Development and Validation of MATLAB-Based 3D Multibody Kinematics and Dynamics Simulation Engine

Michael F. Vignos
mvignos@wisc.edu

University of Wisconsin-Madison

Developed for Mechanical Engineering 751: Advanced Computational Multibody Dynamics
December 15, 2016

Table of Contents

1. Introduction

The purpose of this document is to give the reader an overview of the functionality of simEngine3D. The following sections include discussions of how to use simEngine3D, the capabilities and features of this multibody simulation engine, and an overview of efforts to validate simEngine3D. This document is not intended to give an in-depth, technical description of the theory applied to develop simEngine3D and, thus, assumes the reader has a general understanding of computational multibody dynamics. For a more thorough description of multibody dynamics theory, please consult the following references:

1. Haug, Edward J. *Computer Aided Kinematics and Dynamics of Mechanical Systems*. Boston: Allyn and Bacon, 1989. Print.
2. Negrut, Dan. *Mechanical Engineering 751: Advanced Computational Multibody Dynamics.* University of Wisconsin-Madison, Madison, WI. Fall 2016. Lecture.

2. Problem Statement

The goal of this work was to develop and validate a straight-forward and easy to use 3D multibody kinematics and dynamics simulation engine within MATLAB. This simulation engine was intended to allow users to perform kinematics, inverse dynamics, and dynamics analyses of multibody systems consisting of bodies, joints connecting the bodies, and externally applied forces and torques.

3. simEngine3D Framework

As previously mentioned, all code for simEngine3D was developed using MATLAB R2014b. It is likely that simEngine3D will perform properly for future versions of MATLAB as the functions used are not version specific. However, this has not been verified.

3.1 Organization of GitHub Repository

All of the code for simEngine3D is maintained in the GitHub repository named simEngine3D-Vignos. Within this repository there are 3 folders that are of interest to the reader: *simEngine3DCode*, *testExamples*, and *ME751assignments*. The folder *simEngine3DCode* contains all of the MATLAB code that is used to run this simulation engine. The folder *testExamples* contains example driver files that were used to validate this simulation engine. These files contain examples of kinematics, dynamics, and inverse dynamics analyses. The results of these driver files were compared to results of either other validated simulation packages, the results reported in *Computer Aided Kinematics and Dynamics of Mechanical Systems* (1), or analytical results computed by hand for validation. The folder *ME751assignments* contains homework assignments that were completed as benchmarks when developing this code for a course at UW-Madison, but are likely no longer useful.

3.2 Object-Oriented Programming

The framework of simEngine3D was written using object-oriented programming. Using object-oriented programming allows for the creation of MATLAB classes. These classes can then be leveraged by the user to develop a multibody system in a hierarchical manner (Fig. 1). The classes that exist in this code are contained within the folder *simEngine3DCode* and are as follows: *multibodySystem.m, body.m, CDconstraint.m, Dconstraint.m, DP1constraint.m, DP2constraint.m,* and *simEngine3DUtilities.m*. In addition to these classes, there is also a *plot* folder that contains various functions that can be used to

display and animate a multibody system. The purpose of each class and the *plot* commands will be covered in a bit more detail in the following sections.

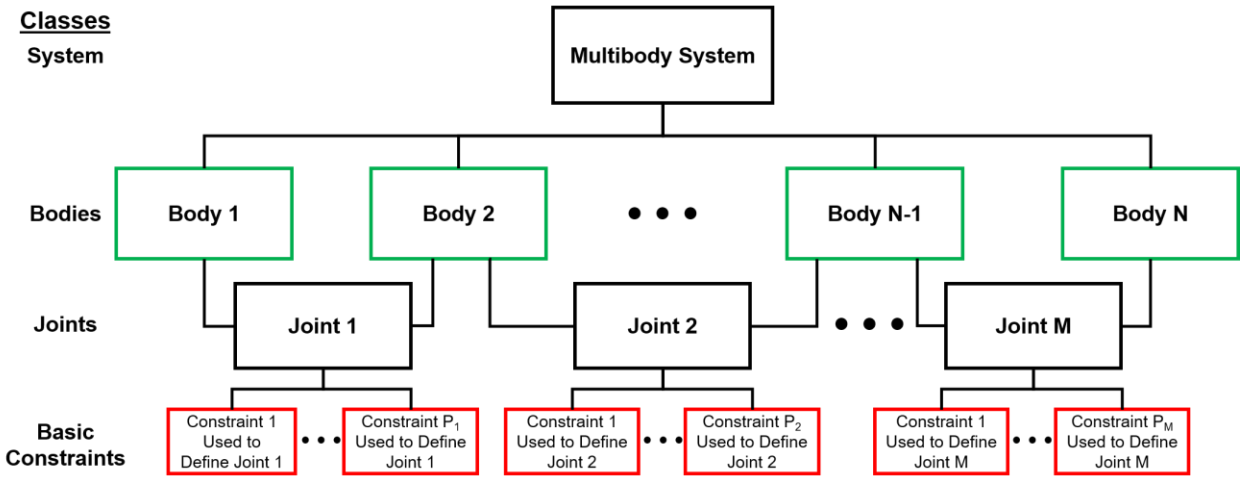


Figure 1: Outline of hierarchical structure created through the use of classes in simEngine3D. The highest level class is the multibody system. A multibody system is made of a set of bodies, which are constrained to each other using joints. Each joint is then comprised of a set of basic constraints. This set of basic constraints varies depending on which joint is used.

3.2.1. multibodySystem Class

An instance of this class contains all of the bodies, constraints, and externally applied forces and torques that are used to perform a simulation. This class also contains all of the methods needed to perform kinematics, inverse dynamics, and dynamics analyses. Following completion of a simulation, the instance of this class used to perform the simulation will also contain all of the state information stored throughout the simulation (e.g. kinematics of each body, reaction forces and torques, etc.). Additionally, since this information is stored at each time step, a simulation can be stopped prematurely and the state information contained within the instance of this class can be visualized. In other word, the data for a simulation can still be visualized even if the simulation is stopped early.

3.2.2. body Class

An instance of this class contains all of the attributes of a body (e.g. mass, body number, kinematics, etc.) that are required to perform a simulation. This class also contains functions that are commonly used to compute variables related to a single body (e.g. the orientation matrix of a body, the total torque applied to a body, etc.). Additionally, the state information throughout a simulation is stored within a body class. This approach of having each body in the system defined as its own class allows for improved organization of the system and allows for easy visualization of state information for a single body in post-processing.

