This is to certify that the

thesis entitled

THE DEVELOPMENT OF A KINEMATICS SOLVER BASED ON OBJECT ORIENTED PROGRAMMING PRINCIPLES

presented by

Jiyoung Sung

has been accepted towards fulfillment
of the requirements for

M.S. degree in Mechanical Engineering

Joseph W. Whitesell
Major professor

Date November 7, 1989

The Development of a Kinematics Solver Based on Object-Oriented Programming Principles

By

Jiyoung Sung

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Mechanical Engineering

1989

ABSTRACT

The Development of a Kinematics Solver Based on Object-Oriented Programming Principles

By

Jiyoung Sung

The methods of object-oriented programming are used to develop a multibody kinematics solver. It appears that several benefits can result from this approach. These include improved reliability, incremental capability, readability, and flexibility. To demonstrate how these benefits can be obtained through the object-oriented programming principles, a class hierarchy which describes the kinematic elements in terms of objects has been designed. From this class hierarchy, two specific mechanisms have been created and shown as examples: the first one is a four bar mechanism and the other is a mechanism which combines a four bar mechanism and a slider crank mechanism. Also shown in the examples is how much simulation process can be simplified through the object-oriented programming principles.

ACKNOWLEDGEMENTS

I would like to express sincere gratitude to my advisor, Prof. J. Whitesell, for his invaluable guidance and encouragement during the course of this work as well as patient review of the thesis and extraordinary help for the preparation of the presentation. I also would like to express my appreciation to professors R. Rosenberg and A. Diaz for being members of committee in such a short notice and also for encouraging comments on the presentation.

Of course, I owe a great deal to my family. Especially I would like to express the most sincere gratitude to my father and mother. Without their support and understanding, it would have been impossible to start my graduate life at MSU in the first place.

I would like to thank many Korean students in Mechanical Engineering Dept. and friends at MSU for their support and friendship during the years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In representing a way of thinking and a methodology for computer programming, object-oriented programming(OOP) takes a quite different path compared with the one taken by the conventional structured high level programming languages.

According to Pascoe[22], an object-oriented language should formally support abstraction, encapsulation, inheritance, and polymorphism. Complete definitions of these principles are given in chapter 2. He claims that data abstraction and encapsulation increases reliability and helps decouple procedural and representational specification from implementation. Polymorphism increases flexibility by permitting the addition of new classes of object without having to modify existing code. Inheritance coupled with polymorphism allows code to be reused and this reduces overall code bulk. By reducing the size of code, object-oriented programming provides major advantages in the production and maintenance of software: shorter development time, a high degree of code sharing, and flexibility.[22]

Based on all of these advantages stated so far, object-oriented programming claims improved programmer productivity and easy program maintenance.[2,21]

The objective of this thesis is to explore and demonstrate the effectiveness of the object-oriented programming and its programming environment for building engineering analysis programs. The object-oriented language Smalltalk will be used. In doing so, we will evaluate how the basic principles of the object-oriented language might lead us to the creation of improved computer codes for the analysis of mechanical systems in terms of the software qualities such as reliability, flexibility and ease of maintenance. Specifically, effort will be limited to two dimensional kinematic analysis of the mechanical systems

The layout of the thesis is as follows. In chapter 2, we give definitions of some of the terminology in the object-oriented programming and discuss differences between object-oriented programming and structured programming. specifically we discuss the object-oriented language Smalltalk. Chapter 3 reviews the basic principle of planar kinematics. Chapter 4 describes the program written in Smalltalk followed by some simple examples in chapter 5. Finally the summary and conclusion are discussed in chapter 6. Also discussions on Smalltalk and a brief summary of the class hierarchy are given in the appendices.

CHAPTER II

OBJECT-ORIENTED PROGRAMMING

2.1 Definitions

In essence, object-oriented programming involves sending messages to objects. An object is a package of information and descriptions of its manipulations. A message is a specification of one of an objects's manipulations and a method, which is similar to a procedure or subroutine, is the description of the actions to be taken when a message is received by an object. A protocol is a set of messages to which an object can respond. A class is a description of one or more similar objects and an instance is an object described by a particular class. A subclass is a class that is created by sharing the description of another class, often modifying some aspects of that description. An instance variable is the information used to distinguish an instance from other instances of the same class. A class variable is a variable shared by all instances of a class and the class itself. A global variable is a variable shared by

instances of all classes. An effective abstraction is a simplified description of a system which emphasizes the relevant characteristics of the system but suppresses other details. Abstraction techniques have become an important element in the management of intellectual complexity and they can greatly simplify the process of creating, verifying, maintaining and extending complex system.[28] Encapsulation is the process by which individual software components are defined. A good method of encapsulation has following desirable features[1]:

a) A clear boundary defining the scope of all its internal software
b) A well-defined interface that describes how the software component interacts with other software
c) A protected internal implementation that gives the details of the functionality provided by the software component

The contribution of encapsulation is that it restricts the effects of change by placing a wall of code around each piece of data. All access to the data is handled by procedures that were put there to mediate access to the data.[2] Inheritance is a formal ordering of classes. Inheritance of class description reduces the information needed to build up descriptions since each statement describes how a new class differs from a previous one in the class hierarchy. An advantage of inheritance is that it is possible to postpone specific details of information to lower levels in forming a class hierarchy. Polymorphism is a unique characteristic that different objects respond to the same

message with their own behavior. Dynamic binding(or late binding) means that binding or linking is done later than compile time, generally while the program is running. Dynamic binding is needed in loosely coupled collections[2] where computer code can not predict the type of data to be operated on until the code is being run. The notion of structured programming is a procedure for developing complex systems wherein a developer is free to assume the existence of any data structures or operational procedures, even if they do not yet formally exist. This approach depends heavily on effective and coherent abstraction technique for its success.[29]

2.2 Comparison and Contrast of Object-Oriented Problem Solving with Structured Problem Solving

Material in this section is a brief summary taken from [1].

To understand what object-oriented problem solving is about, we make a brief comparison between object-oriented languages and procedural languages. We will compare the two approaches in the following categories.

2.2.1 Method for accomplishing actions with data

A basic difference between the object-oriented paradigm and the structured paradigm for computer problem solving is the way in which actions on data are accomplished. In the object-oriented programming(OOP) approach, messages(actions) are sent to objects(data) and the object responds to the message in a predetermined way. In a structured approach, parameters(data) are sent to procedures(actions) and the procedures operate on the data in a predetermined way using a relatively small and fixed instruction set. If we further examine the details of the two approaches, some internal details of the receiver object in an OOP approach must be known by the sender in the structured approach. Thus for a procedure call, we have to give further explanations(e.g. Which parameters are input?, Which are output?, and What is their type?). This explanation, if it is included, is usually in the form of ad hoc comments attached to the parameters.

2.2.2 Abstraction

In an object-oriented language, selected classes of objects, can be represented as data abstractions and messages can be represented as functional abstractions. Although not all classes

can be considered to be data abstractions, they may be abstractions for certain physical objects, ideas, processes, or concepts. This is a more general capability than provided by data abstraction alone. The key issue is on classes of objects and how they can be used to represent the other abstraction.

In a procedural language, combining preset data types that the specific language provides represent data abstractions, and functional abstractions are represented as procedures operating on the data abstractions. Again the key issue is on data types and how they can be used to represent various data abstractions as well as more general abstractions.

Besides differences in implementation details, the major difference for representing abstractions is that data types are the central focus in procedural languages while classes are the central focus for object-oriented languages.

2.2.3 Encapsulation[1]

In object-oriented problem solving, the unit of encapsulation is the object. It consists of the complete protocol as given in its class description and the private data of the particular instance of that class. A class description is for one kind of object only. Distinct objects that are instances of the same class are separate units of encapsulation.

In a procedural language, encapsulation is usually in the form of library elements. Such elements may contain more than one data abstraction, as well as the associated functional abstractions. Depending upon the particular language, the interface definition may or may not be separate from the implementation. Further, in some procedural languages the internal implementation details are not protected.

2.2.4 Inheritance and polymorphism

In general procedural languages do not support inheritance and polymorphism as all object-oriented languages do. Without support of inheritance, library elements and the data abstractions are of equal hierarchical level which places a severe restriction in understanding the relationship among various elements of a problem solution in non-object oriented languages. Also lack of support for polymorphism causes a number of complicating factors in the choice of names for procedures as similar operations in different library elements must be distinct.

As an advantage of dynamic binding, polymorphism allows code to be written that is insensitive to the types of object receiving the message. Of course, if the object does not happen to have a method for the message sent, an error will occur at run

time.[26]

Due to inheritance and polymorphism, we can find a more natural solution to problems and also the ability to show dependency relationships through subclasses and the reduction of redundancy can be beneficial. As an added advantage, polymorphism enhances the readability of software by allowing the same message, indicating a particular action, to be sent to different kinds of objects.[1]

2.2.5 Extensibility and relative status of new protocol

Extensibility is a property of computer languages that allows the user to define new constructs. Most modern languages are extensible; however, there are significant differences in the methodology supported by individual languages for adding new constructs. In most languages, the new constructs have status that is secondary to those constructs provided by the language. This typically means that the new constructs suffer significant degradation in efficiency.

In a truly object-oriented language all objects have equal status. This includes objects that are user-generated and objects that are part of the system kernel. This consistency of status is achieved at the expense of overall efficiency for existing OOP languages. Thus the tradeoff is consistency versus

local optimization.[1]

2.2.6 Refinement of class hierarchy vs. equivalent status of all software components

With the support of inheritance and polymorphism, it is possible to develop a hierarchy of classes in an object-oriented approach to problem solving. Advantage of a class hierarchy is that we can add incremental capability to currently existing class hierarchy through subclasses as new problem solution requires.

In a structured approach to problem solving, as all new library elements have equal status, they cannot draw on existing capability without redundant storage of copies of selected procedures and further, the interdependence of various library elements is not clear without a hierarchical structure. In procedural languages, variations on modular design charts have been used to show relative interdependence of the library elements. A solution in a procedural language must define separate and complete library elements for each of the "subclasses", and it can cause much of the functional abstraction to be repeated.

2.2.7 Passing objects as parameters in an OOP approach and passing records as parameters in a structured approach

The major difference in passing structured parameters in or out is that a record parameter, in a procedural language, can be accessed in any way by the receiving procedure while an object parameter sent to another object can be accessed by that object only. Specially in the object-oriented approach, accessing is provided by the protocol of the object parameter, thus we can eliminate chance for errors from misuse due to object-oriented encapsulation.

CHAPTER III

KINEMATICS

The kinematics analysis is used to determine the displacement, velocity and acceleration of mechanical parts as a result of the generated motion. In specific, kinematic analysis is a study of motion of the system, regardless of the forces that produce the motion.

