)
 Z(220) = (Z(198) -Z(196)*Z(203))/Z(215)
 Z(221) = (Z(199) -Z(196)*Z(204))/Z(215)
```

```
 Z(222) = (PC(13) -Z(187)*Z(205) -Z(152)*Z(210) -Z(194)*Z(216))
 Z(223) = -(Z(187)*Z(206) + Z(152)*Z(211) -Z(194)*Z(217))/Z(222)
 Z(224) = (Z(181) + Z(152)*Z(212) + Z(194)*Z(218))/Z(222)
 Z(225) = (-Z(139) -Z(187)*Z(207) + Z(152)*Z(213) +
&          Z(194)*Z(219))/Z(222)
 Z(226) = (Z(143) -Z(187)*Z(208) -Z(152)*Z(214)
&          -Z(194)*Z(220))/Z(222)
 Z(227) = (-Z(152) + Z(153) -Z(187)*Z(209) -Z(194)*Z(221))/Z(222)
 Z(228) = -(Z(188)*Z(205) + Z(154)*Z(210) -Z(195)*Z(216))
 Z(229) = (PC(13) -Z(188)*Z(206) -Z(154)*Z(211) -Z(195)*Z(217)
&          -Z(223)*Z(228))
 Z(230) = (Z(182) -Z(154)*Z(212) + Z(195)*Z(218) +
&          Z(224)*Z(228))/Z(229)
 Z(231) = (Z(140) -Z(188)*Z(207) + Z(154)*Z(213) -Z(195)*Z(219)
&          -Z(225)*Z(228))/Z(229)
 Z(232) = (-Z(144) -Z(188)*Z(208) -Z(154)*Z(214) + Z(195)*Z(220)
&          -Z(226)*Z(228))/Z(229)
 Z(233) = (-Z(154) + Z(155) -Z(188)*Z(209) + Z(195)*Z(221)
&          -Z(227)*Z(228))/Z(229)
 Z(234) = Z(183)*Z(200)
 Z(235) = (Z(181) + Z(115)*Z(210) + Z(216)*Z(234))
 Z(236) = (Z(182) -Z(115)*Z(211) + Z(217)*Z(234) + Z(223)*Z(235))
 Z(237) = (PC(16) -Z(183)*Z(201) -Z(115)*Z(212) -Z(218)*Z(234)
&          -Z(224)*Z(235) -Z(230)*Z(236))
 Z(238) = (-Z(184) -Z(183)*Z(202) + Z(115)*Z(213) + Z(219)*Z(234)
&          + Z(225)*Z(235) -Z(231)*Z(236))/Z(237)
 Z(239) = (-Z(185) -Z(183)*Z(203) + Z(115)*Z(214) + Z(220)*Z(234)
&          -Z(226)*Z(235) + Z(232)*Z(236))/Z(237)
 Z(240) = (Z(183)*Z(204) -Z(221)*Z(234) + Z(227)*Z(235)
&          -Z(233)*Z(236))/Z(237)
 Z(241) = (Z(197) -Z(141)*Z(200))
 Z(242) = (-Z(139) -Z(189)*Z(205) + Z(177)*Z(210) + Z(216)*Z(241))
 Z(243) = (Z(140) -Z(189)*Z(206) + Z(177)*Z(211) -Z(217)*Z(241)
&          -Z(223)*Z(242))
 Z(244) = (-Z(184) -Z(141)*Z(201) + Z(177)*Z(212) + Z(218)*Z(241)
&          + Z(224)*Z(242) -Z(230)*Z(243))
 Z(245) = (PC(14) + FI1*Z(18)**2 + FI2*Z(19)**2 + Z(29)*Z(97) +
&          Z(28)*Z(98) + Z(26)*Z(99) + Z(39)*Z(119) + Z(38)*Z(120)
&          + Z(37)*Z(121) -Z(141)*Z(202) -Z(189)*Z(207)
&          -Z(177)*Z(213) -Z(219)*Z(241) -Z(225)*Z(242)
&          -Z(231)*Z(243) -Z(238)*Z(244) + (-Z(100) + Z(101) +
&          Z(102))*C(7) -Z(9)*(-DI33*Z(9) + DI23*Z(10) +
&          DI13*C(7)) + Z(10)*(-DI23*Z(9) + DI22*Z(10) +
&          DI12*C(7)) + Z(122)*C(9))
 Z(246) = (Z(146) -Z(141)*Z(203) + Z(189)*Z(208) -Z(177)*Z(214)
&          -Z(220)*Z(241) + Z(226)*Z(242) + Z(232)*Z(243)
&          -Z(239)*Z(244))/Z(245)
 Z(247) = (Z(163) + Z(177) + Z(141)*Z(204) -Z(189)*Z(209) +
&          Z(221)*Z(241) -Z(227)*Z(242) -Z(233)*Z(243)
&          -Z(240)*Z(244))/Z(245)
 Z(248) = (Z(198) -Z(145)*Z(200))
 Z(249) = (Z(143) -Z(190)*Z(205) -Z(178)*Z(210) -Z(216)*Z(248))
 Z(250) = (-Z(144) -Z(190)*Z(206) -Z(178)*Z(211) + Z(217)*Z(248)
&          -Z(223)*Z(249))
 Z(251) = (-Z(185) -Z(145)*Z(201) + Z(178)*Z(212) + Z(218)*Z(248)
&          -Z(224)*Z(249) + Z(230)*Z(250))
 Z(252) = (Z(146) -Z(145)*Z(202) + Z(190)*Z(207) -Z(178)*Z(213)
```

```
&          -Z(219)*Z(248) + Z(225)*Z(249) + Z(231)*Z(250)
&          -Z(238)*Z(251))
 Z(253) = (PC(15) + Z(32)*Z(103) + Z(31)*Z(104) + Z(30)*Z(105)
&          -Z(12)*Z(106) + Z(11)*Z(107) + Z(42)*Z(123) +
&          Z(41)*Z(124) + Z(40)*Z(125) + Z(21)*Z(127) +
&          Z(20)*Z(128) -Z(145)*Z(203) -Z(190)*Z(208)
&          -Z(178)*Z(214) -Z(220)*Z(248) -Z(226)*Z(249)
&          -Z(232)*Z(250) -Z(239)*Z(251) -Z(246)*Z(252) +
&          Z(111)*S(7) + Z(126)*S(9))
 Z(254) = (Z(170) + Z(178) + Z(145)*Z(204) + Z(190)*Z(209) +
&          Z(221)*Z(248) + Z(227)*Z(249) + Z(233)*Z(250)
&          -Z(240)*Z(251) -Z(247)*Z(252))/Z(253)
 Z(255) = (Z(199) -Z(156)*Z(200))
 Z(256) = (Z(153) -Z(191)*Z(205) -Z(179)*Z(210) -Z(216)*Z(255))
 Z(257) = (Z(155) -Z(191)*Z(206) -Z(179)*Z(211) + Z(217)*Z(255)
&          -Z(223)*Z(256))
 Z(258) = (Z(115) + Z(156)*Z(201) -Z(179)*Z(212) -Z(218)*Z(255) +
&          Z(224)*Z(256) -Z(230)*Z(257))
 Z(259) = (Z(163) + Z(156)*Z(202) -Z(191)*Z(207) + Z(179)*Z(213) +
&          Z(219)*Z(255) -Z(225)*Z(256) -Z(231)*Z(257)
&          -Z(238)*Z(258))
 Z(260) = (Z(170) + Z(156)*Z(203) + Z(191)*Z(208) + Z(179)*Z(214)
&          + Z(220)*Z(255) + Z(226)*Z(256) + Z(232)*Z(257)
&          -Z(239)*Z(258) -Z(246)*Z(259))
 Z(261) = Z(137)*Z(200)
 Z(262) = (Z(193) + Z(261))
 Z(263) = Z(186)*Z(205)
 Z(264) = Z(176)*Z(210)
 Z(265) = Z(216)*Z(262)