3.2.3. Basic Constraint Classes: CDconstraint, Dconstraint, DP1constraint, and DP2constraint

Instances of each of these classes are similar in that they define the bodies impacted by each constraint, they contain the attributes of each of the four basic constraints, and they contain functions used to compute attributes of these constraints that are needed when performing a simulation (e.g. current state of the constraint, the right hand side of the acceleration equation, partial derivatives of the constraint, etc.). All higher level constraints (i.e. joints) used in defining a multibody system are composed of instances of these basic constraints.

3.2.4. simEngine3DUtilities Class

This class is simply a collection of functions that are commonly used when performing a simulation (e.g. computing the distance between two points, computing a skew symmetric matrix from a vector, etc.).

3.2.5 plot Folder

This folder contains a collection of functions that can be used to display the position of all bodies in a system at a specific state or to create an animation to visualize the output of a simulation.

3.3 Example Model Definition

Below is a screen shot of an example model definition within a driver script (Fig 2). This model is a simple pendulum with two bodies (the mass and the ground) and a revolute joint defined between them. The revolute joint is actually composed of 5 basic constraints. When the user tells the multibody system to add the revolute joint, the system automatically adds these 5 basic constraints based on the attributes provided by the user. This concept of defining joints using basic constraints will be further discussed in the following section. More thorough and complex examples of model definitions can be found in the *testExamples* folder in the *simEngine3D-Vignos* repository.

```
%% simplePendulum.m
% Taken from: http://lim.ii.udc.es/mbsbenchmark/dist/A01/A01_specification.xml.
clear; close all; clc;

%% Create instance of multibody system
sys = multibodySystem();

%% Define bodies and properties of bodies in system
% Add body1. This is the ground.
mass1 = 1;
length1 = 0;
isGround1 = 1;
JMatrix1 = eye(3,3);
gravityDirection = '-y';
sys.addBody(1, 'ground', isGround1, mass1, length1, JMatrix1, gravityDirection);

% Add body2. The is the mass.
length2 = 0.0; % meters. This is actually the radius of the wheel.
mass2 = 1.0; % kg

% Define inertial properties of the mass
Jxx = 0.1;
Jyy = 0.1;
Jzz = 0.1;
JMatrix2 = zeros(3,3);
JMatrix2(1,1) = Jxx;
JMatrix2(2,2) = Jyy;
JMatrix2(3,3) = Jzz;

isGround2 = 0;
sys.addBody(2, 'point', isGround2, mass2, length2, JMatrix2, gravityDirection);

%% Define revolute joint between ground and point mass
a1.body1 = 1;
a1.body2 = 2;
a1.pointOnBody1 = [0 0 0]';
a1.pointOnBody2 = [1 0 0]';
a1.vector1OnBody1 = [1 0 0]';
a1.vector2OnBody1 = [0 1 0]';
a1.vectorOnBody2 = [0 0 1]';
a1.constraintName = 'Revolute Joint b/w Ground and Mass';

sys.addJoint('revolute',a1);
```

Figure 2: Example driver file showing the definition of a simple pendulum. This example contains two bodies (the ground and a point mass) and a revolute joint between them.

4. System Constraints

4.1. Kinematic Constraints

Kinematic constraints are constraints that are added into a multibody system to define the geometry of the motion of the system. In other words, kinematic constraints between two bodies define the relative motion that is allowed between these bodies (i.e. they remove degrees of freedom of each body). Within simEngine3D there are three different levels of constraints: basic constraints, intermediate constraints, and joints. Both intermediate constraints and joints are made up of basic constraints. When developing a model, it is most common to define constraints between bodies using joints, as the definition of these joints are most easily understood.

4.1.1. Basic Constraints

As previously discussed, there are four basic constraints that are used as the foundation to all kinematic constraints in simEngine3D. These constraints are a coordinate distance constraint (CD constraint), a distance constraint (D constraint), a dot product one constraint (DP1 constraint), and a dot product two constraint (DP2 constraint). Each of these basic constraints removes one degree of freedom from a multibody system when it is added into the system meaning they are scalar quantities.

Coordinate Difference (CD) Constraint

A CD constraint defines the distance between two bodies for a single Cartesian coordinate (x, y, or z) (Fig 3A). The attributes for this constraint are as follows:

1. The two bodies to which the constraint is applied (body i and body j).
2. A point attached to body i in the body i reference frame ($\bar{s}_i^P$).
3. A point attached to body j in the body j reference frame ($\bar{s}_j^Q$).
4. The Cartesian coordinate that is being constrained (e.g. to constrain x, c = $[1\ 0\ 0]^T$)
5. The prescribed distance between the coordinates of the two bodies, f(t).

Distance (D) Constraint

A D constraint defines the distance between two bodies (Fig 3B). The attributes for this constraint are as follows:

1. The two bodies to which the constraint is applied (i.e. body i and body j).
2. A point attached to body i in the body i reference frame ($\bar{s}_i^P$).
3. A point attached to body j in the body j reference frame ($\bar{s}_j^Q$).
4. The prescribed squared distance between the two bodies ($f(t)^2$). It is important to note that in the implementation of this constraint the user must provide the square of the distance.

Dot Product One (DP1) Constraint

A DP1 constraint constrains the dot product between two vectors, one attached to body i and one attached to body j, to be equal to a prescribed function (Fig 3C). This function is commonly set to zero, which constrains the two vectors to be orthogonal. The attributes for this constraint are as follows:

1. The two bodies to which the constraint is applied (i.e. body i and body j)
2. A vector attached to body i in the body i reference frame ($\bar{a}_i$).
3. A vector attached to body j in the body j reference frame ($\bar{a}_j$).

4. A function prescribing the value of the dot product between the two vectors ($f(t)$). If these vectors are defined to be orthogonal, this function must be zero.

Dot Product Two (DP2) Constraint

Similar to a DP1 constraint, a DP2 constraint constrains the dot product between two vectors to be equal to a prescribed function (Fig 3D). However, in this case one vector is attached to body i and the second vector is defined as the vector between two points. One of these points is attached to body i and the other is attached to body j. The attributes for this constraint are as follows:

1. The two bodies to which the constraint is applied (i.e. body i and body j)
2. A vector attached to body i in the body i reference frame ($\bar{a}_i$)
3. A point attached to body i in the body i reference frame ($\bar{s}_i^P$)
4. A point attached to body j in the body j reference frame ($\bar{s}_j^Q$)
5. A function prescribing the value of the dot product between the two vectors($f(t)$). If these vectors are defined to be orthogonal, this function must be zero.

