Computational Mechanics 14 (1994) 277–287 © Springer-Verlag 1994

Object oriented dynamics simulator

A. S. Koh, J. P. Park

Abstract Multibody Dynamics and Object Oriented Programming are both leading technologies in their fields. This research combines them to produce a general purpose multibody dynamics code which is robust and easy to maintain. The simulator permits creation of mechanical systems from information specifying the properties and locations of parts, joints, springs, dampers, forces, gravity, etc. The mechanical components of the system are mirrored by software objects which interact to automatically assemble and integrate the equations of motion. This paper presents a multibody dynamics formulation suitable for object oriented programming, the principles of OOP and the procedures to create software objects with specific behaviors to represent parts, joints, forces springs, dampers etc. The aim is to demonstrate the use of OOP in serious engineering computing.

1
Introduction

Multibody Dynamics (MbD) and Object Oriented Programming (OOP) are leading technologies in mechanics and computer science respectively. But, to the best of the authors' knowledge, no one has produced a general purpose multibody dynamics simulator using OOP technology. A general dynamic simulator permits a designer to create his mechanical system by specifying the properties and states of masses, joints, springs, dampers, forces, gravity, etc. With OOP, the mechanical components of the system are mirrored by software objects which then automatically assemble and integrate the equations of motion. The ability to write software thinking of objects allows greater programming productivity. This is OOP's claim to fame over structured programming with procedural languages. This research implements an objected oriented dynamics simulator. The theory developed is general and can simulate an ensemble of rigid parts connected by any compatible combination of spherical, universal, revolute, cylindrical, planar, and translational joints, linear springs and dampers, and prescribed forces and motions (Fig. 1). Time histories of 3D motions, velocities, accelerations, forces and torques are possible outputs.

2
Multibody dynamics formulation

Multibody Dynamics (MbD) has been around since the time of Euler and Lagrange, but progress was slow until the 1960's. The aerospace program in the United States demanded a thorough understanding of MbD and the newly invented electronic computer provided the means for solution. The reader can discover the state of the art in some recently published books (Amirouche 1992; Huston 1990; Roberson and Schwertassek 1988; Schielen 1990; Shabana 1989). A good coverage of the interplay between MbD, controls and optimization is given by Haug (1983).

There are several methods of MbD formulation, but it is not known if any has a clear advantage over the others. The method in vogue uses relative coordinates and Kane's equations (equivalent to Jourdain's principle in Europe). The method produces a smaller matrix to solve, but the matrix is dense and required more work to obtain. By contrast, the earliest methods use absolute coordinates and Newton-Euler or Lagrange's equations. The resulting matrix is large but very sparse and easier to obtain. Presently,

Communicated by S. N. Atluri, 24 November 1993

A. S. Koh
School of Mechanical and Production Engineering,
Nanyang Technological University, Singapore 2263

J. P. Park
Department of Mechanical Engineering, Texas Tech University,
Lubbock TX 79409 USA

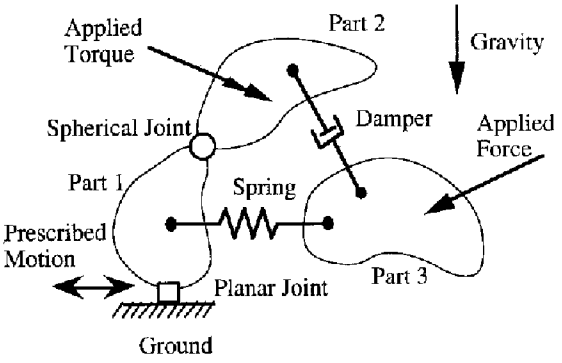

Fig. 1. Generic mechanical system

there are suggestions (Huston 1990) that the use of absolute coordinates and Kane's equations can be advantages. Whether this will highlight the earlier methods again remains to be seen.

The formulation chosen is the one developed by Orlandea et al. (1977) and Chace (1984). It is chosen because it forms the basis of the most successful and thoroughly tested dynamics modelers available. By using absolute coordinates, the parts, joints, etc. can contribute to the system matrix equation independently. Adding new and modifying old dynamic components are more safely done because of the modular nature of the formulation. In addition, the modular nature is better suited to OOP, and extensions to include controls, hydraulics, pneumatics, thermal and electromagnetic systems can be quite straight forward. Finally, with an eye towards using OODS for teaching, this formulation is perhaps more suitable than the compact methods because it is most intuitive to novices and intermediate users of MbD.

The starting point of the formulation is the familiar Lagrange's equations of the first kind:

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{q}_{i_q}}\right)-\frac{\partial T}{\partial q_{i_q}}-Q_{i_q}-\sum_{i_c=1}^{n_c}\frac{\partial G_{i_c}}{\partial q_{i_q}}\lambda_{i_c}=0 \tag{1}$$

for $i_q = 1, 2, \ldots, n_q$, where T is the kinetic energy of the system, q's are the generalized coordinates, Q's are the generalized forces, G's are the holonomic constraint equations, λ's are the Lagrange multipliers, and t is time. Nonholonomic constraints can be easily included but are not presented here for lack of space. To avoid any second derivatives with time, the generalized momentum p's are defined and substituted into Lagrange's equations.

$$p_{i_q}=\frac{\partial T}{\partial \dot{q}_{i_q}} \tag{2}$$

$$\dot{p}_{i_q}-\frac{\partial T}{\partial \dot{q}_{i_q}}-Q_{i_q}-\sum_{i_c=1}^{n_c}\frac{\partial G_{i_c}}{\partial q_{i_q}}\lambda_{i_c}=0. \tag{3}$$