 Z(266) = (Z(135) -Z(263) + Z(264) + Z(265))
 Z(267) = Z(186)*Z(206)
 Z(268) = Z(176)*Z(211)
 Z(269) = Z(217)*Z(262)
 Z(270) = Z(223)*Z(266)
 Z(271) = (Z(136) -Z(267) + Z(268) -Z(269) -Z(270))
 Z(272) = Z(137)*Z(201)
 Z(273) = Z(176)*Z(212)
 Z(274) = Z(218)*Z(262)
 Z(275) = Z(224)*Z(266)
 Z(276) = Z(230)*Z(271)
 Z(277) = (Z(180) + Z(272) + Z(273) + Z(274) + Z(275) -Z(276))
 Z(278) = Z(137)*Z(202)
 Z(279) = Z(186)*Z(207)
 Z(280) = Z(176)*Z(213)
 Z(281) = Z(219)*Z(262)
 Z(282) = Z(225)*Z(266)
 Z(283) = Z(231)*Z(271)
 Z(284) = Z(238)*Z(277)
 Z(285) = (Z(138) -Z(278) + Z(279) + Z(280) + Z(281) + Z(282) +
&          Z(283) + Z(284))
 Z(286) = Z(137)*Z(203)
 Z(287) = Z(186)*Z(208)
 Z(288) = Z(176)*Z(214)
 Z(289) = Z(220)*Z(262)
 Z(290) = Z(226)*Z(266)
 Z(291) = Z(232)*Z(271)
 Z(292) = Z(239)*Z(277)
```

```
 Z(293) = Z(246)*Z(285)
 Z(294) = (-BI12*Z(62) -U(6)*(BI23*U(4) -Z(86)) + U(5)*(PC(6)*U(4)
&          + Z(87)) -Z(85)*Z(129) + Z(83)*Z(130) + Z(84)*Z(131) +
&          Z(147) -Z(148) -Z(149) -Z(150) -Z(151) -Z(176) +
&          Z(137)*Z(204) + Z(186)*Z(209) -Z(221)*Z(262) +
&          Z(227)*Z(266) + Z(233)*Z(271) + Z(240)*Z(277)
&          -Z(247)*Z(285) -Z(254)*(Z(142) -Z(286) -Z(287) + Z(288)
&          + Z(289) -Z(290) -Z(291) + Z(292) -Z(293)) +
&          FORCEM(7))/(BI33 + Z(46)*Z(129) + Z(44)*Z(130) +
&          Z(43)*Z(131) + Z(171) + Z(172) + Z(173) + Z(174) +
&          Z(175) -Z(179) -Z(156)*Z(204) -Z(191)*Z(209)
&          -Z(221)*Z(255) -Z(227)*Z(256) -Z(233)*Z(257)
&          -Z(240)*Z(258) -Z(247)*Z(259) -Z(254)*Z(260))
 Z(295) = (Z(142) -Z(286) -Z(287) + Z(288) + Z(289) -Z(290)
&          -Z(291) + Z(292) -Z(293) -Z(260)*Z(294))/Z(253)
 Z(296) = (Z(138) -Z(278) + Z(279) + Z(280) + Z(281) + Z(282) +
&          Z(283) + Z(284) -Z(259)*Z(294) -Z(252)*Z(295))/Z(245)
 Z(297) = (Z(180) + Z(272) + Z(273) + Z(274) + Z(275) -Z(276) +
&          Z(258)*Z(294) + Z(251)*Z(295) + Z(244)*Z(296))/Z(237)
 Z(298) = (-Z(136) + Z(267) -Z(268) + Z(269) + Z(270)
&          -Z(257)*Z(294) + Z(250)*Z(295) -Z(243)*Z(296) +
&          Z(236)*Z(297))/Z(229)
 Z(299) = -(Z(135) -Z(263) + Z(264) + Z(265) + Z(256)*Z(294)
&          -Z(249)*Z(295) + Z(242)*Z(296) + Z(235)*Z(297) +
&          Z(228)*Z(298))/Z(222)
 Z(300) = (Z(193) + Z(261) -Z(255)*Z(294) + Z(248)*Z(295) +
&          Z(241)*Z(296) + Z(234)*Z(297) + Z(195)*Z(298)
&          -Z(194)*Z(299))/Z(215)
 UP(6) = Z(294)
 UP(5) = Z(295)
 UP(4) = Z(296)
 UP(8) = Z(297)
 UP(2) = Z(298)
 UP(1) = Z(299)
 UP(10) = -Z(300)
 UP(7) = (Z(176) -Z(179)*Z(294) + Z(178)*Z(295) + Z(177)*Z(296) +
&         Z(115)*Z(297) -Z(154)*Z(298) -Z(152)*Z(299))/Z(179)
 UP(9) = (-Z(186) -Z(191)*Z(294) + Z(190)*Z(295) -Z(189)*Z(296)
&         -Z(188)*Z(298) -Z(187)*Z(299))/Z(192)
 UP(3) = -PC(18)*(Z(137) + Z(156)*Z(294) -Z(145)*Z(295)
&         -Z(141)*Z(296) + Z(183)*Z(297) + Z(196)*Z(300))
```

The above equations refer to precomputed constants, defined as follows:

```
 PC(1) = (BM + CM)
 PC(2) = (L1 -L2)*CM/(BM + CM)
 PC(3) = (L1 -(L1 -L2)*CM/(BM + CM))
 PC(4) = (CM*(L1 -L2 -(L1 -L2)*CM/(BM + CM))**2 + BM*((L1
&         -L2)*CM)**2/(BM + CM)**2 + BI22 -BI33)
 PC(5) = (CM*(L1 -L2 -(L1 -L2)*CM/(BM + CM))**2 + BM*((L1
&         -L2)*CM)**2/(BM + CM)**2 + BI11)
 PC(6) = (BI11 -BI22)
 PC(7) = (DI11 -DI33)
 PC(8) = (DI11 -DI22)
 PC(9) = L5*DM
 PC(10) = L6*DM
 PC(11) = (FI1 -FI2)
 PC(12) = L8*FM
 PC(13) = (BM + CM + DM + FM)
 PC(14) = (CI + BM*((L1 -L2)*CM)**2/(BM + CM)**2 + CM*(L1 -L2 -(L1
&          -L2)*CM/(BM + CM))**2 + BI11)
 PC(15) = (CI + BM*((L1 -L2)*CM)**2/(BM + CM)**2 + CM*(L1 -L2 -(L1
&          -L2)*CM/(BM + CM))**2 + BI22)
 PC(16) = ((L5**2 + L6**2)*DM + DI11)
 PC(17) = (FM*L8**2 + FI1)
 PC(18) = 1.0/PC(13)
 RETURN
 END
```

APPENDIX E — MANIPULATOR EQUATIONS

This appendix contains the equations of motion for the “Stanford Arm” manipulator described in Section 9.6.

```
C  Stanford arm simulation program.
C  Version created December 22, 1989 by AUTOSIM
C
C  (c) Mike Sayers and The Regents of The University of Michigan, 1989.
C   All rights reserved.
C
C  This program simulates the stanford arm by numerically integrating
C  the 12 ordinary differential equations that describe the kinematics
C  and dynamics of the system.  The stanford arm is composed of 6 bodies
C  and has 6 degrees of freedom.
C
C  Each derivative evaluation requires 378 multiply/divides, 268
C  add/subtracts, and 8 function/subroutine calls.