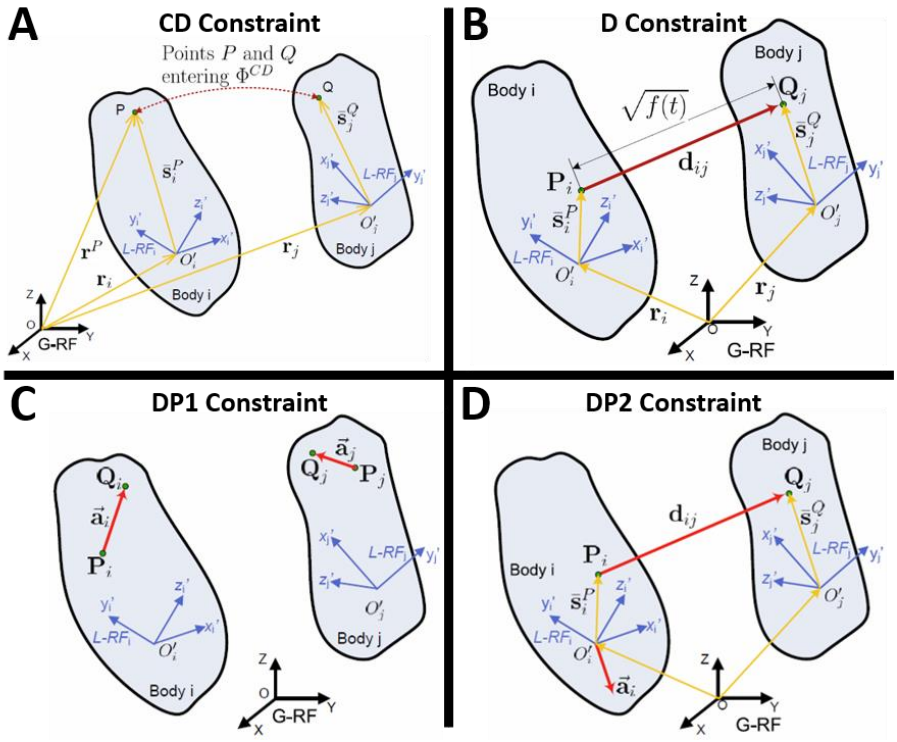


Figure 3: Figures showing the attributes of the basic constraints. These figures were taken from reference [2] in the Introduction.

4.1.2. Intermediate Constraints

In simEngine3D there are two intermediate constraints: perpendicular one (B1) constraint and perpendicular two (B2) constraint. Both of these constraints are composed of two basic constraints.

Perpendicular One (B1) Constraint

A B1 constraint constrains a vector attached to body j to be orthogonal to a plane attached to body i. The plane on body i is defined by two vectors, attached to body i (Fig 4A). The attributes for this constraint are as follows:

1. The two bodies to which the constraint is applied (i.e. body i and body j)
2. A vector attached to body i in the body i reference frame $(\bar{a}_i)$
3. Another vector attached to body i in the body i reference frame$(\bar{b}_i)$. This vector and the previous vector define the plane on body i.
4. A vector attached to body j in the body j reference frame $(\bar{c}_j)$

Perpendicular Two (B2) Constraint

Similar to a B1 constraint, a B2 constraint constrains a vector to be orthogonal to a plane attached to body i (Fig 4B). However, in this case the vector is defined as the vector between a point attached to body i and a point attached to body j. The attributes for this constraint are as follows:

1. The two bodies to which the constraint is applied (i.e. body i and body j)
2. A vector attached to body i in the body i reference frame $(\bar{a}_i)$
3. Another vector attached to body i in the body i reference frame$(\bar{b}_i)$. This vector and the previous vector define the plane on body i.
4. A point attached to body i in the body i reference frame $(\bar{s}_i^P)$
5. A point attached to body j in the body j reference frame $(\bar{s}_j^Q)$

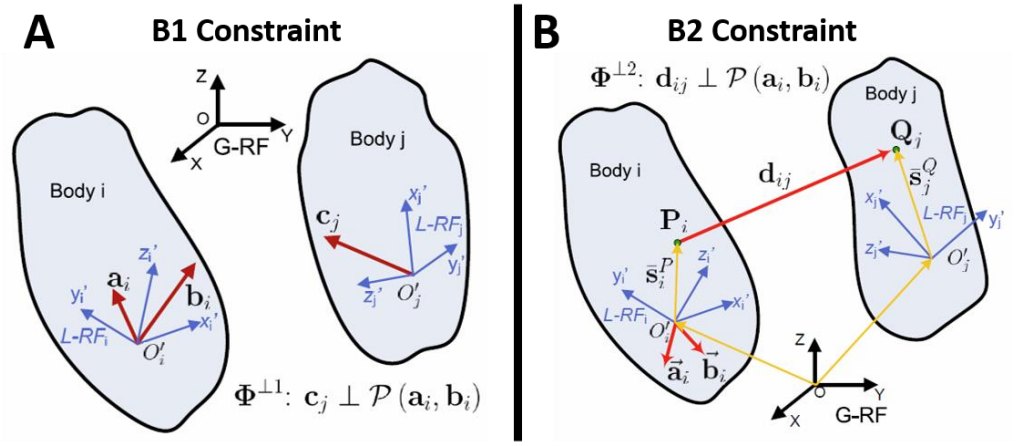


Figure 4: Figures showing the attributes of the intermediate constraints. These figures were taken from reference [2] in the Introduction.

4.1.3. Joints

To date, there are six joints that have been implemented and validated in simEngine3D. Each of these joints is constructed from the set of basic constraints. Below is a description of the attributes needed to define each of these joints. However, to gain a better understanding of how the basic constraints are used to create each of these joints, please see the methods in the *multibodySystem* class used to create each of these joints (e.g. *createSphericalJoint(), createCylindricalJoint(),* etc.)

Spherical Joint Attributes (Fig 5)

1. The two bodies connected by this joint (i.e. body i and body j)
2. A point attached to body i in the body i reference frame$(\bar{s}_i^P)$.
3. A point attached to body j in the body j reference frame $(\bar{s}_j^Q)$. This second point will be constrained to be in the same location as the point defined in j.

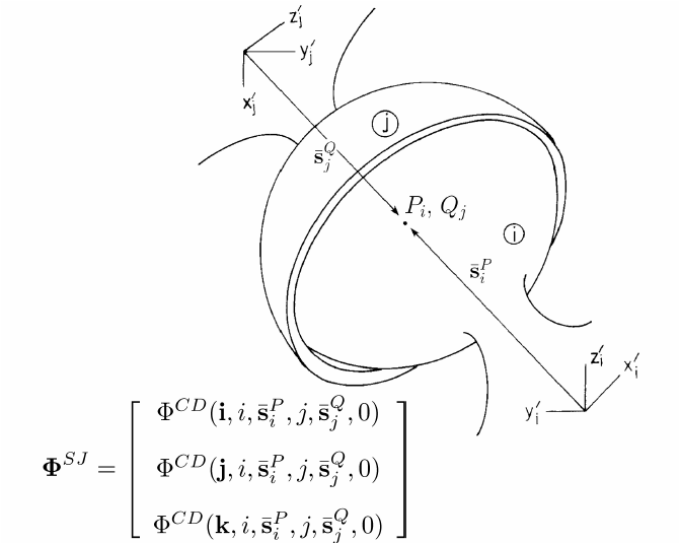


Figure 5: Attributes of a spherical joint. Figure from reference [2] in the Introduction.