2.1
Generalized coordinates

The q's chosen are the absolute coordinates X, Y, Z of the center of mass and the Euler parameters of the principal axes of each part. Euler parameters describe rotation θ about the physical unit vector $\vec{n}$ of the part wrt the inertial frame and are defined as follows

$$E_1=\vec{n}\cdot\vec{I}\sin\frac{\theta}{2}\quad E_2=\vec{n}\cdot\vec{J}\sin\frac{\theta}{2}\quad E_3=\vec{n}\cdot\vec{K}\sin\frac{\theta}{2}\quad E_4=\cos\frac{\theta}{2} \tag{4}$$

where $\vec{I}, \vec{J}, \vec{K}$ = unit Cartesian base vectors in the inertial frame. The Euler parameters are not independent and must satisfy

$$E_1^2+E_2^2+E_3^2+E_4^2=1. \tag{5}$$

Euler parameters are chosen over Euler angles because they do not cause singularities in expressions affecting angular velocities. But they do contribute one constraint equation for each part while Euler angles would not. This trade off is justified because more programming is necessary to avoid singularities while the additional constraints can be added to the inevitable constraints introduced by joints or prescribed motions and treated similarly.

2.2
Rotation matrix

If $\vec{i}, \vec{j}, \vec{k}$ = unit Cartesian base vectors in a part frame, then rotation matrix [A] for part i_p is defined by

$$\begin{Bmatrix} \vec{I} \\ \vec{J} \\ \vec{K} \end{Bmatrix} = [A]_{i_p} \begin{Bmatrix} \vec{i} \\ \vec{j} \\ \vec{k} \end{Bmatrix}_{i_p} \tag{6}$$

$$[A]_{i_p} = \begin{bmatrix} 1-2E_2^2-2E_3^2 & 2(E_1E_2-E_3E_4) & 2(E_1E_3+E_2E_4) \\ 2(E_1E_2+E_3E_4) & 1-2E_1^2-2E_3^2 & 2(E_2E_3-E_1E_4) \\ 2(E_1E_3-E_2E_4) & 2(E_2E_3+E_1E_4) & 1-2E_1^2-2E_2^2 \end{bmatrix}_{i_p}. \tag{7}$$

The rotation matrix and its partial derivatives wrt to E_1, E_2, E_3, E_4 and time are heavily used. Note that every part can be treated identically.

2.3
Angular velocity

The angular velocity vector for part i_p is found to be

$$\vec{\omega}_{i_p} = \omega_{i_px}\vec{i}_{i_p} + \omega_{i_py}\vec{j}_{i_p} + \omega_{i_pz}\vec{k}_{i_p} \tag{8}$$

$$\begin{Bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{Bmatrix}_{i_p} = 2 \begin{bmatrix} E_4 & E_3 & -E_2 & -E_1 \\ -E_3 & E_4 & E_1 & -E_2 \\ E_2 & -E_1 & E_4 & -E_3 \end{bmatrix}_{i_p} \begin{bmatrix} \dot{E}_1 \\ \dot{E}_2 \\ \dot{E}_3 \\ \dot{E}_4 \end{bmatrix}_{i_p} \tag{9}$$

$$\boldsymbol{\omega}_{i_px} = [B]_{i_p}\dot{\mathbf{q}}_{Ei_p}. \tag{10}$$

A vector with an overstrike arrow is a physical vector, otherwise it is a components vector.

2.4
Kinetic energy

The system is assumed to be made up of rigid bodies only. If m = mass, $\vec{v}$ = velocity of part, $\vec{\vec{J}}$ = mass moment of inertia dyadic about center of mass of a part, the kinetic energy is

$$T = \sum_{i_p=1}^{n_p} \frac{1}{2} m_{i_p} \vec{v}_{i_p} \cdot \vec{v}_{i_p} + \sum_{i_p=1}^{n_p} \frac{1}{2} \boldsymbol{\omega}_{i_p} \cdot \vec{\vec{J}}_{i_p} \cdot \boldsymbol{\omega}_{i_p} \tag{11}$$

n_p is the total number of parts in the system. The specific form of Eq. (11) chosen is

$$T = \sum_{i_p=1}^{n_p} \frac{1}{2} m_{i_p}(\dot{X}^2 + \dot{Y}^2 + \dot{Z}^2)_{i_p} + \sum_{i_p=1}^{n_p} \frac{1}{2}(J_{xx}\omega_x^2 + J_{yy}\omega_y^2 + J_{zz}\omega_z^2)_{i_p} \tag{12}$$

where J_{xx}, J_{yy}, J_{zz} are the principal mass moment of inertias of the part. The first and second partial derivatives of T wrt to $\mathbf{q}$ and $\dot{\mathbf{q}}$ are heavily used.

2.5
Generalized force

The generalized forces associated with the q's is derived from virtual work

$$\begin{aligned} \delta W &= \sum_{i_f=1}^{n_f} \vec{F}_{i_f} \cdot \delta \vec{r}_{i_f} + \sum_{i_t=1}^{n_t} \vec{T}_{i_t} \cdot \delta \vec{\theta}_{i_t} \\ &= \sum_{i_q=1}^{7n_p} \left(\sum_{i_f=1}^{n_f} \vec{F}_{i_f} \cdot \frac{\partial \vec{r}_{i_f}}{\partial q_{i_q}} + \sum_{i_t=1}^{n_t} \vec{T}_{i_t} \cdot \frac{\partial \vec{\theta}_{i_t}}{\partial q_{i_q}} \right) \delta q_{i_q} = \sum_{i_q=1}^{7n_p} Q_{i_q} \delta q_{i_q} \end{aligned} \tag{13}$$