C
C  Bodies:
C  =======
C  A; parent=N; 1 DOF: Q(1)
C  B; parent=A; 1 DOF: Q(2)
C  C; parent=B; 1 DOF: Q(3)
C  D; parent=C; 1 DOF: Q(4)
C  E; parent=D; 1 DOF: Q(5)
C  F; parent=E; 1 DOF: Q(6)
C
C  Generalized Coordinates:
C  ========================
C    Q(1): Rotation of A relative to the inertial reference about axis
C          #2. (rad)
C    Q(2): Rotation of B relative to A about axis #1. (rad)
C    Q(3): Translation of C0 relative to the center of mass of B along
C          [b2]. (m)
C    Q(4): Rotation of D relative to C about axis #2. (rad)
C    Q(5): Rotation of E relative to D about axis #1. (rad)
C    Q(6): Rotation of F relative to E about axis #2. (rad)
C
C  Independent Speeds:
C  ===================
C    U(1): Abs. rot. of A, axis 2. (rad/s)
C    U(2): Rot. of B relative to A, axis 1. (rad/s)
C    U(3): Trans. speed of C0 relative to center of mass of B along
C          [b2]. (m/s)
C    U(4): Rot. of D relative to C, axis 2. (rad/s)
C    U(5): Rot. of E relative to D, axis 1. (rad/s)
C    U(6): Rot. of F relative to E, axis 2. (rad/s)
C
```

```
C  Active Forces:
C  ==============
C  FORCEM(6): (negative) force applied to C
C
C  Active Moments:
C  ===============
C  FORCEM(1): torque applied to A
C  FORCEM(2): (negative) torque applied to B
C  FORCEM(3): (negative) torque applied to D
C  FORCEM(4): (negative) torque applied to E
C  FORCEM(5): torque applied to F
```

The computations that are performed “in the loop” are the following:

```
      S(2) = SIN(Q(2))
      S(4) = SIN(Q(4))
      S(5) = SIN(Q(5))
      S(6) = SIN(Q(6))
      C(2) = COS(Q(2))
      C(4) = COS(Q(4))
      C(5) = COS(Q(5))
      C(6) = COS(Q(6))
C
C
C  Kinematical equations
C
      QP(1) = U(1)
      QP(2) = U(2)
      QP(3) = U(3)
      QP(4) = U(4)
      QP(5) = U(5)
      QP(6) = U(6)
C
C  define expression for torque applied to A
C
      FORCEM(1) = (PC(56) -K1*Q(1) -K2*QP(1))
C
C  define expression for torque applied to B
C
      Z(1) = C(5)*S(2)
      FORCEM(2) = -(PC(49) -K3*Q(2) -K4*QP(2) -(PC(55) +
     &      PC(54)*Q(3))*S(2) -PC(53)*(Z(1) + C(2)*C(4)*S(5)))
C
C  define expression for torque applied to D
C
      Z(2) = S(4)*S(5)
      FORCEM(3) = -(PC(48) -K5*Q(4) -K6*QP(4) + PC(5)*Z(2)*S(2))
C
C  define expression for torque applied to E
C
      Z(3) = C(2)*S(5)
      FORCEM(4) = -(PC(47) -K7*Q(5) -K8*QP(5) -PC(5)*(Z(3) +
     &      Z(1)*C(4)))
C
C  define expression for torque applied to F
C
```

```
      FORCEM(5) = (PC(46) -K9*Q(6) -K10*QP(6))
C
C  define expression for force applied to C
C
      FORCEM(6) = -(PC(45) -K11*Q(3) -K12*QP(3) + PC(7)*C(2))
C
C  Dynamical equations
C
      Z(4) = U(1)*S(2)
      Z(5) = U(1)*C(2)
      Z(6) = -(Z(4)*C(4) -U(2)*S(4))
      Z(7) = (U(2)*C(4) + Z(4)*S(4))
      Z(8) = S(2)*S(4)
      Z(9) = C(4)*S(2)
      Z(10) = (U(4) + Z(5))
      Z(11) = (Z(6)*C(5) -Z(10)*S(5))
      Z(12) = (Z(10)*C(5) + Z(6)*S(5))
      Z(13) = (Z(3) + Z(9)*C(5))
      Z(14) = (C(2)*C(5) -Z(9)*S(5))
      Z(15) = C(5)*S(4)
      Z(16) = (U(5) + Z(7))
      Z(17) = (Z(11)*C(6) + Z(16)*S(6))
      Z(18) = (Z(16)*C(6) -Z(11)*S(6))
      Z(19) = (Z(13)*C(6) -Z(8)*S(6))
      Z(20) = (Z(8)*C(6) + Z(13)*S(6))
      Z(21) = (Z(15)*C(6) + C(4)*S(6))
      Z(22) = (C(4)*C(6) -Z(15)*S(6))
      Z(23) = S(5)*S(6)
      Z(24) = C(6)*S(5)
      Z(25) = (U(6) + Z(12))
      Z(26) = (PC(9) + Q(3))
      Z(27) = Z(26)*S(2)
      Z(28) = L1*S(2)
      Z(29) = L1*C(2)
      Z(30) = -(Z(27) -PC(9)*S(2))
      Z(31) = (PC(9) -Z(26))
      Z(32) = Z(31)*C(4)
      Z(33) = -(-L2*Z(8) + Z(29)*C(4) + Z(30)*S(4))
      Z(34) = -(Z(28)*C(5) -Z(33)*S(5))
      Z(35) = (L2*Z(9) + PC(10)*Z(13) -Z(30)*C(4) + Z(29)*S(4))
      Z(36) = (PC(10)*Z(8) + Z(33)*C(5) + Z(28)*S(5))
      Z(37) = (-Z(32) + L2*C(4))*S(5)
      Z(38) = (C(4)*(PC(10) + L2*C(5)) -Z(32)*C(5))
      Z(39) = (PC(10)*Z(15) + (L2 -Z(31))*S(4))
      Z(40) = PC(10)*S(5)
      Z(41) = U(2)*Z(5)
      Z(42) = U(2)*Z(4)
      Z(43) = -(U(4)*Z(7) -Z(41)*C(4))
      Z(44) = (U(4)*Z(6) -Z(41)*S(4))
      Z(45) = (U(5)*Z(12) + Z(43)*C(5) -Z(42)*S(5))
      Z(46) = -(-U(5)*Z(11) + Z(42)*C(5) + Z(43)*S(5))
      Z(47) = Z(4)*Z(5)
      Z(48) = 2.0*Z(41)
      Z(49) = Z(4)**2
      Z(50) = (Z(49) + U(2)**2)
      Z(51) = (-Z(26)*Z(50) + GEES*C(2))
      Z(52) = (2.0*U(3)*Z(4) + Z(26)*Z(48) -L1*(Z(49) + Z(5)**2))
```

```
 Z(53) = (-2.0*U(2)*U(3) + Z(26)*Z(47) + GEES*S(2))
 Z(54) = Z(6)*Z(10)
 Z(55) = Z(7)*Z(10)
 Z(56) = (PC(9)*Z(48) -Z(52))
 Z(57) = (-PC(9)*Z(47) + Z(53))
 Z(58) = Z(12)*Z(16)
 Z(59) = Z(11)*Z(12)
 Z(60) = (-L2*(Z(6)**2 + Z(7)**2) + PC(9)*Z(50) + Z(51))
 Z(61) = (L2*(Z(44) -Z(54)) + Z(57)*C(4) + Z(56)*S(4))
 Z(62) = (-PC(10)*(Z(11)**2 + Z(16)**2) + Z(60)*C(5) -Z(61)*S(5))
 Z(63) = (PC(10)*(Z(44) -Z(59)) + Z(61)*C(5) + Z(60)*S(5))
 Z(64) = (L2*(Z(43) + Z(55)) + PC(10)*(Z(45) + Z(58)) -Z(56)*C(4)
&         + Z(57)*S(4))
 Z(65) = PC(2)*Z(29)
 Z(66) = PC(2)*Z(28)
 Z(67) = PC(2)*Z(27)
 Z(68) = PC(2)*Z(26)