Revolute Joint Attributes (Fig 6)

1. The two bodies connected by this joint (i.e. body i and body j)
2. A point attached to body i in the body i reference frame ($\bar{s}_i^P$).
3. A point attached to body j in the body j reference frame ($\bar{s}_j^Q$). This second point will be constrained to be in the same location as the point defined in j.
4. A vector attached to body i in the body i reference frame ($\bar{a}_i$). This vector is the 1st vector used to define the plane in which the revolute joint moves.
5. A second vector attached to body i in the body i reference frame($\bar{b}_i$). This vector is the 2nd vector used to define the plane in which the revolute joint moves.
6. A vector attached to body j in the body j reference frame ($\bar{c}_j$). This vector is orthogonal to the plane on body i (i.e. the plane defined by the vectors created in 4 and 5). In other words, this is the vector about which the revolute joint rotates.

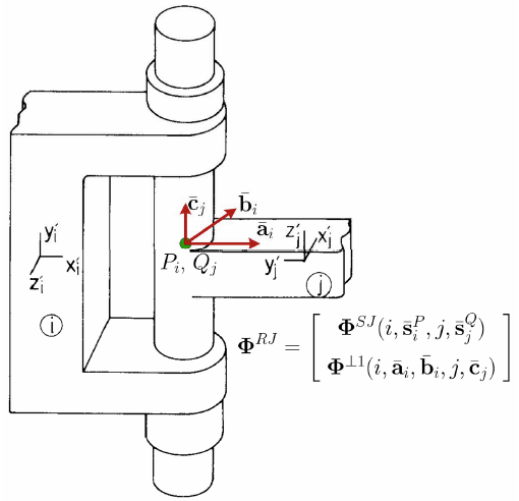


Figure 6: Attributes of a revolute joint. Figure from reference [2] in the Introduction.

Cylindrical Joint Attributes (Fig 7)

1. The two bodies connected by this joint (i.e. body i and body j)
2. A point attached to body i in the body i reference frame along the translational axis of body i ($\bar{s}_i^P$).
3. A point attached to body j in the body j reference frame along the translational axis of body j ($\bar{s}_j^Q$).
4. A vector attached to body i in the body i reference frame ($\bar{a}_i$). This vector is the 1st vector used to define the plane orthogonal to the translational axis of this joint.
5. A second vector attached to body i in the body i reference frame ($\bar{b}_i$). This vector is the 2nd vector used to define the plane orthogonal to the translational axis of this joint.
6. A vector attached to body j in the body j reference frame ($\bar{c}_j$). This vector is orthogonal to the plane on body i (i.e. the plane defined by the vectors defined in 4 and 5).

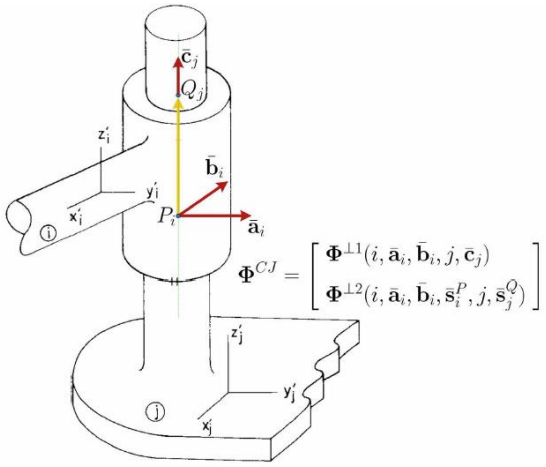
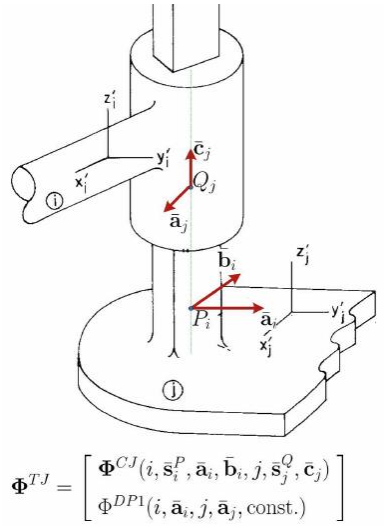


Figure 7: Attributes of a cylindrical joint. Figure from reference [2] in the Introduction.

Translational Joint Attributes (Fig 8)

1. The two bodies connected by this joint (i.e. body i and body j)
2. A point attached to body i in the body i reference frame along the translational axis of body i ($\bar{s}_i^P$).
3. A point attached to body j in the body j reference frame along the translational axis of body j ($\bar{s}_j^Q$).
4. A vector attached to body i in the body i reference frame ($\bar{a}_i$). This vector is the 1st vector used to define the plane orthogonal to the translational axis of this joint.
5. A second vector attached to body i in the body i reference frame ($\bar{b}_i$). This vector is the 2nd vector used to define the plane orthogonal to the translational axis of this joint.
6. A vector attached to body j in the body j reference frame ($\bar{c}_j$). This vector is orthogonal to the plane on body 1 (i.e. the plane defined by the vectors defined in 4 and 5).
7. A second vector attached to body j in the body j reference frame ($\bar{a}_j$). This vector is parallel to the plane on body i (i.e. the plane defined by the vectors defined in 4 and 5) and perpendicular to the vector defined in 4.

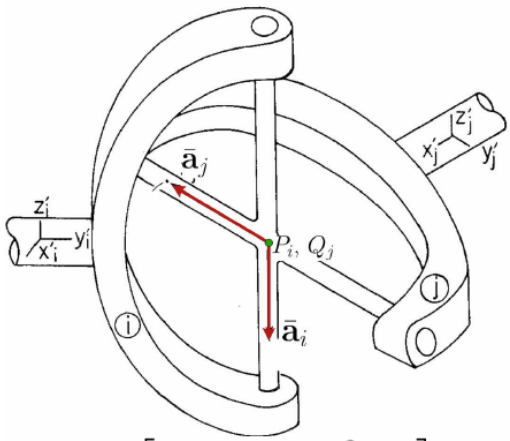


$$\mathbf{\Phi}^{TJ} = \begin{bmatrix} \mathbf{\Phi}^{CJ}(i, \bar{\mathbf{s}}_i^P, \bar{\mathbf{a}}_i, \bar{\mathbf{b}}_i, j, \bar{\mathbf{s}}_j^Q, \bar{\mathbf{c}}_j) \\ \Phi^{DP1}(i, \bar{\mathbf{a}}_i, j, \bar{\mathbf{a}}_j, \text{const.}) \end{bmatrix}$$

Figure 8: Attributes of a translational joint. Figure from reference [2] in the Introduction.