where $\delta\vec{r}_{i_f}$, $\delta\vec{\theta}_{i_t}$ are the virtual displacement of the force $\vec{F}_{i_f}$ and virtual rotation of the torque $\vec{T}_{i_t}$, respectively. More specifically, the generalized forces are found to be

$$\begin{Bmatrix} Q_X \\ Q_Y \\ Q_Z \end{Bmatrix}_{i_p} = \sum_{i_f}^{\substack{\text{forces} \\ \text{on } i_p}} \begin{Bmatrix} F_X \\ F_Y \\ F_Z \end{Bmatrix}_{i_f} \tag{14}$$

$$\begin{Bmatrix} Q_{E_1} \\ Q_{E_2} \\ Q_{E_3} \\ Q_{E_4} \end{Bmatrix}_{i_p} = \sum_{i_f}^{\substack{\text{forces} \\ \text{on } i_p}} [\, F_X \quad F_Y \quad F_Z \,]_{i_f} \begin{Bmatrix} \dfrac{\partial [\mathbf{A}]_{i_p}}{\partial E_{1i_p}} \\ \dfrac{\partial [\mathbf{A}]_{i_p}}{\partial E_{2i_p}} \\ \dfrac{\partial [\mathbf{A}]_{i_p}}{\partial E_{3i_p}} \\ \dfrac{\partial [\mathbf{A}]_{i_p}}{\partial E_{4i_p}} \end{Bmatrix} \begin{Bmatrix} x \\ y \\ z \end{Bmatrix}_{i_m i_f} + \sum_{i_t}^{\substack{\text{torques} \\ \text{on } i_p}} [\mathbf{B}]_{i_p}^T [\mathbf{A}]_{i_p}^T \begin{Bmatrix} T_X \\ T_Y \\ T_Z \end{Bmatrix}_{i_t} \tag{15}$$

where xyz is the location of the point of application of force in principal body coordinates. The indices i_m, i_p identify the marker and part on which the force i_f or torque i_t is applied. The most commonly used forces are prescribed forces, springs, and dampers. This true for torques too.

2.6
Specific forces

Some commonly used forces are constant gravity:

$$\vec{F}_{gi_p} = m_{i_p}\vec{g}. \tag{16}$$

Spring tension force on marker I:

$$\vec{F}_{sI} = f_s(r_{IJ} - r_0)\frac{\vec{r}_{IJ}}{r_{IJ}} \tag{17}$$

where $\vec{r}_{IJ}$ is the position vector from marker I to marker J. The opposite force is on marker J. Clearly, $f_s(.)$ can also be nonlinear.

Dashpot tension force on marker I:

$$\vec{F}_{dI} = f_d(\dot{r}_{IJ})\frac{\vec{r}_{IJ}}{r_{IJ}}. \tag{18}$$

Prescribed forces:

$$\vec{F}_{XYZ}(t) = F_X(t)\vec{I} + F_Y(t)\vec{J} + F_Z(t)\vec{K} \tag{19}$$

$$\vec{f}(t) = f_x(t)\vec{i}_{Im} + f_y(t)\vec{j}_{Im} + f_z(t)\vec{k}_{Im}. \tag{20}$$

Spring torque on marker I about its z axis:

$$\vec{T}_{sI} = t_s(\theta_{IJ} - \theta_0)\vec{k}_{Im}. \tag{21}$$

Dashpot torque on marker I about its z axis:

$$\vec{T}_{dI} = t_d(\dot{\theta}_{IJ})\vec{k}_{Im}. \tag{22}$$

Prescribed torques:

$$\vec{T}(t) = T_X(t)\vec{I} + T_Y(t)\vec{J} + T_Z(t)\vec{K} \tag{23}$$

$$\vec{t}(t) = t_x(t)\vec{i}_{Im} + t_y(t)\vec{j}_{Im} + t_z(t)\vec{k}_{Im}. \tag{24}$$

Other forces like fluid dynamic, tire, electromagnetic, variable gravitational forces are derivable and can be incorporated.

2.7 Joint constraints

The six most commonly used joints are shown Fig. 2. They can be simulated by specifying combinations of three types of holonomic constraints only.

Point constraint

$$\vec{\mathbf{r}}_{IJ} = \vec{0}. \tag{25}$$

Plane constraint

$$\vec{\mathbf{r}}_{IJ} \cdot \vec{\mathbf{i}}_{Im} = 0. \tag{26}$$

Perpendicular constraint

$$\vec{\mathbf{k}}_{Im} \cdot \vec{\mathbf{i}}_{Jm} = 0. \tag{27}$$

The combination of constraints to model the different joints are shown in Fig. 2. Other combinations are also possible.

3 Numerical solution

Grouping the set of equations from Eq. (3) as

$$\mathbf{M}(\mathbf{q}, \dot{\mathbf{q}}, \dot{\mathbf{p}}, \mathbf{Q}, \lambda) = \mathbf{0} \tag{28}$$

from Eq. (2) as

$$\mathbf{p}(\mathbf{q}, \dot{\mathbf{q}}, \mathbf{p}) = \mathbf{0} \tag{29}$$

from Eqn. (14), (15) as

$$\mathbf{N}(\mathbf{q}, \dot{\mathbf{q}}, \mathbf{Q}, t) = \mathbf{0} \tag{30}$$