For large problems with many variables and many equations, it is difficult and often tedious to write and solve these nonlinear algebraic equations by hand, thus numerical methods and computer programs are the usual way to solve these problems.

This chapter presents some of the definitions used in kinematics, general forms of kinematic equations, and numerical methods for solving such equations. Following material is taken from [9]. An interested reader can find more comprehensive explanation from the sources[17,18].

3.1 Definitions of the Kinematic Elements

The definition of a rigid body is a system of particles in which distances between particles remain unchanged. A mechanism is a collection of rigid elements which produce a specified motion. The link is a individual rigid body which makes up a mechanism. A kinematic pair or joint is combination of two links in contact. The definition of coordinates is any set of parameters that uniquely specifies the configuration of all bodies of a mechanism. In this thesis, the cartesian coordinates which normally require that the position of each body in space be defined relative to a fixed global coordinate system are used exclusively. The minimum number of coordinates required to completely describe the system configuration is called the number of degrees of freedom of the system. A kinematic pair imposes certain conditions on the relative motion between the two bodies it comprises. When expressed in analytical form, they are called equations of constraint which reduces the number of degrees of freedom in a system. In a kinematic pair, since the motion of one body fully or partially determines the motion of the other, it is obvious that the number of degrees of freedom of a kinematic pair is less than the total number of degrees of freedom of two rigid bodies.

3.2 Cartesian Coordinates[9]

The coordinates that specify the location of each body need to be defined to specify the configuration of a planar mechanical system. Let the xy coordinates system shown in Figure 3.1 be a global reference frame. Define a body-fixed $\xi_i\eta_i$ coordinate system embedded in body i. Body i can be located in the plane by specifying the global coordinates $\mathbf{r}_i = [x,y]^T_i$ of the origin of the body-fixed coordinate system and the angle ϕ_i of rotation of this system relative to the global coordinate system. The usual convention is that the angle is positive if the rotation from positive x axis to positive ξ_i axis is counterclockwise.

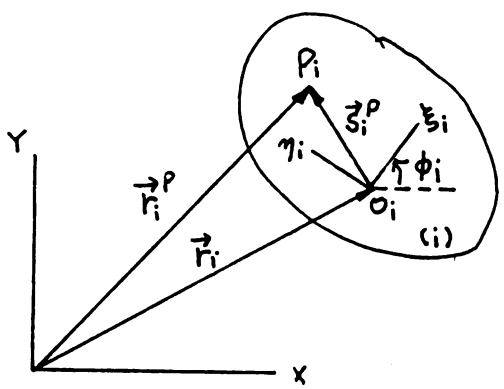


Figure 3.1 Locating point *P* relative to the body-fixed frame and global coordinate systems

A point P_i on body i can be located from the origin of $\xi_i\eta_i$

axes by the vector $\vec{s}^P_i$. The coordinates of point P_i with respect to the $\xi_i\eta_i$ coordinate system are ξ^P_i and η^P_i. The body-fixed components of vector $\vec{s}^P_i$ are shown as $s'^P_i = [\ \xi^P, \eta^P\ ]^T_i$. Since P_i is a fixed point on body i, ξ^P_i and η^P_i are constants, therefore s'^P_i is a constant vector. The global xy components of vector $\vec{s}^P_i$ vary when body i rotates. Point P_i may also be located by its global coordinates $r^P_i = [\ x^P, y^P\ ]^T_i$.

The relation between the local and global coordinates of point P_i is

$$r^P_i = r_i + A_i s'^P_i \qquad \text{(Eq. 3.1)}$$

where

$$A_i = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}_i$$

is the rotational transformation matrix for body i.

Equation 3.1 in expanded form can be written as

$$x_i = x_i + \xi^P_i \cos\phi_i - \eta^P_i \sin\phi_i$$
$$y_i = y_i + \xi^P_i \sin\phi_i + \eta^P_i \cos\phi_i$$

3.3 Kinematic Constraints[7]

In most kinematic systems, it is necessary to impose constraints on relative position and orientation between bodies. The objective for each joint is to define a set of algebraic constraint equations that approximate a physical joint. Since

the physical joint is to be represented by the constraint equations, it is important that:

(a) the equations employed imply the relative positional restrictions imposed by the physical joint,

(b) the number of constraint equations derived be equal to the number of degree-of-freedom restricted by the joint,

(c) the equations derived be independent.

3.3.1 Revolute joints[9]

Schematic representation of a revolute joint connecting to bodies i and j is shown in Figure 3.2. The center of the joint is denoted by the point P that can be considered to be two coincident points. The constraint equations for revolute joint are obtained from the vector loop equation.

$$r_i + s^P_i - r_j - s^P_j = 0$$

which is equivalent to

$$\Phi = r_i + A_i s'^P_i - r_j - A_j s'^P_j = 0$$

more explicitly,

$$\Phi = \begin{matrix} x^P_i - x^P_j = 0 \\ y^P_i - y^P_j \quad 0 \end{matrix}$$

The two constraints of above equation reduces the number of degrees of freedom of the system by 2.

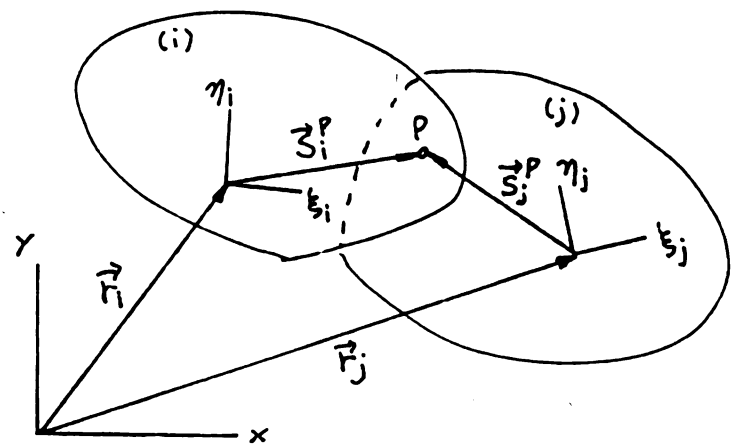


Figure 3.2 Revolute joint P connecting bodies i and j

3.3.2 Translational joints[9]

In a translational joint, the two bodies translate with respect to each other parallel to an axis known as the line of translation; therefore, there is no relative rotation between the bodies. For translational joint, there are infinite number of parallel lines of translation. A constraint equation for eliminating the relative rotation between two bodies i and j is written as

$$\phi_i - \phi_j - (\phi^0_i - \phi^0_j) = 0 \qquad \text{(Eq. 3.2)}$$

where ϕ^0_i and ϕ^0_j are the initial rotational angles.

In order to eliminate the relative motion between the two bodies in a direction perpendicular to the line of translation, the two vectors $\vec{s}_i$ and $\vec{d}$ shown in Figure 3.3 must remain parallel. These vectors are defined by locating three points on the line of translation - two points on body i and one point on body j. This condition is enforced by letting the vector product of these two vectors be zero. A simple method would be to define another vector $\vec{n}_i$ perpendicular to the line of translation and to require that $\vec{d}$ remain perpendicular to $\vec{n}_i$; i.e., that

$$n^T_i d = 0 \qquad \text{(Eq. 3.3)}$$

where

$$n_i = \begin{bmatrix} x^P_i - x^R_i \\ y^P_i - y^R_i \end{bmatrix}$$

$$d = \begin{bmatrix} x^P_j - x^P_i \\ y^P_j - y^P_i \end{bmatrix}$$

if $n_i = s_i$, then

$$n_i = \begin{bmatrix} x^P_i - x^R_i \\ y^P_i - y^R_i \end{bmatrix} = \begin{bmatrix} -(y^P_i - y^Q_i) \\ x^P_i - x^Q_i \end{bmatrix}$$

Thus, equations 3.2 and 3.3 yield the two constraint equations for a translation joint as

$$\Phi = \begin{bmatrix} (x^P_i - x^Q_i)(y^P_j - y^P_i) - (y^P_i - y^Q_i)(x^P_j - x^P_i) \\ \phi_i - \phi_j - (\phi^0_i - \phi^0_j) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Note that a translational joint reduces the number of degrees of

freedom of a system by 2.

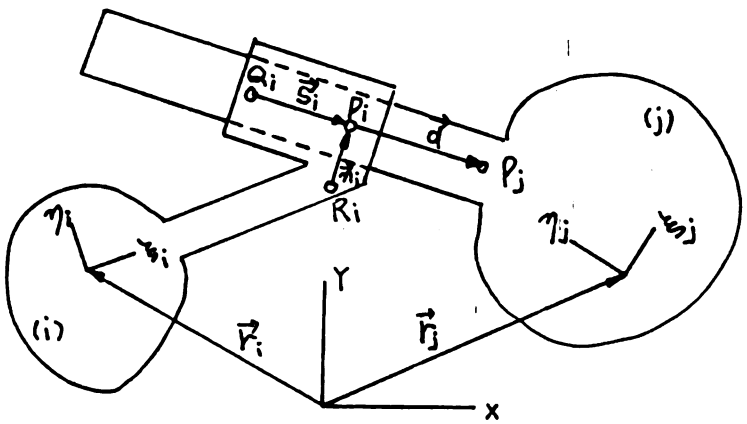


Figure 3.3 A translational joint between bodies i and j

3.3.3 Simplified constraints[9]

Generally the constraint equations describing certain kinematic conditions between two bodies, if one of the bodies is a nonmoving body, can be simplified or replaced by other simple equations. In order to constraint translation of the origin or angular motion of a rigid body, one or more of the following equations may be used:

$$\Phi = x_i - c_1 = 0$$

$$\Phi = y_i - c_2 = 0$$

$$\Phi = \phi_i - c_3 = 0$$

where c_1, c_2, and c_3 are constant quantities. Figure 3.4 illustrates graphically the three above constraint equations.