 Z(69) = (D3*Z(43) + PC(19)*Z(55))
 Z(70) = -(PC(21)*Z(6)*Z(7) -D2*Z(42))
 Z(71) = D2*C(2)
 Z(72) = D3*S(4)
 Z(73) = (PC(23)*Z(45) + PC(22)*Z(58))
 Z(74) = (PC(26)*Z(44) + PC(25)*Z(59))
 Z(75) = PC(3)*Z(36)
 Z(76) = PC(3)*Z(35)
 Z(77) = PC(3)*Z(34)
 Z(78) = PC(26)*Z(8)
 Z(79) = PC(3)*Z(39)
 Z(80) = PC(3)*Z(38)
 Z(81) = PC(3)*Z(37)
 Z(82) = PC(26)*C(4)
 Z(83) = PC(23)*Z(15)
 Z(84) = PC(3)*S(5)
 Z(85) = PC(3)*C(5)
 Z(86) = PC(3)*Z(40)
 Z(87) = PC(23)*S(5)
 Z(88) = (PC(27)*Z(18)*Z(25) + F3*(-U(6)*Z(18) + Z(45)*C(6) +
&         Z(44)*S(6)))
 Z(89) = (PC(28)*Z(17)*Z(25) + F1*(U(6)*Z(17) + Z(44)*C(6)
&         -Z(45)*S(6)))
 Z(90) = (PC(29)*Z(17)*Z(18) + F2*Z(46))
 Z(91) = F2*Z(14)
 Z(92) = F1*Z(22)
 Z(93) = F3*Z(21)
 Z(94) = F2*Z(2)
 Z(95) = F2*C(5)
 Z(96) = F3*Z(24)
 Z(97) = F1*Z(23)
 Z(98) = F3*S(6)
 Z(99) = F1*C(6)
 Z(100) = (D1*Z(44) + PC(20)*Z(54) + Z(74))
 Z(101) = -(PC(24)*Z(11)*Z(16) + E2*Z(46) + Z(90))
 Z(102) = (-Z(53)*Z(65) + Z(51)*Z(66) -Z(52)*Z(67) -Z(9)*Z(69)
&          -Z(13)*Z(73) + Z(63)*Z(75) -Z(64)*Z(76) -Z(62)*Z(77)
&          -Z(19)*Z(88) + Z(20)*Z(89) + Z(8)*Z(100) + Z(14)*Z(101)
&          + FORCEM(1) + Z(28)*FORCEM(6) + (PC(40)*Z(42) +
&          Z(70))*C(2) -(PC(43)*Z(41) + PC(44)*FORCEM(6))*S(2))
```

```
 Z(103) = (Z(82) + D1*C(4))
 Z(104) = (E2*Z(2) + Z(94))
 Z(105) = (-Z(29)*Z(68) -Z(9)*Z(72) -Z(35)*Z(79) + Z(36)*Z(80) +
&          Z(34)*Z(81) -Z(13)*Z(83) + Z(20)*Z(92) -Z(19)*Z(93) +
&          Z(8)*Z(103) + Z(14)*Z(104))
 Z(106) = (PC(2)*Z(51) + Z(63)*Z(84) + Z(62)*Z(85) + FORCEM(6))
 Z(107) = (Z(66) + Z(36)*Z(84) -Z(34)*Z(85))
 Z(108) = (Z(38)*Z(84) -Z(37)*Z(85))
 Z(109) = (-Z(70) + Z(64)*Z(86) + Z(24)*Z(88) -Z(23)*Z(89) +
&          FORCEM(3) -Z(101)*C(5) + Z(73)*S(5))
 Z(110) = (Z(95) + E2*C(5))
 Z(111) = (Z(71) + Z(35)*Z(86) + Z(13)*Z(87) + Z(19)*Z(96) +
&          Z(20)*Z(97) + Z(14)*Z(110))
 Z(112) = (-Z(39)*Z(86) -Z(15)*Z(87) -Z(21)*Z(96) + Z(22)*Z(97) +
&          Z(2)*Z(110))
 Z(113) = (-PC(4)*Z(63) -Z(74) + FORCEM(4) -Z(89)*C(6)
&          -Z(88)*S(6))
 Z(114) = (PC(4)*Z(36) + Z(78) -Z(19)*Z(98) + Z(20)*Z(99))
 Z(115) = (PC(4)*Z(38) + Z(82) + Z(21)*Z(98) + Z(22)*Z(99))
 Z(116) = PC(4)*S(5)
 Z(117) = -(Z(24)*Z(98) -Z(23)*Z(99))
 Z(118) = (-Z(90) + FORCEM(5))
 Z(119) = PC(33)*Z(116)
 Z(120) = PC(33)*Z(107)
 Z(121) = PC(33)*Z(108)
 Z(122) = PC(34)*Z(95)
 Z(123) = PC(34)*Z(91)
 Z(124) = PC(34)*Z(94)
 Z(125) = (D2 + Z(40)*Z(86) + Z(24)*Z(96) + Z(23)*Z(97)
&          -Z(95)*Z(122) + Z(110)*C(5) + Z(87)*S(5))
 Z(126) = Z(117)/Z(125)
 Z(127) = (Z(111) -Z(95)*Z(123))/Z(125)
 Z(128) = (Z(112) -Z(95)*Z(124))/Z(125)
 Z(129) = (PC(32) -Z(116)*Z(119) -Z(117)*Z(126) + Z(99)*C(6) +
&          Z(98)*S(6))
 Z(130) = (Z(114) -Z(116)*Z(120) -Z(117)*Z(127))/Z(129)
 Z(131) = (Z(115) -Z(116)*Z(121) -Z(117)*Z(128))/Z(129)
 Z(132) = (Z(111) -Z(91)*Z(122))
 Z(133) = (Z(114) -Z(107)*Z(119) -Z(126)*Z(132))
 Z(134) = (PC(30) + D3*Z(9)**2 + PC(23)*Z(13)**2 + F3*Z(19)**2 +
&          F1*Z(20)**2 + Z(29)*Z(65) + Z(28)*Z(66) + Z(27)*Z(67) +
&          Z(36)*Z(75) + Z(35)*Z(76) + Z(34)*Z(77) + Z(8)*(D1*Z(8)
&          + Z(78)) + Z(14)*(E2*Z(14) + Z(91)) -Z(107)*Z(120)
&          -Z(91)*Z(123) -Z(127)*Z(132) -Z(130)*Z(133) + (Z(71) +
&          PC(36)*C(2))*C(2) + PC(37)*S(2)**2)
 Z(135) = (Z(105) -Z(107)*Z(121) -Z(91)*Z(124) -Z(128)*Z(132)
&          -Z(131)*Z(133))/Z(134)
 Z(136) = (Z(112) -Z(94)*Z(122))
 Z(137) = (Z(115) -Z(108)*Z(119) -Z(126)*Z(136))
 Z(138) = (Z(105) -Z(108)*Z(120) -Z(94)*Z(123) -Z(127)*Z(136)
&          -Z(130)*Z(137))
 Z(139) = Z(118)*Z(122)
 Z(140) = (Z(109) + Z(139))
 Z(141) = Z(106)*Z(119)
 Z(142) = Z(126)*Z(140)
 Z(143) = (Z(113) + Z(141) -Z(142))
 Z(144) = Z(106)*Z(120)
```

```
 Z(145) = Z(118)*Z(123)
 Z(146) = Z(127)*Z(140)
 Z(147) = Z(130)*Z(143)
 Z(148) = (PC(35)*Z(47) -Z(53)*Z(68) -Z(15)*Z(73) -Z(64)*Z(79)
&          -Z(63)*Z(80) + Z(62)*Z(81) -Z(21)*Z(88) -Z(22)*Z(89)
&          -Z(2)*Z(101) + Z(106)*Z(121) + Z(118)*Z(124)
&          -Z(128)*Z(140) -Z(131)*Z(143) + Z(135)*(Z(102) -Z(144)
&          -Z(145) + Z(146) + Z(147)) + FORCEM(2) -Z(100)*C(4)
&          -Z(69)*S(4))/(PC(31) + Z(26)*Z(68) + Z(39)*Z(79) +
&          Z(38)*Z(80) + Z(37)*Z(81) + Z(15)*Z(83) + Z(22)*Z(92) +
&          Z(21)*Z(93) + Z(2)*Z(104) -Z(108)*Z(121) -Z(94)*Z(124)
&          -Z(128)*Z(136) -Z(131)*Z(137) -Z(135)*Z(138) +
&          Z(103)*C(4) + Z(72)*S(4))
 Z(149) = (Z(102) -Z(144) -Z(145) + Z(146) + Z(147) +
&          Z(138)*Z(148))/Z(134)
 Z(150) = (Z(113) + Z(141) -Z(142) -Z(137)*Z(148) +
&          Z(133)*Z(149))/Z(129)
 Z(151) = (Z(109) + Z(139) -Z(136)*Z(148) + Z(132)*Z(149)
&          -Z(117)*Z(150))/Z(125)
 UP(2) = -Z(148)
 UP(1) = Z(149)
 UP(5) = -Z(150)
 UP(4) = -Z(151)
 UP(6) = PC(34)*(Z(118) + Z(94)*Z(148) -Z(91)*Z(149) +
&         Z(95)*Z(151))
 UP(3) = -PC(33)*(Z(106) + Z(108)*Z(148) -Z(107)*Z(149) +
&         Z(116)*Z(150))
```

The above equations refer to constants that can be precomputed, and are define below.