from Eq. (5), (25), (26), (27) as

$$\mathbf{G}(\mathbf{q}, t) = \mathbf{0} \tag{31}$$

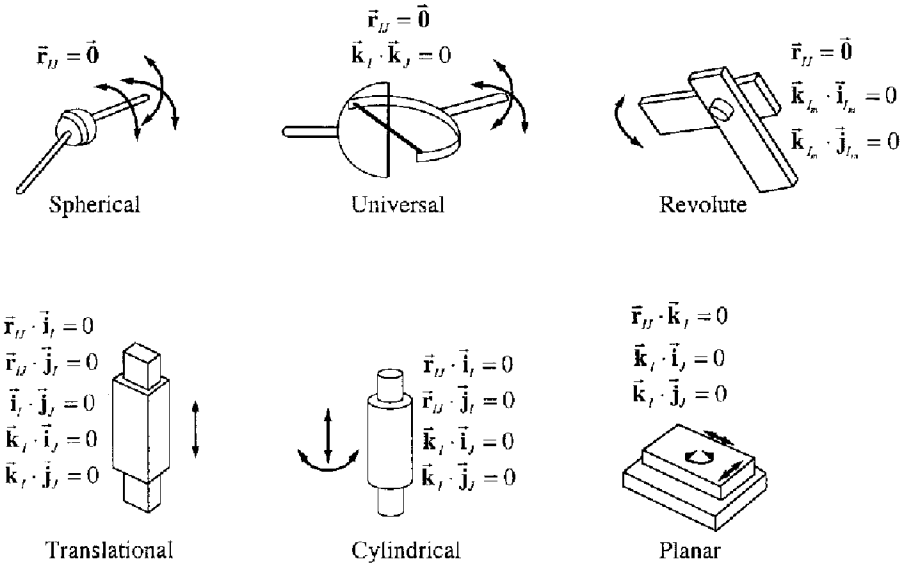

Fig. 2. Types of joints

result in a system of differential-algebraic equations which are coupled and nonlinear. To convert the first order derivatives to algebraic terms, the backward difference formulae for step size h are used:

$$\dot{\mathbf{q}}(t) = \frac{1}{h\beta}(\alpha_0 \mathbf{q}(t) + \alpha_1 \mathbf{q}(t-h) + \cdots + \alpha_i \mathbf{q}(t-ih)) + O(h^i) \tag{32}$$

$$\dot{\mathbf{p}}(t) = \frac{1}{h\beta}(\alpha_0 \mathbf{p}(t) + \alpha_1 \mathbf{p}(t-h) + \cdots + \alpha_i \mathbf{p}(t-ih)) + O(h^i) \tag{33}$$

where β and α's for various order i are known. These formulae have been chosen because they are very stable and allow larger time steps to be taken (Gear 1971). They are however implicit because unknowns at time t occur on both sides of the equations.

After substitution for the first derivatives, the system of equations become a set of nonlinear algebraic equations. These equations are then solved using the Newton–Raphson method.

$$\begin{bmatrix} \left[\frac{\partial \mathbf{M}}{\partial \mathbf{q}} + \frac{\partial \mathbf{M}}{\partial \dot{\mathbf{q}}}\frac{\partial \dot{\mathbf{q}}}{\partial \mathbf{q}}\right] & \left[\frac{\partial \mathbf{M}}{\partial \dot{\mathbf{p}}}\frac{\partial \dot{\mathbf{p}}}{\partial \mathbf{p}}\right] & \left[\frac{\partial \mathbf{M}}{\partial \mathbf{Q}}\right] & \left[\frac{\partial \mathbf{M}}{\partial \boldsymbol{\lambda}}\right] \\ \left[\frac{\partial \mathbf{P}}{\partial \mathbf{q}} + \frac{\partial \mathbf{P}}{\partial \dot{\mathbf{q}}}\frac{\partial \dot{\mathbf{q}}}{\partial \mathbf{q}}\right] & \left[\frac{\partial \mathbf{P}}{\partial \mathbf{p}}\right] & [\mathbf{0}] & [\mathbf{0}] \\ \left[\frac{\partial \mathbf{N}}{\partial \mathbf{q}} + \frac{\partial \mathbf{N}}{\partial \dot{\mathbf{q}}}\frac{\partial \dot{\mathbf{q}}}{\partial \mathbf{q}}\right] & [\mathbf{0}] & \left[\frac{\partial \mathbf{N}}{\partial \mathbf{Q}}\right] & [\mathbf{0}] \\ \left[\frac{\partial \mathbf{G}}{\partial \mathbf{q}}\right] & [\mathbf{0}] & [\mathbf{0}] & [\mathbf{0}] \end{bmatrix}_t \begin{Bmatrix} \Delta \mathbf{q} \\ \Delta \mathbf{p} \\ \Delta \mathbf{Q} \\ \Delta \boldsymbol{\lambda} \end{Bmatrix}_t = - \begin{Bmatrix} \mathbf{M}(\mathbf{q}, \dot{\mathbf{q}}(\mathbf{q}), \dot{\mathbf{p}}(\mathbf{p}), \mathbf{Q}, \boldsymbol{\lambda}) \\ \mathbf{P}(\mathbf{q}, \dot{\mathbf{q}}(\mathbf{q}), \dot{\mathbf{p}}) \\ \mathbf{N}(\mathbf{q}, \dot{\mathbf{q}}(\mathbf{q}), \mathbf{Q}) \\ \mathbf{G}(\mathbf{q}, t) \end{Bmatrix}_t \tag{34}$$

where the Δ's are the corrections to the initial guesses to satisfy the nonlinear equations. The converged solution is the state of the system at give time t.

To march in time, the state at $(t + dt)$ is extrapolated from past values and the above procedure is used to correct that solution. The process is repeated till the desired simulation duration is reached. The extrapolation algorithm to predict new values is

$$\mathbf{q}(t) = \gamma_1 \mathbf{q}(t-h) + \gamma_2 \mathbf{q}(t-2h) + \cdots + \gamma_i \mathbf{q}(t-ih) \tag{35}$$

where γ's for various order i are known. Extrapolation is similarly used for p, Q and λ.