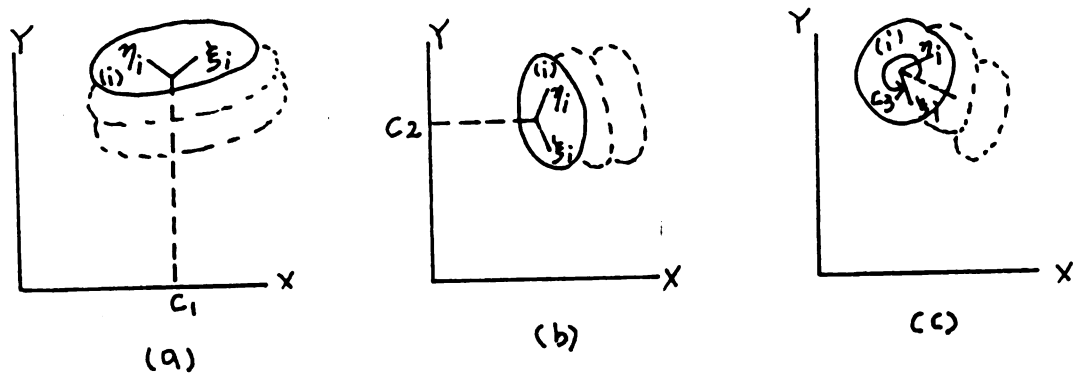


Figure 3.4 The body can move with (a) constant x_i (b) constant y_i, and (c) constant ϕ_i

3.3.4 Driving links[9]

In kinematically driven systems, the motion of one or more bodies is usually defined. For example, in the slider-crank mechanism of Figure 3.5, the driving link i rotates with known constant angular velocity ω. If kinematic analysis is to be

performed using the appended driving constraints method, then the motion of the driving link must be specified in the form of a driving constraint equations. For the mechanism of Figure 3.5, one moving constraint of the form can be employed,

$$\Phi = \phi_i - d(t) = 0$$

where $d(t) = \phi^0_i + \omega t$ and ϕ^0_i is the angle ϕ_i at $t = 0$. If the driving link rotates with a constant angular accelerations α, then the above equation can be used with $d(t) = 0.5\alpha t^2 + \dot{\phi}^0 t + \phi^0$, where $\dot{\phi}^0$ is the angular velocity at $t = 0$.

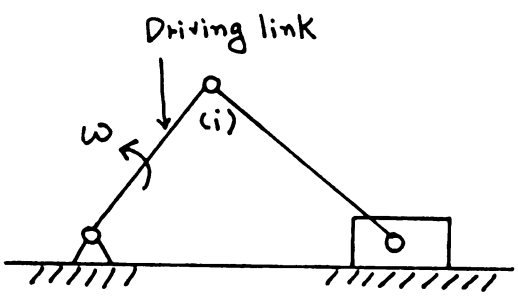


Figure 3.5 A slider-crank mechanism

3.4 Kinematic Analysis

For a mechanical system, kinematic analysis is a study of motion of the system, regardless of the forces that produce the motion. When the time history of position of one or more bodies of the system is prescribed, we need to determine the time

history of position, velocity, and acceleration of the remaining bodies by solving system of nonlinear algebraic equations for position and linear algebraic equations for velocity and acceleration. The only important equations to consider in the kinematic analysis are constraint equations. The first and second time derivatives of the constraint equations provide the kinematic velocity and acceleration equations.

At any given instant in position analysis, we must know the value of k coordinates which is the same as the number of degrees of freedom. Thus the constraint equation can be solved for the other $m = n - k$ coordinates. The same principle applies for velocity and acceleration analysis; the value of k velocities and k accelerations must be known in order to solve kinematic velocity and acceleration equations for the other unknown velocities and accelerations. There are several ways in doing kinematic analysis but in this report, we will be using so called the method of appended driving constraints.[9] In this method, additional constraint equations, called driving constraints, equal in number to the number of degrees of freedom of the system, are appended to the original kinematic constraints. The driving constraints are equations representing each independent coordinate as a function of time.

This method stated in its most general form, if there are m kinematic constraints, the k driving constraints must be appended

to the kinematic constraints to obtain n = m + k equations:

$$\begin{bmatrix} \Phi = \Phi(q) - 0 \\ \Phi^{(d)} = \Phi(q,t) - 0 \end{bmatrix} \qquad \text{(Eq. 3.4)}$$

where superscript (d) denotes the driving constraints. Equation 3.4 represents n equations in n unknowns q which can be solved at any specified time t.

The velocity equations are obtained by taking the time derivative of Equation 3.4:

$$\begin{bmatrix} \Phi_q \dot{q} = 0 \\ \Phi_q{}^{(d)} \dot{q} + \Phi_t{}^{(d)} = 0 \end{bmatrix} \qquad \text{(Eq. 3.5)}$$

which represents n algebraic equations, linear in terms of $\dot{q}$.

Similarly, the time derivative of Equation 3.5 yields the acceleration equations:

$$\begin{bmatrix} \Phi_q \ddot{q} + (\Phi_q \dot{q})_q \dot{q} = 0 \\ \Phi_q{}^{(d)} \ddot{q} + (\Phi_q{}^{(d)} \dot{q})_q \dot{q} + 2\Phi_{qt}{}^{(d)} \dot{q} + \Phi_{tt}{}^{(d)} = 0 \end{bmatrix} \qquad \text{(Eq. 3.6)}$$

which represents n algebraic equations linear in terms of $\ddot{q}$. The term $-(\Phi_q \dot{q})_q \dot{q}$ in Equation 3.6 is referred to as the right side of the kinematic acceleration equations.

In position Analysis, the Newton-Raphson algorithm uses values of the coordinates from the previous time step as an estimate on q^i to start the iteration. Thus the Newton's method can be used to improve these estimates by starting the iterative computation at a good estimate for the position of the system. Presuming that position, velocity, and acceleration are known at

time t^i, one may approximate the generalized coordinate vector at time t^{i+1} using the Taylor second order expansion.

$$q^{i+1} = q^i + (t^{i+1} - t^i)\dot{q}^i + 0.5(t^{i+1} - t^i)^2\ddot{q}^i$$

This initial estimate can be used to begin Newton-Raphson iteration and, if the difference between time points is not extreme, rapid convergence may be expected.[7]

The general procedure for kinematic analysis using this method is summarized in the following algorithm:

Algorithm

(a) Set a time step counter i to i = 0 and initialize $t^i = t^0$.

(b) Append k driving equations to the constraint equations.

(c) Solve Eq. 3.4 iteratively to obtain q^i.

(d) Solve Eq. 3.5 to obtain $\dot{q}^i$.

(e) Solve Eq. 3.6 to obtain $\ddot{q}^i$.

(f) If final time is reached, then terminate; otherwise increment t^i to t^{i+1}, let i → i + 1, and go to (c).

3.5 Newton-Raphson Method for Nonlinear Algebraic Equations[9]

One of the most frequently occurring problems in scientific work is to find the roots of one or a set of nonlinear algebraic equations of the form $\Phi(x) = 0$ i.e., zeros of the functions

$\Phi(x)$. In general, iterative methods are employed and the most common and frequently used method is known as the Newton-Raphson method.

Consider n nonlinear algebraic equations in n unknowns,

$$\Phi(x) = 0$$

where a solution vector x is to be found. The Newton-Raphson algorithm for n equations is stated as

$$x^{j+1} = x^j - \Phi_x^{-1}(x^j)\, \Phi(x^j) \qquad \text{(Eq. 3.7)}$$

where $\Phi_x^{-1}(x^j)$ is the inverse of the Jacobian matrix evaluated at $x = x^j$.

The term $\Phi(x^j)$ on the right side of Equation 3.7 is known as the vector of residuals, which corresponds to the violation in the equations.

Equation 3.7 may be restated as a two-step operation:

$$\Phi_x(x^j)\, \Delta x^j = -\Phi(x^j) \qquad \text{(Eq. 3.8)}$$

$$x^{j+1} = x^j + \Delta x^j \qquad \text{(Eq. 3.9)}$$

where Equation 3.8, which is a set of n linear equations, is solved for Δx^j. Then, x^{j+1} is evaluated from Equation 3.9. Gaussian elimination or LU factorization methods are frequently employed to solve Equation 3.8. The term $\Delta x^j = x^{j+1} - x^j$, known as the Newton difference, shows the amount of correction to the approximated solution in the jth iteration. The Newton-Raphson method, when it works, is very efficient. Because the Newton-Raphson method will not always converge, it is essential to

terminate the process after a finite number of iterations. The computational procedure is stated as follows:

Algorithm

(a) Set the iteration counter $j = 0$.

(b) An initial estimate x^0 is made for the desired solution.

(c) The functions $\Phi(x^j)$ are evaluated. If the magnitudes of all of the residuals $\Phi_i(x^j)$, $i = 1,\ldots,n$, are less than a specified tolerance ε, i.e., if $|\Phi_i| < \varepsilon$, $i = 1,\ldots,n$, then x^j is the desired solution; therefore terminate. Otherwise, go to (d)

(d) Evaluate the Jacobian matrix $\Phi_x(x^j)$ and solve Equation 3.8 and 3.9 for x^{j+1}

(e) Increment j; i.e., set j to j + 1. If j is greater than a specified allowed number of iterations, then stop. Otherwise go to (c).

CHAPTER IV

PROGRAM

This chapter presents in detail how the object-oriented programming methodology can be applied to kinematic analysis of the mechanical system composed of several inter-connected rigid or flexible bodies, it is often useful to divide the problem that must be solved into smaller pieces and to solve those pieces separately, to the extent that is possible. Then the separate pieces must be combined to form a single consistent solution to the original problem. This is the very foundation of object-oriented problem solving because the object-oriented programming principle is to develop a class structure which organizes the elements of a system so that specific details are postponed to lower level of classes. This kind of structuring encourages a hierarchical decomposition of description and computational process.

Many of the practical benefits of object-oriented programming follow from this characteristic. However, achieving this kind of class structuring is by far the most difficult aspect of object-

oriented programming. We see the design process as at least a three level process wherein the class being invented is intellectually coupled to both its superclass and subclass. For example, while inventing a class, we consider potential superclass. In order to raise the level of abstraction over the present perspective. Concurrently, we reflect on the nature of potential subclasses in order to discover how effectively the present class encourages the development of objects at the subclass level. Thus our basic design goal is: To the extent that is possible the upper levels of the class hierarchy should involve abstract description, whereas the lower levels should represent more concrete or specific constructs. This will enable hierarchical descriptions and computational process. It will facilitate the many practical benefits associated with object-oriented programming. For example, this will make it easy to add or change computational schemes.

The first step in an object-oriented solution to a problem is to define the objects. Once the potential objects are identified, the next step is to develop a complete description of each object that is part of the solution. This description includes the characteristics of the objects and that actions to which each object must respond. The actions become method details with appropriate message selectors to indicate the expected response. After this is completed, we may decide how to

add the subclasses in order to represent more specific details.