Universal Joint Attributes (Fig 9)

1. The two bodies connected by this joint (i.e. body i and body j)
2. A point attached to body i in the body i reference frame ($\bar{s}_i^P$).
3. A point attached to body j in the body j reference frame ($\bar{s}_j^Q$). This second point will be constrained to be in the same location as the point defined in j.
4. A vector attached to body i in the body i reference frame that defines the rotation axis of this joint for body i ($\bar{a}_i$).
5. A vector attached to body j in the body j reference frame that defines the rotation axis of this joint for body j ($\bar{a}_j$). This vector is orthogonal to the vector defined in 4.



$$\mathbf{\Phi}^{UJ} = \begin{bmatrix} \mathbf{\Phi}^{SJ}(i, \bar{\mathbf{s}}_i^P, j, \bar{\mathbf{s}}_j^Q) \\ \Phi^{DP1}(i, \bar{\mathbf{a}}_i, j, \bar{\mathbf{a}}_j, 0) \end{bmatrix}$$

Figure 9: Attributes of a universal joint. Figure from reference [2] in the Introduction.

Revolute-Cylindrical Joint Attributes (Fig 10)

Note: When defining a revolute-cylindrical joint, the two origins of the bodies that are connected by this joint typically also need to be constrained to one another using an additional distance constraint.

1. The two bodies connected by this joint (i.e. body i and body j)
2. A point attached to body i in the body i reference frame along the translational axis of body i ($\bar{s}_i^P$).
3. A point attached to body j in the body j reference frame along the translational axis of body j ($\bar{s}_j^Q$).
4. A vector attached to body i in the body i reference frame that defines the rotational axis of the revolute portion of the joint ($\bar{h}_i$).
5. A vector attached to body j in the body j reference frame that is along the translational axis of this joint ($\bar{h}_j$). This vector is orthogonal to the vector defined in 4.
6. A vector attached to body j in the body j reference frame ($\bar{a}_j$). This vector is the 1st vector used to define the plane orthogonal to the translational axis of this joint. This vector is also orthogonal to the vector defined in 5. This vector is not shown on Figure 10, but is similar to vector $\bar{a}_i$ for the cylindrical and translational joints.
7. A second vector attached to body j in the body j reference frame ($\bar{b}_j$). This vector is the 2nd vector used to define the plane orthogonal to the translational axis of this joint. This vector is also orthogonal to the vector defined in 5. This vector is not shown on Figure 10, but is similar to vector $\bar{b}_i$ for the cylindrical and translational joints.

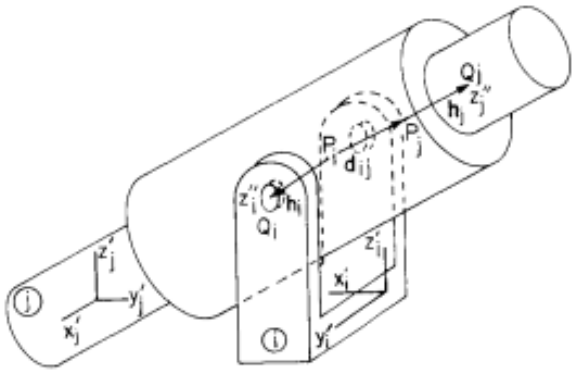


Figure 10: Attributes of a revolute-cylindrical joint. Figure from reference [1] in the Introduction.

4.2. Driving Constraints

A driving constraint is used to specify the time-dependent behavior of one of the bodies in the multibody system. To date, all driving constraints in simEngine3D are prescribed by providing a time-dependent function for a basic constraint. As described in the Basic Constraint section, each basic constraint has a function as an input that defines the value of the constraint. By setting this input to be a time-dependent function, a basic constraint can be used as a driving constraint. For example, a DP1 constraint can be used to drive a revolute joint by prescribing the angle of the free degree of freedom of this joint as a function of time. For an example of using a DP1 constraint as a driving constraint, see the example *testExamples/uniqueFourBarMechanism/uniqueFourBarMechanismDynamicsAnalysis.m.*

4.3. Adding Constraints to a Multibody System

Adding constraints to a multibody system must be done after creating an instance of the multibody system class and adding all of the bodies to this class. Constraints are then added by calling either *addBasicConstraint*(), *addIntermediateConstraint*(), or *addJoint*(), depending on the type of constraint the user wants to add.

```
%% Define revolute joint
a1.body1 = 1;
a1.body2 = 2;
a1.pointOnBody1 = [0 0 0]';
a1.pointOnBody2 = [0 0 0]';
a1.vector1OnBody1 = [0 1 0]';
a1.vector2OnBody1 = [0 0 1]';
a1.vectorOnBody2 = [1 0 0]';
a1.constraintName = 'Revolute Joint b/w Ground and Wheel';

sys.addJoint('revolute',a1);
```

```
%% Add basic constraint
a2.bodyJ = 1;
a2.bodyI = 2;
a2.aBarJ = [0 1 0]';
a2.aBarI = [0 0 1]';
a2.ft = @(t)0;
a2.ftDot = @(t)0;
a2.ftDDot = @(t)0;
a2.constraintName = 'DP1 constraint b/w bodies 1 and 2';
isKinematic = 1;
sys.addBasicConstraint(isKinematic,'dp1',a2);
```

```
%% Add intermediate constraint
a3.bodyI = 2;
a3.bodyJ = 3;
a3.aBarI = [1 0 0]';
a3.bBarI = [0 1 0]';
a3.cBarJ = [0 0 1]';
isKinematic = 1;
obj = addIntermediateConstraint(isKinematic,'b1',a3);
```

Figure 11: Portion of a driver file showing examples of how to add constraints to a multibody system.

4.4. Assembling the Constraints

Prior to performing any analysis of the multibody system, simEngine3D assembles all of the user-defined constraints into a constraint vector (Φ). Each row of this vector contains a single entry for all basic constraints added into the system. In the case of a joint or an intermediate constraint being added to the system, each joint and intermediate constraint is defined by a set of basic constraints. For example, if a joint made of five basic constraints was added to the system, this joint would be defined in Φ by five rows. This assembled constraint vector is then used for all analyses discussed in section 6 Analysis of Mechanisms.