4
Object oriented programming

The guiding principle for Object Oriented programming (OOP) is that the language and its supporting environment should simplify the programming task rather than emphasize computational efficiency. Fortunately, improvements in hardware has allowed this principle to be followed without sacrificing response time. In OOP, software is viewed as a collection of objects with individual properties and behaviors. Performing a task, therefore, involves orchestrating the objects by message passing among objects. Even the user is viewed as an object. This paradigm is good for humans because our life experiences have trained us to understand situations in this manner (LaLonde 1991; Savic 1990).

Of the many object oriented programming languages, Smalltalk (Goldberg 1989; Objectworks 1991) is chosen because it is the most mature OOP environment. Smalltalk is the original and pure OOP language. Its purity prevents programmers from reverting to procedural programming methods which are always tempting in hybrid languages like C++ or Object Pascal. Very importantly, Smalltalk is a rich environment with its own windows, editors, debuggers, browsers, file handlers, backups and graphics. It is also a database of much of the useful objects and codes developed since the 1970's and the user just reuses as much as possible. Whatever that is not available, the user adds to the database in a very safe manner. New objects are created so that they inherit properties and behaviors from available objects without affecting the latter. This encourages reuse. Behaviors that are different can be added to supersede inherited behaviors, and new behaviors can be created anew. The database helps the programmer to understand the relationships between the objects in many ways. Objects with similar properties and behaviors are grouped into classes. Related classes are grouped into class categories. The inheritance tree of the classes are well defined and is visible upon request. An object has behaviors which usually requires the assistance

of other objects for successful execution. The database is excellent at showing the cross-referencing of objects and their properties and behaviors. Finally, this environment is identical from system to system, so the user is shielded from hardware and operating system changes. All these features of Smalltalk enhances programming producitivity many folds.

5
Smalltalk implementation

In procedural programming, the code is essentially a monologue between the programmer and the computer (Fig. 3). Upon execution, the computer speaks in terse and low level terms. It is the programmer's duty to make the computer understand everything. In OOP, however, the programmer can talk to all the objects, real or abstract, in his problem (Fig. 3). He makes each object understand its own properties and behaviors and from then on views it as any ordinary day object unrelated to computing. In addition, the objects talk to one another. In fact, every action is initiated by sending a message to an object. The statement syntax is as follows:


```
anObject  aMessage.
anObject  aMessageKeyword:  anArgumentObject.
anObject  aMsgKey1:  anArgObj1 aMsgKey2:  anArgObj2.
```

where **anObject, anArgumentObject, anArgObj1, anArgObj2** are any noun and **aMessage, aMessageKeyword, aMsgKay1, aMsgKey2** are any command. Statements can be extended by appending keyword-argument pairs. The result of each statement is always an object which can in turn be given another message or be assigned to a variable.

An object responds to a message by looking for a method to respond. A method is a set of instructions associated with a message. If the object finds a method, it will execute the instructions which consist of sending messages to other objects in a coordinated manner to accomplish the task. The method always returns an object as the finally answer. The method may also modify the internal states of the object itself. If, however, no method is found, the object will simply say it does not understand the message. Examples of OODS objects message statements are

```
System  checkCompatibilty.
System  simulateUntil:  10.0  inStepsOf: 0.05.
```

With almost complete freedom to pick object names and message words, it is very easy to develop a meaningful language for problem solution. Even numbers are considered as objects and the arithmetic symbols as messages. This simple syntax also allows code and input to be treated uniformly. The input of a dynamics problem for simulation is also the code to be executed. There is no need for a parser to read an input file.

In the OODS implementation, **System** is the object representing the system, say in Fig. 4. Other objects are the **ground, mass, spring, damper** and **appliedForce**. Less obvious are the **wallMarker, massMarker, baseMarker, floorMarker** and **floorJoint**. The system is always in 3D, so **baseMarker** and **floorMarker** are used by a translational joint (**floorJoint**) to restrict the mass to 1D motion. Examples of some conversations include **System** telling **mass, spring, damper** to fill in the system matrix. **mass** contributes to all elements in the Jacobian matrix and error vector which have the kinetic energy term. This is done independently of other system components. **Spring**, in turn, asks **wallMarker** and **massMarker** to return the distance between them. **massMarker** asks **mass** for its position and transformation matrix

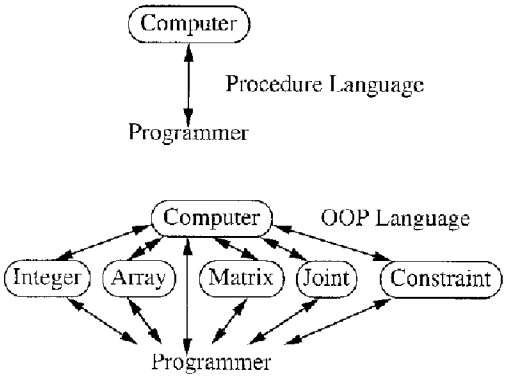

Fig. 3. Computing dialogues

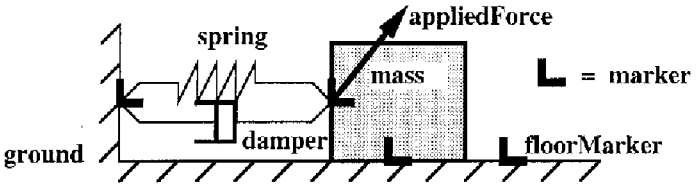

Fig. 4. Spring mass system

before computing the distance to **wallMarker**. **spring** then computes the tension and contributes to all Jacobian and error vector elements containing the generalized forces associated with **mass**.