We now describe a class hierarchy structure which illustrates how object-oriented programming can be applied to kinematics. The general approach is based on following underlined classes which will be described below.

```
Object
    Collection
        IndexedCollection
            FixedSizeCollection
                Array
                Matrix
                    Vector
    MechanicalObject
        Constraint
            RevoluteJoint
            TranslationalJoint
            SimpleConstraint
            DrivingLink
        Element
            RigidBody
            FourBar
            FourSlider
            SliderCrank
            QuickReturn
        Node
```

In order to represent the world of kinematics, we choose an abstract class which contains every kinematic element such as constraints, rigid bodies, and complete mechanisms. The MechanicalObject class plays this role and is a superclass of all of the classes that represent kinematic elements. It also defines two methods such as naming and indexing elements, and these methods with private variables are inherited by all of its subclasses. By making the MechanicalObject be an abstract

superclass, it is possible to postpone details of the mechanical elements to lower levels.

The Constraint class is a superclass of all of the classes which represent mechanical joints and drivers. This class defines common methods for initializing instances of its subclasses. Again details of each joints and drivers are postponed to next levels. For example, the RevoluteJoint class is a subclass of Constraint class. It inherits two instance variables(name and index) from the MechanicalObject class and instance method(initialize) from the Constraint class. It differs from its super class Constraint class in that it represents specific a joint class among several constraints by redefining its class description protocol. As for the encapsulation of RevoluteJoint class, an instance of the RevoluteJoint class can represent a unique revolute joint in mechanism by specifying its instance variables such as two rigid bodies and coordinates of the point in each rigid body and by defining detailed methods involving these internal objects. An advantage of the encapsulation is to limit the effects of change by placing a wall of code around internal data structure. Since access to this internal data structure can only be made through its instance methods, reliability is improved. Similarly, these principles have been applied in defining for translational joints, driving links and simple constraints. A brief summary of

class protocol for these subclasses is given in Appendix B.

The Element class is an object representing any mechanisms which consist of the mechanical constraints and the rigid bodies. Instance variable elementList contains information on the connection of each rigid body in the system and instance variable constraintList provides information about the joints connectivity between rigid bodies. By storing these data in instance variables, an instance of the Element class is created and then we can perform complete kinematic analysis on this object by sending messages.

We note that several of the concepts of OOP play a beneficial role here. For example, the Jacobian matrix is needed for the kinematic analysis of a mechanism and the method for doing this uses the concept of polymorphism. If we send the jacobian message to an instance of the Element class, then each joint in that mechanism gets this same message and responds accordingly. The same principle has been applied in computing the cartesian position vector, velocity vector, acceleration vector etc. The contribution of polymorphism coupled with inheritance is that, if we want to add more classes or subclasses, there will be less original code for a programmer to write.

A single rigid body is a unit element of the mechanism, accordingly this class is classified as subclass of the Element class along with other mechanisms such as four bar mechanisms,

slider crank mechanisms and quick return mechanisms. Of course, as we develop more specific mechanisms, we can add those to existing class hierarchy since the organization of this class hierarchy is very much flexible in terms of adding or modifying class objects. One example which illustrates these points is given in Sec. 5.3.

The Node class represents the cartesian coordinates of the points. The Vector class defines basic concepts of vector algebra and its manipulations. Similarly, so does the Matrix class. The reason behind placing Vector and Matrix classes under the Collection class is to utilize another subclass Array and some of methods in Collection class protocol.

Based on discussions so far, we can notice how the world of kinematics has been broken down from abstract(MechanicalObject) to specific subclasses(RevoluteJoint, TranslationalJoint, SimpleConstraint, DrivingLink, RigidBody, FourBar) for the design of the class hierarchy in terms of the objects. In doing this, the object-oriented principles such as abstraction and encapsulation have been applied. Also the FourSlider class which has been created by combining instances of existing subclasses does not lose any efficiency because this subclass, once added to class structure, becomes part of the system class hierarchy. Particularly, for the kinematic analysis of the Element class object, we have shown how polymorphism is used in forming system

Jacobian matrix and from this, it is evident that this approach reduces overall code to be written. According to Cox's definition of programmer productivity that bulk is bad, we can improve programmer's productivity since we have less original code to write. A summary of each class description is presented in the appendix.

CHAPTER V

SIMPLE EXAMPLES

5.1 Modeling and Analysis

Generally there are many ways to model a particular mechanism. The important factor to consider in kinematic analysis is that there must be no free degrees of freedom for the combination of bodies, kinematic constraint, and drivers in kinematic modeling.

Here, in order to assemble the mechanism, an initial estimate of the position and orientation of each body as well as joints data of the mechanism must be provided. These estimates, x, y, and ϕ for each body, can be obtained from a reasonably scaled diagram of the mechanism. These estimates need not to be extremely accurate. The Newton-Raphson algorithm starts the iterations using the estimated values and finds exact values for the coordinates at $t = 0$.

After specification of the bodies and kinematic constraints, one or more degrees of freedom will remain. To complete the

model a number of drivers equal to the number of degrees of freedom must be specified. Drivers are usually define relative or absolute motion that is imposed by motors or by specifying some characteristics of motion that is desired, regardless of the prime mover that is to generate the motion.

We will now illustrate these ideas with three examples.

5.1.1 Kinematic analysis of a four bar mechanism

A four bar mechanism with four revolute joints is modeled in Figure 5.1. In Model 1, each link and ground is modeled as a body. Four revolute joints complete the model, and the ground constraint is treated as having three simple constraints on its x, y, and ϕ motion, as follows:

Model 1

Bodies

4 bodies	
(3 generalized coordinates / body)	12 g.c.

Constraints

Revolute Joint 1	2
2	2
3	2
4	2
Ground Constraint	3
(Body 1 is ground)	
Driver 1	1
Total No. of Constraints	12

DOF = 12 - 12 = 0

From the Figure 5.1, initial position estimates can be measured

and tabulated in Table 5.1.1.

Table 5.1.1 Initial position estimate of four bar mechanism

Body No.	1	2	3	4
x	0.0	0.5	2.6	3.5
y	0.0	0.8	2.6	1.8
Φ(rad)	0.0	1.047	0.524	1.047

Revolute joint data for Model 1 are shown in Table 5.1.2.

Table 5.1.2 Revolute joint data of four bar mechanism

Joint No.	1	2	3	4
Common Point	A	B	C	D
Body i	1	2	3	4
ξ^P_i	0.0	1.0	2.0	2.5
η^P_i	0.0	0.0	0.0	0.0
Body j	2	3	4	1
ξ^P_i	-1.0	-2.0	2.0	0.5
η^P_i	0.0	0.0	0.0	1.5

Sample program for model 1 of a four bar mechanism is as follows:

Class method for instantiating a Four Bar Mechanism

```
model1

|fourbar ground body2 body3 body4 rjoint1 rjoint2 rjoint3
rjoint4 simple1 simple2 simple3 driver1
node1 node2 node3 node4 node5 node6 node7 node8|

fourbar := Element new initialize:#('fourbar').
ground := RigidBody new initialize:#('ground' 1  0.0 0.0 0.0).
body2 := RigidBody new initialize:#('body2' 2  0.5 0.8 1.047).
body3 := RigidBody new initialize:#('body3' 3  2.6 2.6 0.524).
```

```
body4 := RigidBody new initialize:#('body4' 4  3.5 1.8 1.047).
node1 := Node new initialize:#('node1' 1  0.0 0.0).
node2 := Node new initialize:#('node2' 2  -1.0 0.0).
node3 := Node new initialize:#('node3' 3  1.0 0.0).
node4 := Node new initialize:#('node4' 4  -2.0 0.0).
node5 := Node new initialize:#('node5' 5  2.0 0.0).
node6 := Node new initialize:#('node6' 6  2.0 0.0).
node7 := Node new initialize:#('node7' 7  -2.0 0.0).
node8 := Node new initialize:#('node8' 8  2.5 0.0).
simple1 := SimpleConstraint new initialize:#('simple1' 1).
simple2 := SimpleConstraint new initialize:#('simple2' 2).
simple3 := SimpleConstraint new initialize:#('simple3' 3).
rjoint1 := RevoluteJoint new initialize:#('rjoint1' 4).
rjoint2 := RevoluteJoint new initialize:#('rjoint2' 5).
rjoint3 := RevoluteJoint new initialize:#('rjoint3' 6).
rjoint4 := RevoluteJoint new initialize:#('rjoint4' 7).
driver1 := DrivingLink new initialize:#('driver1' 8).
simple1 isOn:ground direction:'x' with:0.
simple2 isOn:ground direction:'y' with:0.
simple3 isOn:ground direction:'angle' with:0.
driver1 isOn:body2 direction:'angle' with:1.0472 with:6.2832
        with:0.0.
rjoint1 connect:ground with:node1 to:body2 with:node2.
rjoint2 connect:body2 with:node3 to:body3 with:node4.
rjoint3 connect:body3 with:node5 to:body4 with:node6.
rjoint4 connect:body4 with:node7 to:ground with:node8.
fourbar
    addElement:ground;
    addElement:body2;
    addElement:body3;
    addElement:body4;
    addConstraint:simple1;
    addConstraint:simple2;
    addConstraint:simple3;
    addConstraint:rjoint1;
    addConstraint:rjoint2;
    addConstraint:rjoint3;
    addConstraint:rjoint4;
    addConstraint:driver1.
^fourbar
```

This completes an instance creation of the FourBar class object which describes a unique four bar mechanism. This four bar object completely encapsulates its internal data structure.

Suppose we want to add some points of interest on one or more bodies, all we need to do is to access a specific component and then add an interesting point to that component and this is shown below. By dealing directly with a component(body 3) of the fourbar object, rest of internal data structure of the fourbar mechanism has not been changed and the rigid body 3 is also an encapsulation of the RigidBody class, thus adding a point to it does not change its internal data structure. In the case of Fortran programs, we often need to modify data file, which can be cumbersome and also can cause a problem if we change the wrong data. In an object-oriented program, these problems are minimized by encapsulation, thus the simulation process becomes safer and easier. To run the sample program, we execute following statements.

```
|fourbar|
fourbar := FourBar model1.
(fourbar getElement:'body3') addInterestingNode:(Node new
   initialize:#('node9' 9  0.5 1.5)).
fourbar kinematicAnalysisFrom:0.0 to:1.0 with:0.025.
```

A portion of the output for the first two time steps is as follows.