```
 PC(1) = L5*MD
 PC(2) = (MC + MD)
 PC(3) = (ME + MF)
 PC(4) = (L6*ME + L3*MF)
 PC(5) = (L6*ME + L3*MF)*GEES
 PC(6) = (MC + MD + ME + MF)
 PC(7) = (MC + MD + ME + MF)*GEES
 PC(8) = L1*MB/(MA + MB)
 PC(9) = L5*MD/(MC + MD)
 PC(10) = (L6*ME + L3*MF)/(ME + MF)
 PC(11) = L1*(1 -MB/(MA + MB))
 PC(12) = (B1 -B2)
 PC(13) = (B2 -B3)
 PC(14) = (B1 -B3)
 PC(15) = (C1 -C2 + MD*(L5*(1 -MD/(MC + MD)))**2 + MC
&         *(L5*MD)**2/(MC + MD)**2)
 PC(16) = (C3 + MD*(L5*(1 -MD/(MC + MD)))**2 + MC*(L5*MD)**2/(MC +
&          MD)**2)
 PC(17) = (C2 -C3 -MD*(L5*(1 -MD/(MC + MD)))**2 -MC*(L5*MD)**2/(MC
&          + MD)**2)
 PC(18) = (C1 -C3)
 PC(19) = (D1 -D2)
 PC(20) = (D2 -D3)
 PC(21) = (D1 -D3)
 PC(22) = (E1 -E2 + MF*(L3 -(L6*ME + L3*MF)/(ME + MF))**2 + ME*(L6
&          -(L6*ME + L3*MF)/(ME + MF))**2)
 PC(23) = (E3 + MF*(L3 -(L6*ME + L3*MF)/(ME + MF))**2 + ME*(L6
```

```
&          -(L6*ME + L3*MF)/(ME + MF))**2)
 PC(24) = (E1 -E3)
 PC(25) = (E2 -E3 -MF*(L3 -(L6*ME + L3*MF)/(ME + MF))**2 -ME*(L6
&          -(L6*ME + L3*MF)/(ME + MF))**2)
 PC(26) = (E1 + MF*(L3 -(L6*ME + L3*MF)/(ME + MF))**2 + ME*(L6
&          -(L6*ME + L3*MF)/(ME + MF))**2)
 PC(27) = (F1 -F2)
 PC(28) = (F2 -F3)
 PC(29) = (F1 -F3)
 PC(30) = (A2 + MB*(L1*(1 -MB/(MA + MB)))**2 + (2.0*MA + MB)
&         *(L1*MB)**2/(MA + MB)**2)
 PC(31) = (B1 + C1 + MC*(L5*MD)**2/(MC + MD)**2 + MD*(L5*(1
&          -MD/(MC + MD)))**2)
 PC(32) = (E1 + ME*(L6 -(L6*ME + L3*MF)/(ME + MF))**2 + MF*(L3
&          -(L6*ME + L3*MF)/(ME + MF))**2 + (L6*ME + L3*MF)**2/(ME
&          + MF))
 PC(33) = 1.0/PC(6)
 PC(34) = 1.0/F2
 PC(35) = (PC(13) + PC(17))
 PC(36) = (B2 + C2)
 PC(37) = (B3 + PC(16))
 PC(38) = (B2 + PC(14))
 PC(39) = (C2 + PC(18))
 PC(40) = (PC(38) + PC(39))
 PC(41) = (B3 + PC(12))
 PC(42) = (PC(15) + PC(16))
 PC(43) = (PC(41) + PC(42))
 PC(44) = (PC(8) + PC(11))
 PC(45) = K11*CDISP
 PC(46) = K9*FROT
 PC(47) = K7*EROT
 PC(48) = K5*DROT
 PC(49) = K3*BROT
 PC(50) = L2*PC(3)
 PC(51) = (PC(2) + PC(3))
 PC(52) = (PC(1) + PC(50))
 PC(53) = GEES*PC(4)
 PC(54) = GEES*PC(51)
 PC(55) = GEES*PC(52)
 PC(56) = K1*AROT
```

REFERENCES

1. "Laboratory Testing Machines and Procedures for Measuring the Steady State Force and Moment Properties of Passenger Car Tires." Society of Automotive Engineers, Inc., Handbook Supplement HS 210, (Recommended Practice SAE J1106), 1975.
2. "Users Manual for TREETOPS: A Control System Simulation for Structures with a Tree Topology." Honeywell (Space and Strategic Avionics Division), 784-19441, 1984.
3. "ADAMS Applications Manual." Mechanical Dynamics, Inc, MDI 2001-00, 1987.
4. *Common Lisp: The Reference.* 1988, Addison-Wesley.
5. "SD/FAST User's Manual." Symbolic Dynamics, Inc, Mountain View, CA, 1988.
6. "Allegro Common Lisp for the Macintosh." Apple Computer, Inc., 1989.
7. Abelson, H. and G.J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Electrical Engineering and Computer Science Series. 1985, The MIT Press, McGraw-Hill Book Co. New York.
8. Adeli, H. and Y.J. Paek. "Computer-aided Analysis of Structures in Interlisp Environment." *Computers & Structures* 23(3), 1986, pp. 393-407.
9. Amirouche, F.M.L. and S.K. Ider. "Determination of constraint forces in multibody systems dynamics using Kane's equations." *Journal de Mecanique Theorique et Appliquee* 7(1), 1988, pp. 3-20.
10. Amirouche, F.M.L., T. Jia and S.K. Ider. "Recursive householder transformation for complex dynamical systems with constraints." *Journal of Applied Mechanics, Transactions ASME* 55(3), 1988, pp. 729-734.
11. Antoun, R.J., P.B. Hackert, M.C. O'Leary and A. Sitchin, "Vehicle Dynamic Handling Computer Simulation: Model Development, Correlation, And Application Using Adams." *International Congress and Exposition - Society of Automotive Engineers,* Detroit, MI, SAE, paper 860574, 1986.
12. Ausiello, G. and F.M. Giovanni, "On the Design of Algebraic Data Structures with the Approach of Abstract Data Types." *EUROCAL '79 European Computer Algebra Conference,* Ed. E. W. Ng. Lecture Notes in Computer Science. Marseille, France, Springer-Verlag, 1979.
13. Bae, D.-S., R.S. Hwang and E.J. Haug. "A Recursive Formulation for Real-Time Dynamic Simulation." *Advances in Design Automation, ASME DE* 14(September), 1988, pp. 499-508.
14. Baumgarte, J. "Stabilization of Constraints and Integrals of Motion in Dynamical Systems." *Computer Methods in Applied Mech. and Eng.* 1, 1972, pp. 1-16.

15. Benerjee, A.K. “Comment On 'Relationship Between Kane's Equation And The Gibbs-Appell Equations'.” *Journal of Guidance, Control, and Dynamics* 10(6), 1987, pp. 596-597.

16. Bianchi, G. and W. Schiehlen. *Dynamics of Multibody Systems.* IUTAM/IFToMM Symposium Udine/Italy 1985. 1986, Springer-Verlag. Berlin.

17. Caviness, B.F., “Computer Algebra: Past and Future.” *EUROCAL ’85 European Computer Algebra Conference vol 1: invited lectures,* Ed. B. Buchberger. Lecture Notes in Computer Science. Linz, Austria, Springer-Verlag, 1985.

18. Chace, M.A. “Methods and Experience in Computer Aided Design of Large-Displacement Mechanical Systems.” *Computer Aided Analysis and Optimization of Mechanical System Dynamics*. E. G. Haug ed., 1984, Springer-Verlag, Heidelberg. 233-259.

19. Chang, C.O. and P.E. Nikravesh. “An Adaptive Constraint Violation Stabilization method for dynamic Analysis of Mechanical Systems.” *ASME Journal of Mechanisms, Transmissions, and Automation in Design* 107(December), 1985, pp. 488-498.

20. Char, B.W., K.O. Geddes, W.M. Gentleman and G.H. Gonnet, “The Design of MAPLE: A Compact, Portable, and Powerful Computer Algebra System.” *EUROCAL ’83 European Computer Algebra Conference,* Ed. J. A. van Hulzen. Lecture Notes in Computer Science. London, England, Springer-Verlag, 1983.

21. Crespo da Silva, M.R.M. and D.H. Hodges. “Role Of Computerized Symbolic Manipulation In Rotorcraft Dynamics Analysis.” *Computers & Mathematics with Applications* 12a(1), 1986, pp. 161-172.

22. Desloge, E.A. “A Comparison of Kane's Equations of Motion And The Gibbs-Appell Equations of Motion.” *American Journal of Physics* 54(May), 1986.

23. Desloge, E.A. “Relationship Between Kane's Equation And The Gibbs-Appell Equations.” *Journal of Guidance, Control, and Dynamics* 10(Jan-Feb), 1987, pp. 120-122.

24. Duffek, W., C. Fuehrer, W. Schwarz and O. Wallrapp, “Analysis and Simulation of Rail and Road Vehicles with the Program MEDYNA.” *Proceedings, 9th IAVSD Symposium, dynamics of Vehicles on Roads and Tracks,* Linkoping, 1985.