It should now be apparent that object oriented programming involves giving each object the methods to respond to various messages and executing a series of object-message statements to accomplish useful computing. Notice that an object has its own data and methods which can be accessed and modified only through messages. Variables and procedures belong to someone and are not thought as memory locations. This useful property of having clear boundaries is called encapsulation (Fig. 5). The object **mass** has properties like mass, moment of inertias, position, etc. It has methods to compute its transformation matrix, contribute to the system equations, etc. The clear boundaries also allow the same message to be sent to different objects without risk of ambiguity. Each object will use its own method to respond correctly. For example: **mass**, **spring** and **damper** receive the message **updateInSimulation** after each iteration. This property of OOP is called polymorphism and is very natural to human experience.

When there are many objects which have similar properties and behaviors, it is beneficial to define these objects in one place called a class. Each member object is called an instance of the class. Each instance can set its properties to specific values and respond to methods common to the class appropriately. For example: **mass** is an instance of the class **Part**. If a two mass system had been chosen, **mass2** would be another instance of **Part**. Realizing the utility and convenience of classes, it is customary to create classes first and request the classes to create instances. For consistency, classes are objects too.

Figure 6 shows the definitions of **System** class and **Part** class in OODS. Note that they are actually object-message-argument statements to a superclass to spawn a new subclass. The second argument is a list of instance variables that each instance (member) of the class will have. Each instance variable stores the value of a property for that instance and is private to that instance. For example **mass** is an instance of **Part** and has the value 1 stored in 'partNo', **mass2** is another instance of **Part** and has value 2 stored in 'partNo'. The third argument is a list of class variables which are accessible by all instances of the class. There is only one copy of the class variables because there is only one copy of a particular class.

The instance and class variables for **System** and **Part** are quite self explanatory and can be easily inspected within the Smalltalk environment. 'Alpha Beta Gamma' store the backward difference and interpolation coefficients for various orders 'MaxOrder MethodOrder'. 'DeltaVector ErrorVector Jacobian'

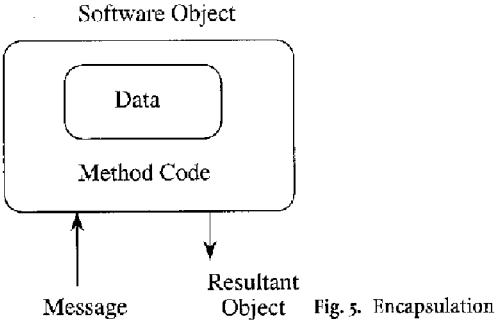

Fig. 5. Encapsulation

```
Object subclass: #System
        instanceVariableNames: 'name '
        classVariableNames: 'Alpha Beta Dampers DeltaVector ErrorVector Forces Gamma H
Jacobian Joints MaxOrder MethodOrder NumberOfConstraints NumberOfCoordinates OutStream
Parts PpdotPp PqdotPq SimTime Springs Tmax Tolerance '

System subclass: #Part
        instanceVariableNames: 'partNo outStream m qX qE qXold qEold qXdot qEdot qXdotOld
qEdotOld qXddot qEddot wqX wqE wqXdot wqEdot pX pE pXold pEold pXdot pEdot pXdotOld
pEdotOld aQX aQE nvec theta aJ aA aAdot pApE aB aBdot aCX aCE aCEdot pTpE ppTpEpE
ppTpEpEdot leu leuOld aGeu pGeupE omega '
        classVariableNames: 'PpGeupEpE '
```

Fig. 6. Class definition

```
correct
        "System correct."

        | n errorNorm |                          "Temporary variables."
        n := 3 * NumberOfCoordinates + NumberOfConstraints.
        Jacobian := Matrix new: n by: n.         "Creating a new matrix."
        ErrorVector := Vector new: n.            "Creating a new vector."

        "Iterate for solution."
        [Jacobian zeroSelf.                                "Clear matrix."
        ErrorVector zeroSelf.
        Parts do: [:part | part fillSimulationEqn].        "Fill simulation eqn."
        Joints do: [:joint | joint fillSimulationEqn].
        Forces do: [:force | force fillSimulationEqn].

        DeltaVector := Jacobian solveWith: ErrorVector.

        Parts do: [:part | part updateInSimulation].       "Update."
        Joints do: [:joint | joint updateInSimulation].
        Forces do: [:force | force updateInSimulation].

        errorNorm := ErrorVector l2norm.                   "Check convergence."
        errorNorm > Tolerance] whileTrue
```

Fig. 7. Method definition

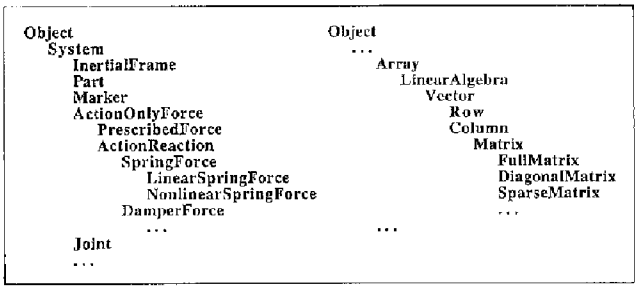

Fig. 8. Class hierarchy for multibody dynamics and linear algebra

are used to represent Eq. (34). 'H' is the time step. 'Parts Joints Forces Springs Dampers' are collections of all the respective items in the system. 'OutStream' is for output to an external file. 'PpdotPq PqdotPq' represent $\frac{\partial \dot{\mathbf{p}}}{\partial \mathbf{q}}, \frac{\partial \dot{\mathbf{q}}}{\partial \mathbf{q}}$ respectively. For **Part**, 'qX qE... qEddot' store the current and old values of the generalized coordinates and their time derivatives for each part. Similarly, 'pX... pEdotOld' store the generalized momenta. 'aQX aQE' store the generalized forces. 'aA aB' are the matrices defined in Eq. (7), (10). 'pTpE... ppTpEpE' are the first and second derivatives of kinetic energy wrt to Euler parameters. 'leu...' store the Lagrange multiplier for the Euler parameter constraint (Eq. 5). 'PpGeupEpE' stores the second partial derivatives of the Euler parameter constraint. It is a class variable because it is constant and can therefore be shared by all parts.