5. Externally Applied Forces and Torques

Within simEngine3D there are three different options for applying external forces and torques to a body. These options consist of applying a constant force or torque, applying a variable torque, or using a translational-spring-damper-actuator.

5.1. Constant Forces and Torques

Constant forces are externally applied forces that do not change magnitude or direction, with respect to the global reference frame, throughout an entire simulation (Fig 12). Constant torques are externally applied torques that do not change magnitude or direction, with respect to the body reference frame, throughout a simulation. With this in mind, the direction of constant forces must be specified in the global reference frame and the direction of constant torques must be defined in the body reference frame.

A special case of a constant force is the gravity force acting on each body. The user does not need to add the gravity force to each body in a multibody system. The force of gravity is added to each body when the body is added to the system (Fig 1). However, the user must properly specify the body mass and the gravity direction to ensure proper magnitude and direction of the gravity force.

```
%% Add force
bodyNumber = 2;
force = [0 -10 0]';
locationToApplyForce = [1 0 0]';
forceName = 'Force applied to body2';
sys.addForce(bodyNumber, force, locationToApplyForce, forceName);

%% Add torque
bodyNumber = 2;
torque = [10 0 0]';
torqueName = 'Torque applied to body2';
sys.addTorque(bodyNumber, torque, torqueName);
```

Figure 12: Portion of a driver file showing how a constant force (top) and a constant torque (bottom) are added into a multibody system. The variable sys is an instance of the multibody system class.

5.2. Variable Torques

A variable torque is a torque that changes magnitude, but maintains a constant direction, with respect to the body reference frame, through a simulation. The behavior of this torque is defined by a user supplied MATLAB function that defines the magnitude of the torque as a function of the multibody system state and/or time (Fig 13). It is important to note that, although the torque magnitude does not need to depend on both the system state and time, both the instance of the *multibodySystem* class and the current time of the system must be provided as inputs to this function (Fig 14).

```
%% Add a variable torques to apply to the axis.
bodyNumber = 2;
torqueFunction = @Ts;
torqueName = 'Ts';
sys.addVariableTorque(bodyNumber, torqueFunction, torqueName);
```

Figure 13: Portion of a driver file showing how to add a variable torque into a multibody system.

```
function [ torque ] = Ts( sys, time )
%Function that will be used to compute the torque to apply to the flyball
%governor.
C = 12500;

% Extract the current y-position of the collar
yLoc = sys.myBodies{5}.myR(2);
L0 = 0.15;
currentL = 0.2 - yLoc;
deltaL = currentL - L0;
torqueY = C*deltaL;
torque = [0; torqueY; 0];

end
```

Figure 14: Example of a function used to define the output of a variable torque component. As seen in this function, the torque depends on the system state, specifically the y-location of body 5, but does not depend on time. However, both the multibody system and the current time must be provided as inputs to the function.

5.3. Translational-Spring-Damper-Actuators

A translational-spring-damper-actuator (TSDA) is attached between two points on two different bodies and applies a force based on the distance between these points, the rate of change of this distance, and the current time in the system (Eq. 1-6).

$$\text{Vector between connection points of TSDA:} \quad \boldsymbol{d_{ij}} = \boldsymbol{r_j} + \boldsymbol{A_j}\bar{\boldsymbol{s}}_{\boldsymbol{j}}^{\boldsymbol{Q}} - \boldsymbol{r_i} - \boldsymbol{A_i}\bar{\boldsymbol{s}}_{\boldsymbol{i}}^{\boldsymbol{P}} \tag{1}$$

$$\text{Length of TSDA:} \quad l = \sqrt{\boldsymbol{d}_{\boldsymbol{ij}}^{T}\boldsymbol{d_{ij}}} \tag{2}$$

$$\text{Rate of change of TSDA length:} \quad \dot{l} = \left(\frac{\boldsymbol{d_{ij}}}{l}\right)^{T}\left(\dot{\boldsymbol{r}}_{\boldsymbol{J}} + \dot{\boldsymbol{A}}_{\boldsymbol{J}}\bar{\boldsymbol{s}}_{\boldsymbol{j}}^{\boldsymbol{Q}} - \dot{\boldsymbol{r}}_{\boldsymbol{\iota}} - \dot{\boldsymbol{A}}_{\boldsymbol{\iota}}\bar{\boldsymbol{s}}_{\boldsymbol{i}}^{\boldsymbol{P}}\right) \tag{3}$$

$$\text{Force magnitude:} \quad f = k(l - l_0) + c\dot{l} + h(l, \dot{l}, t), \tag{4}$$

where l_0 = resting length of spring
k = spring stiffness
c = damping coefficient
$h(l, \dot{l}, t)$ = actuator function

$$\text{Force direction:} \quad \boldsymbol{e_{ij}} = \frac{\boldsymbol{d_{ij}}}{l} \tag{5}$$

$$\text{Total Force of TSDA:} \quad F = f * \boldsymbol{e_{ij}} \tag{6}$$

When adding a TSDA to the multibody system, the user must define all attributes of a TSDA even if they intend to only use a single attribute of this force component. For example, a user can add a spring into the system using a TSDA by setting the damping coefficient and actuator function equal to zero (Fig 15).

```
%% Add TSDA to the system
s1.bodyI = 2;
s1.bodyJ = 5;
s1.sBarIP = [0 0 0]';
s1.sBarJQ = [0.0 0 0]';
s1.stiffness = 1000;
s1.restingLength = 0.15;
s1.dampingCoefficient = 30;
s1.actuatorFunction = @(lij, lijDot, t)0;
s1.name = 'Spring-Damper Element b/w body 2 and 5';
sys.addTSDA(s1);
```

Figure 15: Portion of a driver file showing how to add a TSDA force element into a multibody system.

6. Analysis of Mechanisms

In simEngine3D there are currently three options for performing an analysis of a mechanism: kinematics, inverse dynamics, and dynamics analyses. Prior to performing any of these analyses, the user must provide initial conditions for the mechanism and perform an assembly analysis to ensure the initial conditions are consistent with the user-specified constraints. Each of these steps is discussed in further detail in the following sections.