KINEMATIC ANALYSIS for Four Bar Mechanism

```
TIME = 0.0
element No.    x              y              angle
1              0.0            0.0            0.0
2              4.9999788e-1   8.6602663e-1   1.0472
3              2.8235171      2.5534971      4.2324569e-1
4              3.5735192      1.6874704      1.0042045

element No.    vel X          vel Y          angular Vel
1              0.0            0.0            0.0
```

```
2               -5.4414185      3.1415867       6.2832
3               -11.084945      6.7318329       2.4604019e-1
4               -5.6435267      3.5902462       3.3443707

element No.     acc X           acc Y           angular Acc
1               0.0             0.0             0.0
2               -19.739217      -34.189521      4.1023146e-15
3               -52.441015      -39.898215      15.645856
4               -32.701798      -5.7086945      12.263731

INTERESTING POINT in Element 3
'node9'
pos X = 2.6633146       pos Y = 4.126499
vel X = -11.471967      vel Y = 6.6924166
acc X = -77.04228       acc Y = -42.499944

TIME = 0.025
element No.     x               y               angle
1               0.0             0.0             0.0
2               3.5836532e-1    9.3358144e-1    1.20428
3               2.5314838       2.7078006       4.3379665e-1
4               3.4231185       1.7742192       1.0910443

element No.     vel X           vel Y           angular Vel
1               0.0             0.0             0.0
2               -5.8658789      2.251681        6.2832
3               -12.220203      5.5578081       5.8104113e-1
4               -6.3543239      3.3061271       3.581477

element No.     acc X           acc Y           angular Acc
1               0.0             0.0             0.0
2               -14.147762      -36.85649       3.8054674e-15
3               -38.613196      -53.045822      11.544803
4               -24.465434      -16.189332      7.1155912

INTERESTING POINT in Element 3
'node9'
pos X = 2.3546939       pos Y = 4.2790247
vel X = -13.133149      vel Y = 5.4550858
acc X = -56.692983      acc Y = -55.617287
```

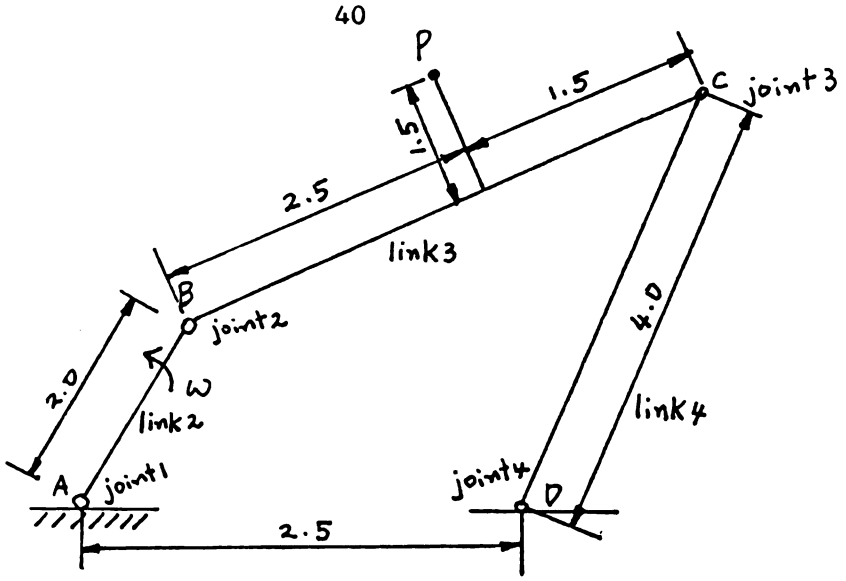


Figure 5.1 Kinematic modeling of a four bar mechanism

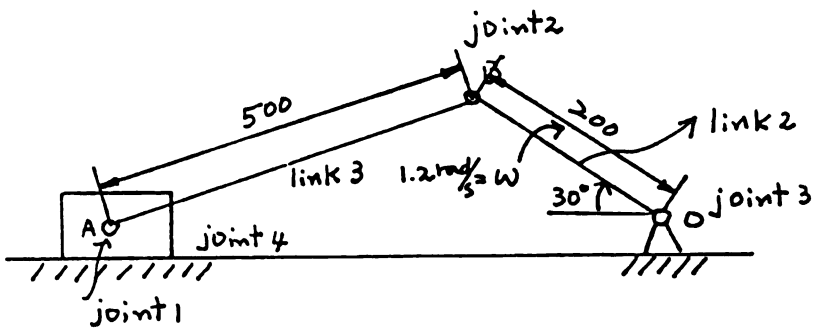


Figure 5.2 Kinematic modeling of a slider crank mechanism

5.1.2 Kinematic analysis of a slider crank mechanism

In Model 1, each link in the mechanism and ground is modeled as a body and this is shown in Figure 5.2. Joint can then be modeled as revolute and translational joints, as follows:

Model 1

Bodies

4 bodies
(3 generalized coordinates / body) 12 g.c.

Constraints

Revolute Joint 1	2
2	2
3	2
Translational Joint 1	2
Ground Constraint (Body 1 is ground)	3
Driver 1	1
Total No. of Constraints	12

DOF = 12 - 12 = 0

For Model 1, initial estimates for position and orientation are tabulated in Table 5.2.1.

Table 5.2.1 Initial position estimate of slider crank mechanism

Body No.	1	2	3	4
x	0.0	-86.6	-467.0	-663.1
y	0.0	50.0	40.0	0.0
Φ(rad)	0.0	5.76	0.2	0.0

The three revolute joints in Model 1 are defined in Table 5.2.2.

Table 5.2.2 Revolute joint data of slider crank mechanism

Joint No.	1	2	3
Common Point	A	B	O
Body i	4	3	2
ξ^P_i	0.0	300.0	100.0
η^P_i	0.0	0.0	0.0
Body j	3	2	1
ξ^P_i	-200.0	-100.0	0.0
η^P_i	0.0	0.0	0.0

A translational joint in Model 1 is defined in Table 5.2.3.

Table 5.2.3 Translational joint data of slider crank mechanism

Joint No.	1
Body i	4
ξ^P_i	0.0
η^P_i	0.0
ξ^q_i	100.0
η^q_i	0.0
Body j	1
ξ^P_i	0.0
η^P_i	0.0

To simulate, we execute following statement.

```
(SliderCrank model1) kinematicAnalysisFrom:0.0 to:1.0 with:0.1.
```

A portion of the output for the first two time steps is as follows:

KINEMATIC ANALYSIS for Slider Crank Mechanism

```
TIME = 0.0
element No.    x                y                angle
```

1	0.0	0.0	0.0
2	-86.623206	49.964188	5.76
3	-467.19395	39.971351	2.0121172e-1
4	-663.15898	0.0	0.0

element No.	vel X	vel Y	angular Vel
1	0.0	0.0	0.0
2	59.957026	103.94785	-1.2
3	145.35697	83.158278	4.2435265e-1
4	162.31892	0.0	0.0

element No.	acc X	acc Y	angular Acc
1	0.0	0.0	0.0
2	124.73742	-71.948431	0.0
3	286.99918	-57.558745	-2.5698921e-1
4	312.01541	0.0	0.0

TIME = 0.1

element No.	x	y	angle
1	0.0	0.0	0.0
2	-80.018944	59.974733	5.64
3	-451.27725	47.979786	2.4226174e-1
4	-645.43682	0.0	0.0

element No.	vel X	vel Y	angular Vel
1	0.0	0.0	0.0
2	71.969679	96.022733	-1.2
3	172.41377	76.818186	0.3956446
4	191.39672	0.0	0.0

element No.	acc X	acc Y	angular Acc
1	0.0	0.0	0.0
2	115.22728	-86.363615	0.0
3	253.21743	-69.090892	-3.1716383e-1
4	268.39268	0.0	0.0

5.1.3 Kinematic analysis of a combined Four bar and Slider crank mechanism

In this section, we develop a mechanism(FourSlider) which consists of a four bar mechanism connected to a slider crank mechanism with a new link. This example specially illustrates application of incremental capability of object-oriented languages. Suppose two mechanisms have been created, now we want to combine these two mechanisms with a new link and as a result, we want to create new mechanism called FourSlider. By encapsulation, data stored inside of a fourbar object and a slider object are protected. In order to create a combined mechanism, we add additional instance methods such as accessing, changing and removing its elements or constraints to the Element class. Having done that, we can create a new mechanism which consists of a four bar mechanism and a slider crank mechanism. Explanations on the specific steps are given later in this section. Also once we have created this combined mechanism, we can add this object as another subclass of the Element class, thus expanding the class hierarchy and making it useful in the future as a subclass. This shows a real advantage of object-oriented languages. In other words, if new mechanism had to be created from the scratch, much more computer code would have been written and debugged. By utilizing already available information, it is possible and much easier to develop a

complicated software system, thus again improving the programmer's productivity. Also, as shown in the example in Sec. 5.1, an instance of the FourSlider class encapsulates a unique combined mechanism and the simulation becomes simple process(e.g. sending a message to that object).

This mechanism has nine revolute joints and one translational joint and is shown in Figure 5.3. In Model 1, each link and ground is modeled as a body. Nine revolute joints, one translational joints and one driver complete the model, as follows:

Model 1

Bodies

8 bodies	
(3 generalized coordinates / body)	24 g.c.

Constraints

Revolute Joint	1	2
	2	2
	3	2
	4	2
	5	2
	6	2
	7	2
	8	2
	9	2
Translational Joint 1		2
Ground Constraint		3
(Body 1 is ground)		
Driver 1		1
Total No. of Constraints		24

DOF = 24 - 24 = 0

For Model 1, initial estimates for position and orientation are tabulated in Table 5.3.1.

Table 5.3.1 Initial position estimate of combined mechanism

Body No.	1	2	3	4	5
x	0.0	0.5	2.6	3.5	5.34
y	0.0	0.8	2.6	1.8	0.94
Φ(rad)	0.0	1.047	0.524	1.047	1.222

Body No.	6	7	8
x	6.88	8.0	4.5
y	0.96	0.0	1.34
Φ(rad)	5.585	0.0	5.864

The nine revolute joints in Model 1 are defined in Table 5.3.2.

Table 5.3.2 Revolute joint data of combined mechanism

Joint No.	1	2	3	4	5
Common Point	A	B	C	D	E
Body i	1	2	3	4	4
ξ^P_i	0.0	1.0	2.0	-2.0	0.0
η^P_i	0.0	0.0	0.0	0.0	0.0
Body j	2	3	4	1	8
ξ^P_i	-1.0	-2.0	2.0	2.5	-1.0
η^P_i	0.0	0.0	0.0	0.0	0.0

Joint No.	6	7	8	9
Common Point	F	G	H	I
Body i	5	1	5	6
ξ^P_i	0.0	5.0	1.0	1.5
η^P_i	0.0	0.0	0.0	0.0
Body j	8	5	6	7

ξ^p_i	1.0	-1.0	-1.5	0.0
η^p_i	0.0	0.0	0.0	0.0

The translational joint in Model 1 is defined in Table 5.3.3.