Figure 7 shows the definition of a typical method. This method belongs to System and is executed whenever the statement

System correct.

is encountered. This method carries out the corrector step of the Gear algorithm. In brief, the parts, joints and forces are asked to contribute to the system equations (Eq. 34) using the initial guesses. The matrix equation is solved to give the correction. The parts, joints and forces are told to update themselves with the correction. A check for convergence is done, and if the error is small, the method is exited, otherwise the cycle is repeated. Note that the programmer is having a dialogue with many objects rather than with the computer only.

Just as a newly created object acquires the properties and behaviors from its class, it also acquires the properties and behaviors from all its superclasses. This is the concept of class hierarchy and inheritance. When classes are created, they are attached to a family tree of classes, the root of which is **Object**. The trick is to sprout a new class from a point where it inherits as many useful properties and behaviors as possible while avoiding useless ones. This encourages reusability and reduces new coding. For example: **System** sprouts from **Object** since there is negligible MbD in Smalltalk, but **LinearAlgebra** inherits from **Array** and its superclasses which are well established classes in Smalltalk. Figure 8 shows some of the classes added to the Smalltalk environment to solve general multibody dynamics. For example, 'name', defined in **System**, is inherited by all parts, etc. That is, every object of the system will have a variable to store its name.

```
runDoublePendulum
        "System runDoublePendulum."

        | ceiling bar barMarker barMarker2 bar2 bar2Marker ceilingMarker
                ballJoint ball2joint torSpr gravity |          "Temporary variables."

        System start.
        SimTime := 0.0.                    "Initial time."
        Tolerance := 1.0d-20.    "Iteration stopping criterion."
        MaxOrder := 6.           "Backward difference order."

        ceiling := InertialFrame new.              "Creating a new object."
        ceiling name: #ceiling.
        ceilingMarker := Marker new.               "Creating a new object."
        ceilingMarker name: #ceilingMarker.
        ceilingMarker
                isOn: ceiling                      "SI units."
                xyz: #(0.0 0.0 0.0 )               "Position wrt ceiling."
                e1234: #(0.0 0.0 0.0 1.0d ).       "Orientation wrt ceiling."

        bar := Part new.
        bar name: #bar.
        bar mass: 1.0d Jxx: 8.33d-2 Jyy: 5.0d-5 Jzz: 8.33d-2.    "Inertias."
        bar     qX: #(0.0 -0.5d 0.0 )              "Initial position."
                qE: #(0.0 0.0 0.0 1.0d )           "Initial orientation."
                qXdot: #(5.0d 0.0 0.0 )            "Initial velocity."
                qEdot: #(0.0 0.0 5.0d 0.0 ).       "Initial angular velocity."
        barMarker := Marker new.
        barMarker name: #barMarker.
        barMarker
                isOn: bar
                xyz: #(0.0 0.5d 0.0 )              "Position wrt c.m."
                e1234: #(0.0 0.0 0.0 1.0d ).       "Orientation wrt c.m."
        barMarker2 := Marker new.
        barMarker2 name: #barMarker2.
        barMarker2
                isOn: bar
                xyz: #(0.0 -0.5d 0.0 )
                e1234: #(0.0 0.0 0.0 1.0d ).

        bar2 := Part new.
        bar2 name: #bar2.
        bar2 mass: 1.0d Jxx: 8.33d-2 Jyy: 5.0d-5 Jzz: 8.33d-2.
        bar2    qX: #(0.0 -1.5d 0.0 )
                qE: #(0.0 0.0 0.0 1.0d )
                qXdot: #(15.0d 0.0 0.0 )
                qEdot: #(0.0 0.0 5.0d 0.0 ).
        bar2Marker := Marker new.
        bar2Marker name: #bar2Marker.
        bar2Marker
                isOn: bar2
                xyz: #(0.0 0.5d 0.0 )
                e1234: #(0.0 0.0 0.0 1.0d ).

        ballJoint := SphericalJoint new.
        ballJoint name: #ballJoint.
        ballJoint connectsI: ceilingMarker toJ: barMarker.
        ball2joint := SphericalJoint new.
        ball2joint name: #ball2joint.
        ball2joint connectsI: barMarker2 toJ: bar2Marker.

        torSpr := TorsionalSpring new.
        torSpr
                name: #torSpr
                connectsI: barMarker
                toJ: ceilingMarker
                stiffness: 10.0d                   "Nm/rad."
                theta0: 0.0
                thetaInit: 0.0.                    "pre-stressing."

        gravity := ConstantGravity new.
        gravity name: #gravity.
        gravity acceleration: #(0.0 -9.81d 0.0 ).  "m/s*s."

        bar outputStates.
        bar2 outputStates.
        ballJoint outputStates.
        ball2joint outputStates.

        System checkCompatibility.
        System simulateUntil: 2.0d inStepsOf: 0.01d.    "second."
        System closeOutputs
```