6.1. Prescribing Initial Positions and Orientations

Prior to performing analysis of the multibody system, the user must specify the initial position and orientation of each body in the global reference frame. In simEngine3D, the orientation of a body is defined using Euler parameters (also called quaternions). As it is sometimes difficult to provide the initial Euler parameters of a body, *simEngine3DUtilities* contains a function (*A2p()*) that converts an orientation matrix *(A*) to Euler parameters (*p*). This allows the user to compute the initial orientation matrix of a body and then convert this orientation matrix into the initial Euler parameters. The initial pose of the body is then specified by calling the function *setInitialPose()* through the instance of the *multibodySystem* class (Fig 16).

```
%% Set initial conditions of each body
% Initial position
r1Initial = zeros(3,1);
r2Initial = [0; sqrt(2); -sqrt(2)];
rInitial = [r1Initial; r2Initial];

% Initial orientation
p1 = [1 0 0 0]';

s2 = sqrt(2)/2;
A2 = [0 0 1;
    s2 s2 0;
    -s2 s2 0];
p2 = simEngine3DUtilities.A2p(A2);

pInitial = [p1; p2];

assemblyAnalysisFlag = 1;
sys.setInitialPose( rInitial, pInitial, assemblyAnalysisFlag);
```

Figure 16: Portion of a driver file showing how to set the initial pose of a body using the function setInitialPose().

6.2. Assembly Analysis

When setting the initial pose of the system, the user also has the option to perform an assembly analysis. An assembly analysis is used to check if the initial pose is consistent with the constraints that have been imposed on the system. If the initial pose is not consistent, the pose is adjusted to ensure consistency. This is done by solving an optimization problem that minimizes an objective function (Eq. 7-9) that aims to find a consistent initial pose that is as close as possible to the user-specified initial pose.

$$\min_{\boldsymbol{q}} \Psi(\boldsymbol{q}, t_0, \mathrm{r}), \tag{7}$$

$$given\ that: \Psi(\boldsymbol{q}, t_0, \mathrm{r}) = (\boldsymbol{q} - \boldsymbol{q}^0)^T(\boldsymbol{q} - \boldsymbol{q}^0) + \boldsymbol{r}\boldsymbol{\Phi}^{\boldsymbol{F}^T}(\boldsymbol{q}, t_0)\boldsymbol{\Phi}^{\boldsymbol{F}}(\boldsymbol{q}, t_0) \tag{8}$$

$$and \quad \Psi_{\boldsymbol{q}} = 2(\boldsymbol{q} - \boldsymbol{q}^0)^T + 2r\boldsymbol{\Phi}^{\boldsymbol{F}^T}(\boldsymbol{q}, t_0)\boldsymbol{\Phi}_{\boldsymbol{q}}(\boldsymbol{q}, t_0), \tag{9}$$

where, $\boldsymbol{q}$ = vector containing position and orientation of all bodies

$\boldsymbol{q}^0$ = guess for initial pose

$\boldsymbol{\Phi}(\boldsymbol{q}, t_0)$ = vector of all constraints

$\boldsymbol{\Phi}_{\boldsymbol{q}}(\boldsymbol{q}, t_0)$ = Jacobian of constraint matrix

t_0 = initial time

r = positive weighting constant

Within simEngine3D, this optimization problem is solved using the MATLAB built-in function *fminunc(),* which solves an unconstrained minimization of the provided objective function. While not required to run a simulation, performing an assembly analysis is highly recommended. The user can perform an assembly analysis through the use of a flag specified in the function call for *setInitialPose()*.

6.3. Prescribing Initial Velocities

Prior to performing a dynamics analysis (but not for a kinematics or inverse dynamics analysis), the user must prescribe the initial velocities of each body in the system. Prescribing the initial velocities of a system can be broken into three cases and simEngine3D has methods to account for each of these cases.

Case 1: Initial velocity known for all bodies.

In this case, the user can specify the initial velocities by calling the function *updateSystemState()* and providing the velocities as an input to this function (Fig 17).

```
t = 0;
sys.updateSystemState( [], rDotInitial, [], [], pDotInitial, [], t);
```

Figure 17: Portion of a driver file showing how to the set the initial velocities of all bodies, if all are known.

Case 2: Initial velocity not known for any of the bodies.

In this case, the user can compute and set the initial velocity for all the bodies in the system using the function *computeAndSetIntialVelocities()*. When calling this function, the user provides an empty matrix for each of the function inputs, which informs the function that none of the initial velocities are known (Fig 18). Initial velocities that satisfy the velocity-level constraints are then computed for all bodies in the system (Eq. 10).

$$\begin{bmatrix} \Phi_r & \Phi_p \\ 0_{nB\ x\ 3nB} & P \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{r}} \\ \dot{\boldsymbol{p}} \end{bmatrix} = \begin{bmatrix} -\Phi_t \\ 0_{nB\ x\ 1} \end{bmatrix}, \tag{10}$$

$$where, \quad \Phi_r = \text{partial derivative of the constraint vector with respect to the body positions}$$

$$\Phi_p = \text{partial derivative of the constraint vector with respect to the body orientations}$$

$$P = \begin{bmatrix} \boldsymbol{p}_1^T & 0_{1\ x\ 4} & \cdots & 0_{1\ x\ 4} \\ 0_{1\ x\ 4} & \boldsymbol{p}_2^T & \cdots & 0_{1\ x\ 4} \\ \cdots & \cdots & \cdots & \cdots \\ 0_{1\ x\ 4} & 0_{1\ x\ 4} & 0_{1\ x\ 4} & \boldsymbol{p}_{nB}^T \end{bmatrix}$$

$$\begin{bmatrix} \dot{\boldsymbol{r}} \\ \dot{\boldsymbol{p}} \end{bmatrix} = \text{ vector of initial velocities}$$

$$\Phi_t = \text{time derivative of constraint matrix}$$

$$nB = \text{number of bodies in system (not including the ground)}$$

```
sys.computeAndSetInitialVelocities([], [], []);
```

Figure 18: Portion of a driver file showing how to compute and set the initial velocities of all bodies if all are unknown.

Case 3: Initial velocity known for a subset of the bodies.

This case is similar to the previous case, in that, the user can compute and set the initial velocity for the rest of the bodies in the system using the function *computeAndSetIntialVelocities()*. However, in this case the user must provide the body numbers with known velocities and the initial velocities of these bodies as inputs to this function (Fig 19). This function then computes the initial velocity for the rest of the bodies, such that these velocities satisfy the velocity-level constraints (Eq. 11). In equation 11, the subscript *known* indicates that this parameter is computed for bodies with known initial velocities and the subscript

unknown indicates that this parameter is computed for bodies with unknown initial velocities. For example, $\Phi_{r_{unknown}}$ is the partial derivative of the constraint vector with respect to the position of the bodies that have unknown initial velocities and $\Phi_{r_{known}}$ is the partial derivative of the constraint vector with respect to the position of the bodies that have known initial velocities.

$$\begin{bmatrix} \Phi_{r_{unknown}} & \Phi_{p_{unknown}} \\ 0_{nB_{unknown}\ x\ 3nB_{unknown}} & P_{unknown} \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{r}}_{unknown} \\ \dot{\boldsymbol{p}}_{unknown} \end{bmatrix} = \begin{bmatrix} -\Phi_t - \Phi_{r_{known}} \dot{\boldsymbol{r}}_{known} - \Phi_{p_{known}} \dot{\boldsymbol{p}}_{known} \\ 0_{nB_{unknown}\ x\ 1} \end{bmatrix}, (11)$$

```
% Initial velocities. Initial velocity known for crank.
known = 2;
knownInitialRDot = [0; 0; 0];
knownInitialOmegaBar = [0; 11.0174; 0];
knownInitialPDot = simEngine3DUtilities.omegaBar2pDot(sys, known, knownInitialOmegaBar);
sys.computeAndSetInitialVelocities(known, knownInitialRDot, knownInitialPDot);
```

Figure 19: Portion of a driver file showing how to compute and set the initial velocities of a system if some of the initial velocities are known.