Table 5.3.3 Translational joint data of combined mechanism

Joint No.	1
Body i	7
ξ^p_i	0.0
η^p_i	0.0
ξ^q_i	0.5
η^q_i	0.0
Body j	1
ξ^p_i	0.0
η^p_i	0.0

Sample program for model 1 of a combined mechanism is as follows:

Class method(model1) for a combined mechanism(FourSlider)

```
|fourSlider fourbar slider ground body4 body5 body8 rjoint5
rjoint6 node9 node10 node11 node12|

Step 1.  The following two statements are used to create a new element,
         a four bar mechanism, and a slider crank mechanism

fourSlider := Element new initialize:#('fourSlider').
fourbar := FourBar model1.
slider := SliderCrank model2.

Step 2.  The following three statements are used to access ground element
         and link( body 4 ) in the four bar mechanism, and link
         ( body 5 ) in the slider crank mechanism

ground := fourbar getElement:'ground'.
body4 := fourbar getElement:'body4'.
body5 := slider getElement:'body5'.
```

```
Step 3.  The following statement is used to create a new link for
         connecting two mechanisms

body8 := RigidBody new initialize:#('body8' 8 4.5 1.34 5.8643).

Step 4.  The following six statements are used to create two revolute
         joints which will be placed in link( body 8) and
         coordinates of each joint

rjoint5 := RevoluteJoint new initialize:#('rjoint5',8).
rjoint6 := RevoluteJoint new initialize:#('rjoint6',9).
node9  := Node new initialize:#('node9' 9 0.0 0.0).
node10 := Node new initialize:#('node10' 10 -1.0 0.0).
node11 := Node new initialize:#('node11' 11 1.0 0.0).
node12 := Node new initialize:#('node12' 12 0.0 0.0).

Step 5.  The following two statements are used to establish joint
         connections between links( body 4 & body 8 and
         body 8 & body 5 )

rjoint5 connect:body4 with:node9 to:body8 with:node10.
rjoint6 connect:body8 with:node11 to:body5 with:node12.

Step 6.  The following is used to transfer the nodes of the ground
         element in the slider crank mechanism to ground element in
         four bar mechanism

ground getNodesFrom:(slider getElement:'ground').

Step 7.  Following statements remove the ground element, three
         simple constraints, and driver constraint from the slider
         crank mechanism
slider
    removeElement:'ground';
    removeConstraint:'simple1';
    removeConstraint:'simple2';
    removeConstraint:'simple3';
    removeConstraint:'driver1'.

Step 8.  The following combines the two mechanisms

fourSlider combine:fourbar with:slider.

Step 9.  The following adds a new link( body 8 ) and the two revolute
         joints to combined mechanism

fourSlider
    addElement:body8;
    addConstraint:rjoint5;
```

```
addConstraint:rjoint6.
```

Step 10. The following changes joint instance variables(second body) to the redefined ground element

```
(fourSlider getConstraint:'rjoint7') changeSecondElement:ground.
(fourSlider getConstraint:'tjoint1') changeSecondElement:ground.
```

To run the sample program, we execute following statement.

```
(FourSlider model2) kinematicAnalysisFrom:0.0 to:0.0 with:0.025.
```

A portion of the output for the first time step is as follows:

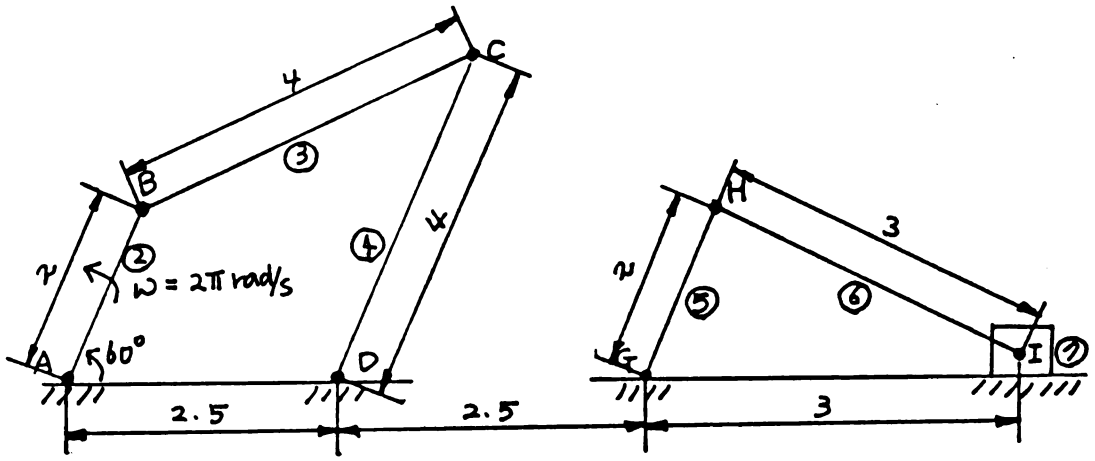
KINEMATIC ANALYSIS for Combined Four Bar & Slider Crank Mechanism

TIME = 0.0

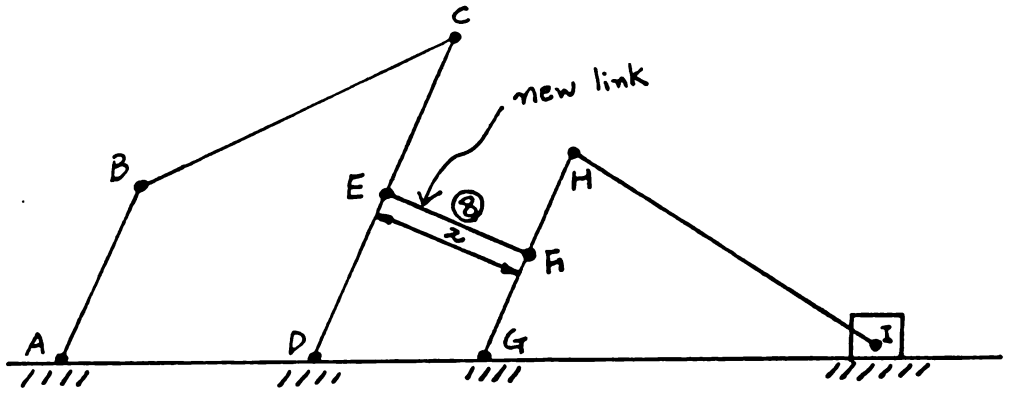
element No.	x	y	angle
1	0.0	0.0	0.0
2	4.9999788e-1	8.6602663e-1	1.0472
3	2.8235171	2.5534971	4.2324569e-1
4	3.5735192	1.6874704	1.0042045
5	5.4159581	9.0938377e-1	1.1418002
6	7.0248208	9.0938377e-1	5.6318412
7	8.2177253	0.0	0.0
8	4.4947387	1.2984271	5.8835924

element No.	vel X	vel Y	angular Vel
1	0.0	0.0	0.0
2	-5.4414185	3.1415867	6.2832
3	-11.084945	6.7318329	2.4604019e-1
4	-5.6435267	3.5902462	3.3443707
5	-6.0006053	2.7447164	6.5985401
6	-14.093583	2.7447164	-2.3008685
7	-16.185955	0.0	0.0
8	-5.822066	3.1674813	-4.5891879e-1

element No.	acc X	acc Y	angular Acc
1	0.0	0.0	0.0
2	-19.739217	-34.189521	4.1023146e-15
3	-52.441015	-39.898215	15.645856
4	-32.701798	-5.7086945	12.263731
5	-42.716733	-28.340466	27.057456
6	-73.814036	-28.340466	19.721773
7	-62.194607	0.0	0.0
8	-37.709265	-17.02458	-12.372536



before combining



after combining

Figure 5.3 Kinematic modeling of a combined four bar and slider crank mechanism

CHAPTER VI

SUMMARY AND CONCLUSIONS

The object paradigm has several features that can be used as guidelines for developing object-oriented programs. In chapter 4, the problem is broken down into the sub-problems and is characterized in terms of objects. In designing a class hierarchy structure for kinematic analysis, we have shown how object-oriented principles can be applied to engineering analysis problems. By using encapsulation and abstraction, we have broken the world of kinematics into objects which describe the basic kinematic elements. From these objects, we were able to form a mechanism and perform kinematic analysis on the mechanism. Due to the encapsulation of objects, each object is described by its internal data structure by instance variables and also a message protocol is defined for accomplishing actions with these variables. Since those instance variables can be accessed only by the methods in specific class description protocol, it is possible to insure a high degree of reliability of the program to a user. A change to one part of the software need not affect the

rest of the system. This is shown in the example in Sec. 5.1. In creating an instance of a four bar mechanism, the object-oriented approach results in much more readable program compared with structured approach, thus it is easy to understand the overall code. Specifically, if a problem has a solution that is an incremental change from existing capability, then its solution is more quickly achieved.

Complex problem solving would often be enhanced by direct access to a language's source code, particularly if modifications could be carried out in a simple and safe manner. In Smalltalk, any part of the image(source code) is readily available to the user and it is easy to browse or modify the image. Also, as an added advantage, Smalltalk's simple edit-execute cycle, which replaces edit-compile-link-execute cycle of other procedural languages, reduces time for editing and execution can be halted and resumed when bugs are encountered and fixed. Bug fixing is especially easy since hierarchical connection between a fault and the original message leading to it is always available. Thus, the Smalltalk language is coupled with powerful supporting tools which can reduce time for compiling, testing, and debugging phases of program development. This capability to integrate changes rapidly is a desirable advantage over conventional software systems based on procedural languages which may require a lot of time to rebuild systems after changes.

The example in the section 5.3 shows how we can apply Smalltalk's incremental problem solving capability. In this example, an instance of Element class combines a four bar mechanism and a slider crank mechanism and as a result, a new mechanism called FourSlider is created and subsequently, this mechanism is added to the class hierarchy as a subclass. From this example, we have demonstrated that the existing class hierarchy is flexible because the addition of a new subclass can be easily achieved without disrupting the whole software system.

The inheritance principle was used both in the design of class hierarchy structure and in the case of the subclasses of Constraint class and the Element class. Specially if we want to expand from the world of kinematics to the world of dynamics to perform dynamic analysis on mechanical systems, I believe that the advantage of the inheritance principle could be even more apparent. The polymorphism principle that the same message can elicit a different response depending on the receiver object has been utilized throughout classes. For example, we need to form a global jacobian matrix for a system in the Element class and the same message jacobian is sent to each joint which connects each rigid body. In turn, each joint computes its own Jacobian matrix and sends it back to the instance of Element class, thus forming a system Jacobian matrix. This illustrates how the object-oriented approach places the responsibility for computing

Jacobian matrix onto the joints themselves. Also polymorphism increases flexibility by permitting the addition of new classes of objects without the need to modify existing code. If we want to add more constraints as subclasses to the existing class hierarchy, all we need to do is to provide new Jacobian methods for each new constraints without changing the method in the Element class. Polymorphism coupled with inheritance reduces code to be written and as a result, we can increase the programmer's productivity because the programmer has to write less original code. This, in turn, improves maintenance of the program because there is only one place for code to perform a specific job. Polymorphism also enables dynamic or late binding. By reducing type dependence from the language, it is easier to write and modify programs written in Smalltalk.