Fig. 10. Program for double pendulum

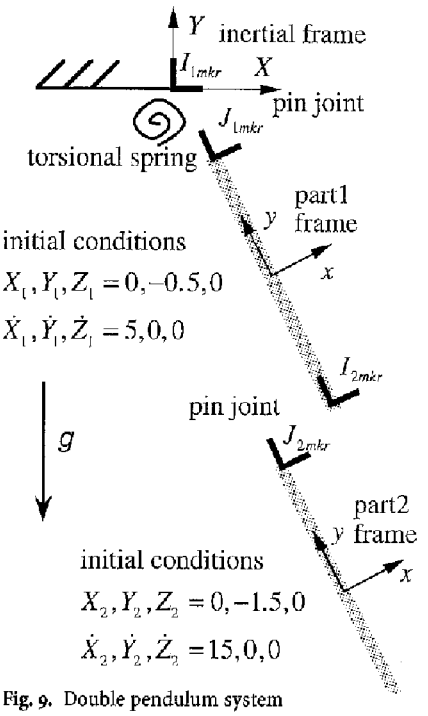

Fig. 9. Double pendulum system

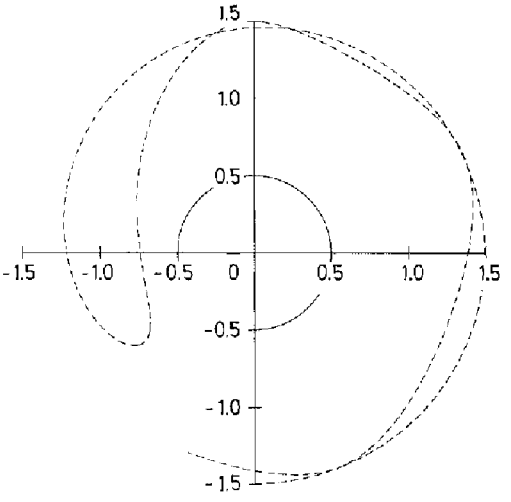

Fig. 11. Loci of center of mass for double pendulum

6
Program example

OODS is a full 3D general purpose dynamics simulator, but the simple example in Fig. 9 is used to demonstrate the tasks and input needed. The system is a double pendulum with spherical joints, and a torsional spring in the XY plane is at the top joint. Gravity acts downwards. The bars are uniform slender rods of length 1 m, radius 10 mm and mass 1 kg. Initially, the pendulums are vertical and swings like a rigid body about the ceiling joint with angular velocity 10 rad/s. The torsional spring exerts no torque when the top bar is hanging down vertically. Figure 10 shows the commented program input. Note that it is actually a method which is executed by the statement

```
System runDoublePendulum.
```

The simulation is carried out for 1.8 s and the loci of the center of mass are shown in Fig. 11. The full nonlinear nature of the whiplash response is clearly demonstrated. All the kinematics and dynamics information are available as output but will not be shown for lack of space.

7 Conclusions

An OOP implementation of a very general multibody dynamics simulator has been achieved on a 8 MB RAM 80486 computer using Objectworks/Smalltalk running under Windows 3.1. The formulation above is chosen because of its simplicity and modularity. A part contributes to the system equations independently of other parts, unlike formulations using relative coordinates. A joint depends only on the two adjoining parts similarly with any spring or damper. A force depends only on the part it acts on. The independent manner in which system components contribute to the system equations makes an object oriented design very suitable. The software objects are simple to understand, implement and modify. Smalltalk has provided enhanced programming productivity and has allowed the software to mirror the mechanical world more easily. It will be easy to extend OODS by creating new objects to model flexible bodies, variables mass bodies, tires, rolling contact, etc. Old objects can have new methods to animate themselves, check space obstructions, and optimize their own dimensions. This freedom to modify and extend a software safely is a major feature of OOP. To the authors' minds, creating the dynamics modeler has confirmed the advantages of OOP or, more specifically, Smalltalk.

References

Amirouche, F. M. L. 1992: Computational methods in multibody dynamics. Prentice-Hall, USA
Chace, M. A. 1984: Methods and experience in computer aided design of large-displacement mechanical systems. In: Haug, E. J. (ed.): Computer aided analysis and optimization of Mechanical System Dynamics, pp. 233–259. Springer-Verlag, Berlin
Gear, C. W. 1971: Simultaneous numerical solution of differential-algebraic equations. IEEE Trans. Circuit Theory CT-18 # 1, pp. 89–95
Goldberg, A. 1989: Smalltalk 80: the language. Addison Wesley, USA
Haug, E. J. 1984: Computer aided analysis and optimization of mechanical system dynamics. Springer-Verlag, Berlin
Huston, R. L. 1990: Multibody dynamics. Butterworth-Heinemann, Stoneham, MA, USA
LaLonde, W. R. 1991: Inside Smalltalk, Prentice Hall, USA
Objectworks\Smalltalk, 1991: Release 4, User's Guide, ParcPlace Systems, USA
Orlandea, N.; Chace, M. A.; Calahan, D. A. 1977: A sparsity-oriented approach to the dynamics analysis and design of mechanical systems, Part 1. J. Engrg. Ind. 99: 773–784
Roberson, R. E.; Schwertassek, R. 1988: Dynamics of multibody systems. Springer-Verlag, Berlin
Savic, D. 1990: Objected oriented programming with Smallertalk/V, Ellis Horwood, USA
Schielen, W. 1990: Handbook of Multibody Dynamics. Springer-Verlag, Berlin
Shabana, A. A. 1989: Dynamics of Multibody Systems. Wiley-Interscience, USA