25. Featherstone, R. *Robot Dynamics Algorithms.* The Kluwer International Series in Engineering and Computer Science. Robotics: Vision, Manipulation, and Sensors. 1987, Kluwer Academic Publishers. Boston.

26. Frisch, H.P., “A Vector-Dyadic Development of the Equations of Motion for N-coupled Rigid Bodies and Point Masses.” Goddard Space Flight Center, D-7767, 1974.

27. Frisch, H.P., “A digital computer program for the dynamic interaction simulation of controls and structures.” NASA, Tech memo 80546, 1979.

28. Ge, Z., -M. and Y.-H. Cheng. “Extended Kane's Equations for Nonholonomic Mass System.” *ASME Journal of Applied Mechanics* 49(June), 1982, pp. 429-431.

29. Gear, C.W. *Numerical Initial Value Problems in Ordinary Differential Equations.* 1971, Prentice-Hall. Englewood Cliffs, N.J.

30. Gear, C.W. “Differential-Algebraic Equations.” *Computer Aided Analysis and Optimization of Mechanical System Dynamics*. E. G. Haug ed., 1984, Springer-Verlag, Heidelberg. 323-334.

31. Gilmore, B.J. and R.J. Cipra, “Simulation of Planar Dynamic Mechanical Systems with Changing Topologies: Part 1 — Characterization and Prediction of the Kinematic Constraint Changes.” *ASME Design Technology Copnverences — The Design Automation Conference,* Ed. S. S. Rao. Boston, ASME, 1987.

32. Gilmore, B.J. and R.J. Cipra, “Simulation of Planar Dynamic Mechanical Systems with Changing Topologies: Part 2 — Implementation Strategy and Simulation Results for Example Dynamic Systems.” *ASME Design Technology Copnverences — The Design Automation Conference,* Ed. S. S. Rao. Boston, ASME, 1987.

33. Golnaraghi, M., W. Keith and F.C. Moon. “Stability Analysis of a Robotic Mechanism Using Computer Algebra.” *Applications of Computer Algebra*. R. Pavelle ed., 1984, Kluwer Academic Publishers, Boston. 281-292.

34. Gomez, C., J.P. Quadrat and A. Sulem. “Computer Algebra as a Tool for Solving Optimal Control Problems.” *Applications of Computer Algebra*. R. Pavelle ed., 1984, Kluwer Academic Publishers, Boston. 241-261.

35. Greenwood, D.T. *Principles of Dynamics.* Second. 1988, Prentice-Hall, Inc. Englewood Cliffs.

36. Hackert, P.B., M.C. O'Leary and A. Sitchin, “Dynamic Simulation Of Light Truck Handling Maneuvers Using Adams.” *Symposium on Simulation and Control of Ground Vehicles and Transportation Systems*. *(Presented at the Winter Annual Meeting of the American Society of Mechanical Engineers.),* Anaheim, CA, ASME (DSC v 2), 1986.

37. Hamming, R.W. *Numerical Methods for Engineers and Scientists.* 1962, McGraw Hill. New York.

38. Haug, E.G. *Computer Aided Analysis and Optimization of Mechanical System Dynamics.* NATO ASI Series, Vol. F9. 1984, Springer-Verlag. Heidelberg.

39. Haug, E.G. “Elements and Methods of Computational Dynamics.” *Computer Aided Analysis and Optimization of Mechanical System Dynamics*. E. G. Haug ed., 1984, Springer-Verlag, Heidelberg. 3-40.

40. Hirschberg, W. and D. Schramm. “Application of NEWEUL in Robot Dynamics.” *Journal of Symbolic Computation* (7), 1989, pp. 199-204.

41. Howe, R.M., “Dynamics of Real-Time Digital Simulation: course notes.” Applied Dynamics International, Ann Arbor, 1986.

42. Howe, R.M. and A. Nwankpa. “Some Improved Methods for Real-Time Integration of State Variable Derivatives wit Discontinuities.” , 1988.

43. Hsu, S., “An Improved Method for Modeling Constrained Rigid Body Systems.” PhD thesis, University of Mighigan, 1986.

44. Hussain, M.A. and B. Noble. “Application of Macsyma to Kinematics and Mechanical Systems.” *Applications of Computer Algebra*. R. Pavelle ed., 1984, Kluwer Academic Publishers, Boston. 262-280.

45. Hussain, M.A. and B. Noble. “Application of symbolic Computation to the Analysis of Mechanical systems, Including Robot Arms.” *Computer Aided Analysis and Optimization of Mechanical System Dynamics*. E. G. Haug ed., 1984, Springer-Verlag, Heidelberg. 283-306.

46. Huston, R.L., “Useful Procedures in Multibody Dynamics.” *Dynamics of Multibody Systems, IUTAM/IFToMM Symposium,* Ed. G. Bianchi and W. Schiehlen. Udine, Italy, Springer-Verlag, 1985.

47. Huston, R.L. and C. Passerello. “On Multi-Rigid-Body System Dynamics.” *Computers and Structures* 10, 1979, pp. 439-446.

48. Huston, R.L. and C.E. Passerello. “On the Dynamics of Chain Systems.” *Automatic Control Division of the American Society of Mechanical Engineers* , 1974.

49. Huston, R.L. and C.E. Passerello. “Multibody Structural Dynamics Including Translation Between the Bodies.” *Computers and Structures* 12, 1980, pp. 713-720.

50. Huston, R.L., C.E. Passerello and M.W. Harlow. “Dynamics of Multirigid-Body Systems.” *ASME Journal of Applied Mechanics* 45(December), 1978, pp. 889-894.

51. Ider, S.K. and F.M.L. Amirouche. “Coordinate reduction in the dynamics of constrained multibody systems - a new approach.” *Journal of Applied Mechanics, Transactions ASME* 55(4), 1988, pp. 899-904.

52. Jaschinski, A., W. Kortuem and O. Wallrapp. “Simulation of ground vehicles with the multibody program MEDYNA.” , 1986.

53. Kamman, J.W. and R.L. Huston. “Constrained Multibody System Dynamics, an Automated Approach.” *Computers and Structures* 18(6), 1984, pp. 999-1003.

54. Kamman, J.W. and R.L. Huston. “Dynamics Of Constrained Multibody Systems.” *Journal of Applied Mechanics, Transactions ASME* 51(4), 1984, pp. 899-903.

55. Kane, T.R. and D.A. Levinson. “Formulation of Equations of Motion for Complex Spacecraft.” *Journal of Guidance and Control* 3(2), 1980, pp. 99-112.

56. Kane, T.R. and D.A. Levinson. “Multibody dynamics.” *Journal of Applied Mechanics, Transactions ASME* 50(4b), 1983, pp. 1071-1078.

57. Kane, T.R. and D.A. Levinson. “The Use of Kane’s Dynamical Equations in Robotics.” *International Journal of Robotics Research* 2(3), 1983, pp. 3-21.

58. Kane, T.R. and D.A. Levinson. *Dynamics, theory and applications.* McGraw-Hill Series in Mechanical Engineering. 1985, McGraw-Hill Book Company.

59. Keat, J.E. “Comment on "Relationship Between Kane's Equations and the Gibbs-Appell Equations".” *Journal of Guidance, Control, and Dynamics* 10(6), 1987, pp. 594-595.

60. Kessler, R.R. *LISP, Objects and Symbolic Programming.* 1988, Scott, Foresman and Co. Glenview, Illinois.

61. Kim, S.S. and M.J. Vanderploeg. “QR Decomposition for State Space Representation of Constrained Mechanical Dynamic Systems.” *ASME Journal of Mechanisms, Transmission, and Automation in Design* 108(June), 1986, pp. 183-188.

62. Kortuem, W., “Simulation of the dynamics of high speed ground transportation vehicles with MEDYNA - potentials and case studies.” *International Conference on Maglev and Linear Drives,* Las Vegas, NV, USA, IEEE, 1987.

63. Kortuem, W. and W. Schiehlen. “General Purpose Vehicle System Dynamics Software Based on Multibody Formalism.” *Vehicle System Dynamics* 14(4-6), 1985, pp. 229-263.

64. Kreuzer, E. and O. Schiehlen, “Generation of Symbolic Equations of Motion for Complex Spacecraft Using Formalism NEWEUL.” *AIAA Astrodynamics Specialist Conference,* 1983.