In summing up discussions thus far, it is my belief that object-oriented programming and its environment provides several important advantages in the production and maintenance of complex software systems in terms of reliability, readability, extensibility, and flexibility. Unfortunately computational efficiency is not a strong point of Smalltalk. Ungar[5] suggested two ways to improve this poor cost-performance. One way is with clever software on a cheap, conventional machine. In addition to innovative software, special purpose hardware may further reduce the cost. For more detailed information on

Smalltalk's performance in speed and efficiency, an interested reader is referred to [5].

APPENDICES

APPENDIX A

SMALLTALK

My intention in this section is to give a brief basic understanding of the principles of the Smalltalk language. For a full description, an interested reader is referred to Smalltalk-80: The Language and Its Implementation by Goldberg and Robson(Addison-Wesley 1983). Also some material in this section are taken from [1].

Among several languages that support object-oriented problem solving, Smalltalk which was developed at the Xerox Palo Alto Research Center in the 1970s with the help of Alan Kay, is the most consistent with definitions and properties of the object-oriented paradigm.

Smalltalk is not just another language. It is an extensive programming environment and its virtual image(Smalltalk source code) consists of more than two hundred classes and several thousand methods. Although the language is small in terms of reserved words and symbols, the entire system is quite large and it takes time to learn what is in the image. Within the image,

the Smalltalk provides the capability for solving many standard computer problems because all the classes and methods are available for changing with the exception of primitives which are a group of low-level operations written in assembly language. Many new problems are solved by using or modifying existing classes and methods in the image, thus the image can grow in size as new capability are added to the system.

Since Smalltalk is an interpretive language, Smalltalk programs execute more slowly than those written in other object-oriented languages that are compiled. Compilers for Smalltalk programs that produce machine code are currently under development. Their success will help eliminate the speed disadvantage for Smalltalk production software systems. Although it is not the only widely used object-oriented language, the Smalltalk language and system continue to serve as an inspiration and model for object-oriented problem solving.[1]

Following subsections present how the underlying features of the Smalltalk language support the object-oriented paradigm.

A.1 Objects and Messages

In Smalltalk, everything is accomplished by sending messages to objects. The result of sending a message to an object is another object. By utilizing existing objects and messages in

the Smalltalk image, we can establish the desired object-message sequence as a way of problem solving.

Objects are instances of a particular class. The messages to which an object can respond are defined in the protocol for its class. Methods give the implementation details for messages and are a part of the class description protocol for a given class. These are the fundamental relationships among the five key components(object, instance, class, message, method) of the Smalltalk system. Understanding these five key components and their relationships is understanding Smalltalk.[1]

There are three kinds of messages in Smalltalk: unary, binary, keyword. A unary message is a single message selector with no arguments. A binary message is a single message selector with one argument and one or two special characters as the selector. A keyword message is a message to a single object with one or more arguments. Message selectors are typically colon-terminated identifiers.

The order of arithmetic expressions in Smalltalk is strictly from left to right unless altered by the presence of parentheses or by message priorities. The precedence of the three kinds of messages is unary, binary, and keyword. Because of this left-to-right precedence in Smalltalk which differs from most other languages, careful consideration of thinking is required to avoid unexpected error.

A.2 Abstraction of Objects and Methods

In Smalltalk, there are two types of abstraction, data and functional abstractions. The former represents private or shared data which defines the properties of the object. The latter represents the details of methods how an object is to respond to messages. There is a method for each message to which an object can respond and a message always returns a single object as its result.

Every object belongs to a specific class which has a unique name and represents a specific kind of object. To create instances of object(classes), it is required to send instance creation messages to the class name. Everything that is needed to be defined for an object, such as private data, shared data, and methods can be found in its class description protocol.

Class Object is the superclass of all classes and defines the protocol common to all object. Class Object defines the default behavior for displaying, comparing, copying, accessing indexed instance variables and error handling. Class Object includes capabilities to maintain dependency relationships between objects and to broadcast messages from an object to its dependents. Subclasses may polymorphically redefine any of the methods that are part of Class Object and they may also add new private or shared data.

All the classes in the Smalltalk are organized according to categories and a dependency hierarchy and some of the classes are abstract classes which are identified by the following properties:

- No objects are instances of an abstract class. They will always be instances of a subclass of the abstract class.
- Methods contained in abstract classes represent protocol common to all its subclasses. Subclasses can polymorphically redefine methods and add new data.
- Abstract classes provide a logical hierarchical organization by serving as an umbrella for related subclass of equal stature [1]

A.3 Encapsulation of Objects

In Smalltalk, the class description protocols for individual classes provide encapsulation of objects. The class description protocol consists of basic elements such as definition, private data, shared data, pool data, instance methods, class methods. Some of classes will not have all these elements. The existence of private data or shared data is determined by how the objects is represented by the class; the number and type of methods are also determined by the complexity and richness of functional abstractions to which objects of the class must respond.

- **Definition: the location of the class in the class hierarchy. list of identifiers for private, shared, and pool data objects that are part of the class.**
- **Private data: instance variables which represent the private memory of an object and they can be accessed only by instance methods.**
- **Shared data: class variables whose value are shared by all instances of the class and they can be accessed by both instance methods and class methods.**
- **Pool data: pool variables whose values are shared across multiple classes. Pool variables are contained in named pool dictionaries that the user specifically creates. To make pool variables accessible to a class and its instances, the user must modify the class specification.**
- **Instance methods: implementation details for messages to which instances of the class can respond or receive.**
- **Class methods: implementation details for messages to which the class can respond or receive. Typically they are used to initialize class variables or to create instances of the class.**

A.4 Inheritance of the Class Hierarchy

Inheritance is the Smalltalk capability which allows user to

reuse software by specializing already existing general solutions. Classes higher in the hierarchy represent more general characteristics, while classes lower in the hierarchy represent specific characteristics. Superclasses do not inherit from their subclasses; subclasses inherit from their superclasses, and subclasses in a different hierarchical subtree generally do not inherit while some implementations of Smalltalk support multiple inheritance. Things inherited by a subclass include private and shared data, instance and class methods. The same rules as the class description protocol apply for accessing private and shared data for inherited. A subclass inherits from its immediate superclass to all the way to class *Object* which is the superclass of all classes. Because a subclass has different protocol such as new data or methods from its super class, it may need to redefine methods inherited from a superclass. As a way of polymorphism, redefinition of inherited methods is called method overriding.

A.5 Polymorphism in Smalltalk

Polymorphism is a unique characteristic of object-oriented programming whereby different objects respond to the same message with their own unique behavior even though the same message selectors may exist in many classes in the Smalltalk image.

Polymorphism enhances the readability of software by allowing the introduction of entirely new classes of objects in existing applications, as long as they implement the message protocol required by the application, thus facilitating the reuse of generic code.

As an example, the message *printOn:* can be sent to any object in the Smalltalk system. The only requirement is that the details for *printOn:* be included somewhere in the hierarchy path of the object's class. Conceptually, *printOn:* implies a particular action to be taken. The concept is identical for any object; only the implementation details may be different.[1]

APPENDIX B

SUMMARY OF CLASS HIERARCHY

Protocol Summary for Class MechanicalObject

Superclass: Object

Definition: abstract super class of all of the classes objects used in kinematics

Private Data: two instance variables

name: an instance of String that is the name of all of the subclass elements

index: an instance of Integer that is used for numbering for the subclass elements

Instance Methods

name to assign name for a object

index to assign number for a object

Class Method: no class method of its own

Protocol Summary for Class Constraint

Superclass: MechanicalObject

Definition: abstract super class of all of the constraint elements

Private Data: Instance variables(name & index) are inherited from the class MechanicalObject.

Instance Method

initialize:array assigning name and index no. for each joint

Class Method: no class method of its own

Protocol Summary for Class RevoluteJoint

Superclass: Constraint

Definition: Object representing revolute joints

Private Data: six instance variables

firstBody: an instance of RigidBody for first element

xCompOfFirstNode: an instance of Float for x component of first node

yCompOfFirstNode: an instance of Float for y component of first node

secondBody: an instance of RigidBody for second element

xCompOfSecondNode: an instance of Float for x component of second node

yCompOfSecondNode: an instance of Float for y component of second node

Instance Method

changeFirstElement:aElement used for changing first rigid body

changeSecondElement:aElement used for changing second rigid body

positionCoords used for computing joint position

coordinates in the global frame

connect:firstElement with:firstNode to:secondElement

with:secondNo used for connection of rigid bodies with points

constraintEqn:time used to compute local constraint equation

velocityEqn:time used to compute local velocity equation

accelerationEqn used to compute local acceleration equation

jacobian:noOfColumn used to compute local jacobian

Class Method: no class method of its own

Protocol Summary for Class TranslationalJoint

Superclass: Constraint

Definition: Object representing translational joints

Private Data: ten instance variables

firstBody: an instance of RigidBody for first element

xCompOfFirstNode: an instance of Float for x component of first node

yCompOfFirstNode: an instance of Float for y component of first node

xCompOfSecondNode: an instance of Float for x component of second node

yCompOfSecondNode: an instance of Float for y component of second node

secondBody: an instance of RigidBody for second element

xCompOfThirdNode: an instance of Float for x component of third node

yCompOfThirdNode: an instance of Float for y component of third node

firstBodyAngle: an instance of Float for angular orientation of first element

secondBodyAngle: an instance of Float for angular orientation of second element

Instance Methods

changeFirstElement:aElement used for changing first rigid body

changeSecondElement:aElement used for changing second rigid body

positionCoords used for computing joint position coordinates in the global frame

connect:firstElement with:firstNode to:secondElement with:secondNo used for connection of rigid bodies with points

constraintEqn:time used to compute local constraint equation

velocityEqn:time used to compute local velocity equation

accelerationEqn used to compute local acceleration equation

jacobian:noOfColumn used to compute local jacobian

Class Method: no class method of its own

Protocol Summary for Class SimpleConstraint

Superclass: Constraint

Definition: Object representing simple constraints

Private Data: three instance variables

constrainedDirection: an instance of Integer for constrained direction

element: an instance of RigidBody for constrained element

constant: an instance of Float for constrained constant quantities

Instance Methods

columnIndex used for computing column number for the jacobian matrix

isOn:body direction:string with:aConstant used for setting values of the instance variables

constraintEqn:time used to compute local constraint equation

velocityEqn:time used to compute local velocity equation

accelerationEqn used to compute local acceleration equation

jacobian:noOfColumn used to compute local jacobian

Class Method: no class method of its own

Protocol Summary for Class DrivingLink

Superclass: Constraint

Definition: Object representing driving link

Private Data: five instance variables

constrainedDirection: an instance of Integer for constrained direction

element: an instance of RigidBody for driving link element

initialPosition: an instance of Float for initial position

initialVelocity: an instance of Float for initial velocity

initialAcceleration: an instance of Float for initial acceleration

Instance Methods

columnIndex used for computing column number for the jacobian matrix

isOn:body direction:string with:aConstant used for setting values of the instance variables

constraintEqn:time used to compute local constraint equation

velocityEqn:time used to compute local velocity equation

accelerationEqn used to compute local acceleration equation

jacobian:noOfColumn used to compute local jacobian

Class Method: no class method of its own

Protocol Summary for Class Element

Superclass: MechanicalObject

Definition: Object representing mechanical elements

Private Data: six instance variables

nodeList: an instance of OrderedCollection for storing nodes

elementList: an instance of OrderedCollection for storing rigid body elements

constraintList: an instance of OrderedCollection for storing

constraint elements

positionVector: an instance of Vector for the cartesian generalized position vector

velocityVector: an instance of Vector for the cartesian generalized velocity vector

accelerationVector: an instance of Vector for the cartesian generalized acceleration vector

Instance Methods

constraintEqn:time used to compute system constraint equation

velocityEqn:time used to compute system velocity equation

accelerationEqn used to compute system acceleration equation

jacobian:noOfColumn used to compute system jacobian matrix

reportNodes used for reporting position, velocity, and acceleration of interesting nodes of rigid body on the screen

reportNodes:outputFile used for reporting position, velocity, and acceleration of interesting nodes of rigid body to output file

positionVector used for computing the cartesian generalized position vector

velocityVector used for computing the cartesian generalized velocity vector

accelerationVector used for computing the cartesian generalized acceleration vector

changePosition:array used for replacing the components of position vector

changeVelocity:array used for replacing the components of velocity vector

changeAcceleration:array used for replacing the components of acceleration vector

analysis:time used for position, velocity, acceleration analysis at each time step

improvedKinematicAnalysisFrom:startingTime to:finalTime with:timeIncrement used for the kinematic analysis for overall time span

initialize:array used for setting instance variables such as elementList, constraintList, and name

positionAnalysis:time used for Newton iteration method at each time step

addElement:element used for adding an element

addConstraint:constraint used for adding a constraint

removeElement:element used for removing an element

removeConstraint:constraint used for removing a constraint

getElement:element used for accessing a specific element

getConstraint:constraint used for accessing a specific constraint

combine:aElement with:bElement used to combine two elements

resequenceElement used for rearranging order of instance variable elementList

Class Method: one class method of its own

examples shows how to simulate sample model of each mechanism

Protocol Summary for Class RigidBody

Superclass: Element

Definition: Object representing rigid bodies

Private Data: 11 instance variables

x: an instance of Float for x coordinate of rigid body

y: an instance of Float for y coordinate of rigid body

angle: an instance of Float for angular orientation

xVelocity: an instance of Float for x component of velocity

yVelocity: an instance of Float for y component of velocity

angularVelocity: an instance of Float for angular velocity

xAcceleration: an instance of Float for x component of acceleration

yAcceleration: an instance of Float for y component of acceleration

angularAcceleration: an instance of Float for angular acceleration

rotationalCoords: an instance of Array for storing rotational coordinates

interestingNodeList: an instance of OrderedCollection for storing interesting point

Instance Methods

addNode used for storing joint points in the rigid bodies

addInterestingNode used for storing interesting points in the rigid body

interestingNodes used for reporting position, velocity, and acceleration of interesting points of rigid body on the screen

interestingNodes:file used for reporting position, velocity, and acceleration of interesting points of rigid body in the output file

rotation used for computing rotational coordinates of rigid body elements

initialize:array used for setting instance variables such as nodeList, interestingNodeList, name, index x, y, angle, rotationalCoords

getNodesFrom:element used for transferring nodes between rigid bodies

Class Method: no class method of its own

Protocol Summary for Class FourBar

Superclass: Element

Definition: Object representing four bar mechanism

Private Data: no instance variables of its own

Instance Method: no instance methods of its own

Class Methods

model1 used to create sample model 1 of four bar mechanism

model2 used to create sample model 2 of four bar mechanism

Protocol Summary for Class FourSlider

Superclass: Element

Definition: Object representing combined four bar and slider crank mechanism

Private Data: no instance variables of its own

Instance Method: no instance methods of its own

Class Methods

model1 used to create sample model 1 of fourSlider mechanism

Protocol Summary for Class SliderCrank

Superclass: Element

Definition: Object representing slider crank mechanism

Private Data: no instance variables of its own

Instance Method: no instance methods of its own

Class Methods

model1 used to create sample model 1 of slider crank mechanism

model2 used to create sample model 2 of slider crank mechanism

Protocol Summary for Class QuickReturn

Superclass: Element

Definition: Object representing quick return mechanism

Private Data: no instance variables of its own

Instance Method: no instance methods of its own

Class Method

    model1 used to create sample model 1 of quick return mechanism

Protocol Summary for Class Node

Superclass: MechanicalObject

Definition: Object representing points

Private Data: two instance variables

    x: an instance of Float for x coordinate

    y: an instance of Float for y coordinate

Instance Method

    initialize:array used to set values for instance variables
                    such as name, index, x, y

Class Method: no class method of its own

Protocol Summary for Class Matrix

Superclass: FixedSizeCollection

Definition: Object representing matrices

Private Data: no instance variables

Instance Methods

linear:vector used for solving the linear system equations

luFactorization used for performing L-U factorization on the given matrix

row:aI col:aJ used to access specific element of matrix

row:aI col:aJ put:aObject are used for replacing specific element of matrix

add:aMatrix used for adding two matrices

product:aMatrix used for multiplying two matrices

scale:aNumber used for scaling a matrix

subtract:aMatrix used for subtracting two matrices

Class Method

row:rDim col:cDim used for creating an instance of matrix whose element is initialized to zero value

Protocol Summary for Class Vector

Superclass: Matrix

Definition: Object representing vectors

Private Data: one instance variable

compList: an instance of Array for storing the vector element

Instance Methods

components used for accessing vector elements

components:array used for replacing elements of given vector

add:bVector used for adding two vectors

innerProduct:bVector used for computation of dot product of

two vectors

scale:aNumber used for scaling of a vector

subtract:bVector used for subtracting two vectors

Class Method: no class method of its own

LIST OF REFERENCES

LIST OF REFERENCES

[1] Pinson, L.J. and R.S. Wiener. An Introduction to Object-Oriented Programming and Smalltalk. Reading, Massachusetts: Addison-Wesley Publishing Co., 1988.

[2] Cox, B.J. Object-Oriented Programming. Reading, Massachusetts: Addison-Wesley Publishing Co., 1987.

[3] Goldberg, Adele. Smalltalk-80 The Interactive Programming Environment. Reading, Massachusetts: Addison-Wesley Publishing Co., 1984.

[4] Smalltalk/V Tutorial and Programming Handbook. Los Angeles, California: Digitalk Inc., 1986.

[5] Ungar, David M. The Design and Evaluation of a High Performance Smalltalk System. Cambridge, Massachusetts: The MIT Press, 1987

[6] Kaehler, Ted. and D. Patterson. A Taste of Smalltalk. New York, N.Y.: W. W. Norton & Company, Inc., 1986.

[7] Haug, E.J. Computer Aided Kinematics and Dynamics of Mechanical Systems Volume I Basic Methods. Iowa City, Iowa: Dept. of Mechanical Engineering, 1985.

[8] Haug, E.J. Computer Aided Analysis and Optimization of Mechanical System Dynamics. Berlin: Springer-Verlag, 1984.

[9] Nikravesh, P.E. Computer-Aided Analysis of Mechanical Systems. Prentice-Hall, EngleWood Cliffs, N.J., 1987.

[10] Garnham, Alan. Artificial Intelligence An Introduction. New York: Routledge & Kegan Paul, 1988.

[11] Rich, E. Artificial Intelligence. New York, N.Y.: McGraw-Hill Book Co., 1983.

[12] Johnson, Lee W. and Dean R. Riess. Numerical Analysis. Reading, Massachusetts: Addison-Wesley Publishing Co., 1982.

[13] Burden, Richard L. (et al) Numerical Analysis. Boston, Massachusetts: Prindle, Weber & Schmidt, 1978.

[14] DRAM User's Guide. Mechanical Dynamics, inc., 1978.

[15] ADAMS 5.0 User's Guide. Mechanical Dynamics, inc., 1985.

[16] Dittrich, Klaus and Umeshwar Dayal.(ed.) Proceedings: 1986 International Workshop on Object-Oriented Database Systems. ACM, 1986.

[17] Paul, B. Kinematics and Dynamics of Planar Machinery. Prentice-Hall, EngleWood Cliffs, N.J., 1979.

[18] Sandor, G.H. and A.G. Erdman. Advanced Mechanism Design: Analysis and Synthesis. Prentice-Hall, EngleWood Cliffs, N.J., 1984.

[19] Dally, William J. "Concurrent Computer Architecture" Cambridge, Massachusetts: MIT.

[20] Peterson, Robert W. "Object-Oriented Data Base Design" AI expert: March, 1987.

[21] White, Eva and Rich Malloy.(ed.) "Object-Oriented Programming" Byte. August, 1986. p. 137.

[22] Pascoe, G.A. "Elements of Object-Oriented Programming" Byte. August, 1986. pp. 139-144.

[23] Robson, David. "Object-Oriented Software System" Byte. August, 1981. pp. 74-86.

[24] Schmucker, K.J. "Object-Oriented Languages for the Macintosh" Byte. August, 1986. pp. 177-185.

[25] Tesler, L. "Programming Experiences" Byte. August, 1986. pp.195-206.

[26] Duff, C.B. "Designing an Efficient Languages" Byte. August, 1986. pp. 211-224.

[27] Byte volume 6, number 8, August 1981.

[28] Shaw, M. "The Impact of Abstraction Concerns on Modern Programming Languages" The Proceeding of the IEEE, vol.68, No.9, September 1986.

[29] Dahl, O.J., E.W. Dijkstra, and C.A.R. Hoare. Structured Programming. Academic Press, London. 1972.