65. Kreuzer, E.J., “Dynamic Analysis of Mechanisms Using Symbolical Equation Manipulation.” *Proceedings, 5th World Congress on Theory of Machines and Mechanisms,* Montreal, 1979.

66. Kreuzer, E.J. and W.O. Schiehlen. “Computerized Generation Of Symbolic Equations Of Motion For Spacecraft.” *Journal of Guidance, Control, and Dynamics* 8(2), 1985, pp. 284-287.

67. Krishnaswami, P. and M.A. Bhatti. “Symbolic Computing in Optimal Design of Dynamic Systems.” *The American Society of Mechanical Engineers* , 1985, pp. 1-6.

68. Kurdila, A. and M. Kamat, “Concurrent nullspace methods for multibody systems.” *Parallel and Distributed Processing in Structural Engineering, Proceedings. Presented in Conjunction with the ASCE National Convention.,* Nashville, TN, USA, ASCE, 1988.

69. Levinson, D. “The Derivation of Equations of Motion of Multiple-Rigid-Body Systems Using Symbolic Manipulation.” *AIAA paper No. 76-816* , 1976.

70. Levinson, D. “Comment on "Relationship Between Kane's Equations and the Gibbs-Appell Equations".” *Journal of Guidance, Control, and Dynamics* 10(6), 1987, pp. 593.

71. Liang, C.G. and G.M. Lance. “A Differentiable Null Space Method for Constrained Dynamic Analysis.” *ASME Journal of Mechanisms, Transmissions, and Automation in Design* 109(September), 1987, pp. 405-411.

72. Lilov, L. and V. Chirikov. “On the Dynamics Equations of Systems of Interconnected Bodies.” *Journal of Applied Mathematic and Mechanics* 45, 1981, pp. 383-390.

73. Lin, L.-C. and Y. King. “Lagrange-Euler-assumed modes approach to modeling flexible robotic manipulators.” *Chung-kuo Kung Ch'eng Hsueh K'an/Journal of the Chinese Institute of Engineers* 11(4), 1988, pp. 335-347.

74. Lips, K.W.;.S., R. P., “Obstacles to high fidelity multibody dynamics simulation.” *Proceedings of the 1988 American Control Conference.*, Atlanta, GA, USA, IEEE, 1988.

75. Liu, Y. “Screw-matrix method in dynamics of multibody systems.” *Acta Mechanica Sinica/Lixue Xuebao* 4(2), 1988, pp. 165-174.

76. Loos, H. and G. Doedlbacher, “Mathematical 'Prototype' Of The Vehicle To Describe Vehicle Handling Behavior.” *Dynamics of Vehicles on Roads and on Tracks, Proceedings of 9th IAVSD Symposium.*, Linkoping, Swed, Swets North America, 1986.

77. Magnus, K. *Dynamics of Multibody Systems*. IUTAM/IFToMM Symposium Munich/Germany 1977. 1978, Springer-Verlag. Berlin.

78. Mani, N.K. and E.J. Haug. “Application of Singular Value Decomposition for Analysis of Mechanical System Dynamics.” *ASME Journal of Mechanisms, Transmissions, and Automation in Design* 107(March), 1985, pp. 82-87.

79. McConville, J.B. and J.C. Angell, “Dynamic Simulation Of A Moving Vehicle Subject To Transient Steering Inputs Using The Adams Computer Program.” *Design Engineering Technology Conference*, Cambridge, MA, ASME (84-DET-2), 1984.

80. McInnis, J.B. and W.H. ElMaraghy, “Automated bond graph construction and analysis for multibody system dynamics.” *Proceedings of the 1989 ASME International Computers in Engineering Conference and Exposition*, Anheim, CA, USA, 1989.

81. Month, L.A. and R.H. Rand, “Stability Of A Rigid Body With An Oscillating Particle: An Application Of Macsyma.” *1985 Joint ASME/ASCE Applied Mechanics, Fluids Engineering and Bioengineering Conference.*, Albuquerque, NM, ASME (85-APM-28), 1985.

82. Nielan, P. and T. Kane, “Symbolic Generation of Efficient Simulation/Control Routines for Multibody Systems.” *Dynamics of Multibody Systems, IUTAM/IFToMM Symposium*, Ed. G. Bianchi and W. Schiehlen. Udine, Italy, Springer-Verlag, 1985.

83. Nielan, P.E., "Efficient Computer Simulation of Motions of Multibody Systems." PhD thesis, Stanford University, 1986.

84. Nikravesh, P.E. "Some Methods for Dynamic Analysis of Constrained Mechanical Systems: A Survey." *Computer Aided Analysis and Optimization of Mechanical System Dynamics*. E. G. Haug ed., 1984, Springer-Verlag, Heidelberg. 353-367.

85. Nikravesh, P.E. and E.J. Haug. "Generalized Coordinate Partitioning for Analysis of Mechanical Systems with Nonholonomic Constraints." *ASME Journal of Mechanisms, Transmissions, and Automation in Design* 105(September), 1983, pp. 379-384.

86. Orlandea, N. and M.A. Chace, "Simulation Of A Vehicle Suspension With The Adams Computer Program." *SAE,* Detroit, SAE preprint 770053, 1977.

87. Orlandea, N., M.A. Chace and D.A. Calahan. "A Sparsity-Oriented Approach to the Dynamic Analysis and Design of Mechanical Systems, Parts I and II." *Journal of Engineering for Industry* 99(August), 1977, pp. 773-784.

88. Ormrod, M. and G. Andrews. "Advent: A Simulation Program for Constrained Planar Kinematic and Dynamic Systems." , 1986, pp. 1-9.

89. Park, K.C. and J.C. Chiou. "Stabilization of Computational Procedures for Constrained Dynamical Systems." *Journal of Guidance and Control* 11(4), 1988, pp. 365-370.

90. Park, T.W. and E.J. Haug. "A Hybrid Numerical Integration Method for Machine Dynamics Simulation." *ASME Journal of Mechanism, Transmissions, and Automation in Design* 108(June), 1986, pp. 211-216.

91. Passerello, C.E. and R.L. Huston. "Another Look at Nonholonomic Systems." *ASME Journal of Applied Mechanics* 40(1), 1973, pp. 101-104.

92. Pavelle, R., "Macsyma: Capabilities and Applications to Problems in Engineering and the Sciences." *EUROCAL '85 European Computer Algebra Conference,* Ed. B. Buchberger. Lecture Notes in Computer Science. Linz, Austria, Springer-Verlag, 1985.

93. Press, W.H., B.P. Flannery, S.A. Teukolsky and W.T. Vetterling. *Numerical Recipes: the Art of Scientific Computing.* 1986, Cambridge University Press.

94. Rayna, G. *REDUCE software for Algebraic Computation.* Springer Series, Symbolic Computation—Artificial Intilligence. 1987, Springer-Verlag. New York.

95. Richard, M., R. Anderson and G. Andrews. "Generalized Vector-Network Formulation for the Dynamic Simulation of Multibody Systems." *Journal of Dynamic Systems, Measurement, and Control* 108, 1986, pp. 322-329.

96. Roberson, R.E., "Constraint Stabilization for Rigid Bodies: an Extension of Baumgarte's Method." *Dynamics of Multibody Systems,* Ed. K. Magnus. International Union of Theoretical and Applied Mechanics. Munich, Springer-Verlag, 1977.

97. Roberson, R.E. and Schwertassek. *Dynamics of Multibody Systems*. 1988, Springer-Verlag. Berlin.

98. Rosenthal, D.E. “Comment On 'Relationship Between Kane's Equation And The Gibbs-Appell Equations'.” *Journal of Guidance, Control, and Dynamics* 10(6), 1987, pp. 595-596.

99. Rosenthal, D.E. “Triangularization of equations of motion for robotic systems.” *Journal of Guidance, Control, and Dynamics* 11(3), 1988, pp. 278-281.

100. Rosenthal, D.E. and M.A. Sherman, “Symbolic Multibody Equations via Kane’s Method.” *AAS/AIAA Astrodynamics Specialist Conference,* Lake Placid, 1983.

101. Rosenthal, D.E. and M.A. Sherman. “High Performance Multibody Simulations via Symbolic Equation Manipulation and Kane’s Method.” *Journal of the Astronoutical Sciances* 34(3), 1986, pp. 223-239.

102. Sayers, M.W., “ERD Data-Processing Software Reference Manual, Version 2.00.” University of Michigan Transportation Research Institute, UMTRI-87-2, 1987.

103. Sayers, M.W., “Automated Formulation of Efficient Vehicle Simulation Codes by Symbolic Computation (AUTOSIM).” *11th IAVSD Symposium of Vehicles on Roads and Tracks,* Kingston, Ontario, 1989.

104. Sayers, M.W., “AUTOSIM: A Computer Language for Representing Multibody Systems in Symbolic Form to Automatically Formulate Efficient Simulation Codes.” *The Seventh Army Conference on Applied Mathematics and Computing,* West Point, New York, 1989.

105. Sayers, M.W., “EP Users Manual, the ERD Plotter for the Macintosh.” University of Michigan Transportation Research Institute, 1989.

106. Schaechter, D.B. and D.A. Levinson. “Interactive computerized symbolic dynamics for the dynamicist.” *Journal of the Astronautical Sciences* 36(4), 1988, pp. 365-388.

107. Schiehlen, W.O., “Dynamical analysis of suspension systems.” *The Dynamics of Vehicles on Roads and on Tracks. Proceedings. Amsterdam,* Ed. A. Slibar and H. Springer. Amsterdam, Swets and Zeitlinger, 1978.

108. Schiehlen, W.O. “Modeling of Complex Vehicle Systems.” *Vehicle System Dynamics* 12(1-3), 1983, pp. 12-14.

109. Schiehlen, W.O. “Computer Generation of Equations of Motion.” *Computer Aided Analysis and Optimization of Mechanical System Dynamics*. E. G. Haug ed., 1984, Springer-Verlag, Heidelberg. 183-215.

110. Schiehlen, W.O. “Dynamics Of Complex Multibody Systems.” *Solid Mechanics Archives* 9(2), 1984, pp. 159-195.

111. Schiehlen, W.O. “Vehicle Dynamics Applications.” *Computer Aided Analysis and Optimization of Mechanical System Dynamics*. E. G. Haug ed., 1984, Springer-Verlag, Heidelberg. 217-231.

112. Schiehlen, W.O. and E.J. Kreuzer, “Symbolic Computerized Derivation of Equations of Motion.” *Dynamics of Multibody Systems,* Ed. K. Magnus. International Union of Theoretical and Applied Mechanics. Munich, Springer-Verlag, 1977.

113. Schwertassek, R. and R.E. Roberson, “A Perspective on Computer-Oriented Multibody Dynamical Formalisms and their Implementations.” *Dynamics of Multibody Systems, IUTAM/IFToMM Symposium,* Ed. G. Bianchi and W. Schiehlen. Udine, Italy, Springer-Verlag, 1985.

114. Segel, L. “Theoretical Prediction and Experimental Substantiation of the Response of the Automobile to Steering Control.” *Proceedings of the Institute of Mechanical Engineers Automobile Division* , 1957, pp. 310-330.

115. Sheth, P. and J. Uicker. “IMP (Integrated Mechanicms Program), A Computer-aided Design Analysis System for Mechanisms and Linkage.” *Journal of Engineering for Industry* , 1972.

116. Singh, R.P., R.J. VanderVoort, C. Arduini, A. Festa, C. Maccone and D. Sciacovelli, “DCAP: An automated analysis and design tool for strucutural control of space structures.” *Second ESA Workshop on Mechanical Technology for Antennas - Proceedings of a Workshop held at ESTEC.,* Noordwijk, Netherlands, European Space Agency, (Special Publication) ESA SP 261, 1986.

117. Singh, R.P., R.J. VanderVoort and P.W. Likins. “Dynamics Of Flexible Bodies In Tree Topology - A Computer-Oriented Approach.” *Journal of Guidance, Control, and Dynamics* 8(5), 1985, pp. 584-590.

118. Steele, G.L.J. *Common Lisp: The Language.* 1984, Digital Press.

119. Stoer, J. and R. Bulirsch. *Introduction to Numerical Analysis.* 1980, Springer-Verlag. New York.

120. Striberski, A., P.S. Fancher, C.C. MacAdam and M.W. Sayers, “On Nonlinear Oscillations in Road Trains at High Forward Speeds.” *11th IAVSD Symposium of Vehicles on Roads and Tracks,* Kingston, Ontario, 1989.

121. Trom, J.D., J.L. Lopez and M.J. Vanderploeg. “Modeling a Mid-Size Passenger Car Using a Multibody Dynamics Program.” *ASME Journal of Mechanisms, Transmissions, and Automation in Design* 109(December), 1987, pp. 518-523.

122. Tzou, H.S., “Multibody nonlinear dynamics and controls of joint dominated flexible structures.” *Symposium on Robotics Presented at the Winter Annual Meeting of the American Society of Mechanical Engineers,* Chicago, IL, USA, ASME, 1988.

123. van Hulzen, J.A. and J. Calmet. “Computer Algebra Systems.” *Computer Algebra Symbolic and Algebraic Computation.* B. Buchberger, G. E. Collins, R. Loos and R. Albrecht ed., 1982, Springer-Verlag, Wien. 221-243.

124. Walker, M.W. and D.E. Orin. “Efficient Dynamic Computer Simulation of Robotic Mechanisms.” *Journal of Dynamic Systems Measurement and Control* 104(3), 1982, pp. 205-211.

125. Wampler, C.W., “Computer Methods in Manipulator Kinematics, Dynamics, and Control: A Comparative Study.” PhD thesis, Stanford, 1985.

126. Wang, J.T. and R.L. Huston. “Computational methods in constrained multibody dynamics: matrix formalisms.” *Computers and Structures* 29(2), 1988, pp. 331-338.

127. Wang, P.S., “Taking Advantage of Symmetry in the Automatic Generation of Numerical Programs for Finite Element Analysis.” *EUROCAL ’85 European Computer Algebra Conference Vol 2: Research Contributions,* Ed. B. F. Caviness. Lecture Notes in Computer Science. Linz, Austria, Springer-Verlag, 1985.

128. Wehage, R. and A. Shabana. “Application of Generalized Newton-Euler Equations and Recursive Projection Methods to Dynamics of Deformable Multibody Systems.” *Submitted to the ASME Journal of Mechanisms, Transmissions, and Automation in Design* , 1989, pp. 1-23.

129. Wehage, R.A., “Application of Matrix Partitioning and Recursive Projection to Order n Solution of Constrained Equations of Motion.” *20th Biennial ASME Mechanisms Conference,* Orlando, FLA, 1988.

130. Wehage, R.A., “Symbolic Factors of Linear System Coefficient Matrices for Tree-Structured Systems and their Efficient Solution.” *Seventh Army Conference on on Applied Mathematics and Computing,* West Point, New York, 1989.

131. Wehage, R.A. and E.J. Haug. “Dynamic Analysis of Mechanical Systems with Intermittent Motion.” *ASME Journal of Mechanical Design* 104(October), 1982, pp. 778-784.

132. Wehage, R.A. and E.J. Haug. “Generalized Coordinate Partitioning for Dimension Reduction in Analysis of Constrained Dynamic Systems.” *ASME Journal of Mechanical Design* 104(January), 1982, pp. 247-255.

133. Winkler, C.B. and M. Hagan. “A Test Facility for the Measurement of Heavy Vehicle Suspension Parameters.” *Transactions of Society of Automotive Engineers (SAE)* 89(paper 80096), 1980.

134. Wittenburg, J. *Dynamics of Systems of Rigid Bodies.* 1977, B.G. Teubner. Stuttgart.

135. Wittenburg, J., “Dynamics of Multibody Systems.” *Proceedings, XVth IUTAM/ICTAM Congress,* Toronto, 1980.

136. Wittenburg, J. “Analytical Methods in Mechanical System Dynamics.” *Computer Aided Analysis and Optimization of Mechanical System Dynamics*. E. G. Haug ed., 1984, Springer-Verlag, Heidelberg. 89-127.

137. Wittenburg, J. and U. Wolz, “MESA VERDE: A Symbolic Program for Nonlinear Articulated-Rigid-Body Dynamics.” *Proceedings of the 10th Design Engineering division Conference on Mechanical Vibration and Noise,* Cincinati, 1985.

138. Wolfram, S. *Mathematica*™. 1988, Adison-Wesley Publishing Company.

139. Wooff, C. and D. Hodgkinson. *muMATH: A microcomputer algebra system.* 1987, Academic Press. London.