6.4. Kinematics Analysis

A kinematics analysis is used to compute the position, velocity, and acceleration of each body in a multibody system throughout time. Prior to beginning a kinematics analysis, the user must begin with a system that has zero degrees of freedom (i.e. a fully constrained system). This means that the number of user-specified constraints must be equal to six times the number of bodies in the system (not including the ground). As previously mentioned, simEngine3D uses Euler parameters (or quaternions) to define the body orientation, which actually gives each body seven degrees of freedom. However, simEngine3D automatically includes one Euler parameter normalization constraint per body, which requires the user to only constrain the other six degrees of freedom to perform a kinematics analysis.

A kinematics analysis is composed of three steps: position analysis, velocity analysis, and acceleration analysis. For the first time step of a kinematics analysis, simEngine3D only performs a velocity and an acceleration analysis because the initial position is specified from either the assembly analysis or the user-specified initial pose. In subsequent time steps, all three steps are performed. During the position analysis, simEngine3D solves a nonlinear system of equations to find the current position that satisfies the position-level constraints (Eq. 12). This nonlinear system of equations is solved using the Newton-Raphson method. During the velocity and acceleration analyses, simEngine3D solves a linear system of equations to find the current set of velocities and accelerations that satisfy the velocity and acceleration-level constraints, respectively (Eq. 13 and 14). For a more in-depth discussion of how to compute each of the terms needed to solve these systems of equations, please see chapters 3 and 10 in *Computer Aided Kinematics and Dynamics of Mechanical Systems*.

$$\text{Position Analysis: } \Phi^F(\boldsymbol{q},t) = \begin{bmatrix} \Phi^K(\boldsymbol{q}) \\ \Phi^D(\boldsymbol{q},t) \\ \Phi^P(\boldsymbol{q}) \end{bmatrix} = 0, \tag{12}$$

$$\text{where, } \Phi^K(\boldsymbol{q}) = \text{kinematic constraints}$$
$$\Phi^D(\boldsymbol{q},t) = \text{dynamic constraints}$$
$$\Phi^P(\boldsymbol{q}) = \text{Euler parameter normalization constraints}$$

$$\text{Velocity Analysis: } \Phi^F{}_q\dot{\boldsymbol{q}} = -\Phi_t \tag{13}$$

$$\text{Acceleration Analysis: } \Phi^F_q\ddot{\boldsymbol{q}} = -\left(\Phi^F_q\dot{\boldsymbol{q}}\right)_q\dot{\boldsymbol{q}} - 2\Phi^F_{qt}\dot{\boldsymbol{q}} - \Phi^F_{tt} \tag{14}$$

6.5. Inverse Dynamics Analysis

An inverse dynamics analysis is used to compute the kinematics of each body in a multibody system and the reaction forces due to the constraints throughout a simulation. As with a kinematics analysis, prior to performing an inverse dynamics analysis the user must begin with a fully constrained multibody system. An inverse dynamics analysis proceeds in a similar fashion to a kinematics analysis in that this analysis performs a position analysis, a velocity analysis, and an acceleration analysis. However, following the acceleration analysis, the inverse dynamics analysis performs an additional step and computes the reaction forces and torques for each of the constraints. This allows the user to determine the forces and torques applied to the center of mass of each body due to each of the constraints in the system.

6.6. Dynamics Analysis

A dynamics analysis is used to simultaneously compute the kinematics of each body and the reaction forces due to the constraints in a multibody system throughout a simulation. However, in contrast to a kinematics and an inverse dynamics analysis, a dynamics analysis can be performed when the multibody system has free degrees of freedom. This allows the user to study the behavior of a given mechanism when subjected to externally applied forces and torques.

Performing a dynamics analysis requires simEngine3D to solve the Newton-Euler form of the equations of motion at each time step of the simulation (Eq. 15). For a more thorough description of the equations of motion, and how to compute each of the terms seen in the equations, please see chapter 11 in *Computer Aided Kinematics and Dynamics of Mechanical Systems.*

$$\begin{bmatrix} M & 0_{3nB\ x\ 4nB} & 0_{3nB\ x\ nB} & {\Phi_r}^T \\ 0_{4nB\ x\ 3nB} & J^P & P^T & {\Phi_p}^T \\ 0_{nB\ x\ 3nB} & P & 0_{nB\ x\ nB} & 0_{nB\ x\ nC} \\ \Phi_r & \Phi_p & 0_{nC\ x\ nB} & 0_{nC\ x\ nC} \end{bmatrix} \begin{bmatrix} \ddot{r} \\ \ddot{p} \\ \lambda^p \\ \lambda \end{bmatrix} = \begin{bmatrix} F \\ \hat{\tau} \\ {\gamma^P} \\ \hat{\gamma} \end{bmatrix}, \tag{15}$$

where, M = the mass matrix of the multibody system

Φ_r = the partial derivative of the constraint vector with respect to the body positions

Φ_p = the partial derivative of the constraint vector with respect to the body orientations

P = the Euler parameter matrix

J^P = the polar moment of inertia for the Euler parameter form of the equations of motion

$\ddot{r}$ = linear acceleration of each body

$\ddot{p}$ = second time derivative of the Euler parameters for each body

λ^p = Lagrange multipliers associated with the Euler parameter normalization constraints

λ = Lagrange multipliers associated with the kinematic and driving constraints

F = Externally applied forces

$\hat{\tau}$ = Externally applied torques plus additional torques caused by the Coriolis effect

γ^P = Right hand side of acceleration analysis for Euler parameter normilzation constraints

$\hat{\gamma}$ = Right hand side of acceleration analysis for kinematic and driving constraints

nB = number of bodies (not including the ground)

nC = number of kinematic and driving constraints

Solving the equations of motion throughout time requires the use of a numerical integration method. Within simEngine3D, an implicit numerical integration method (specifically the Backward Differentiation Formula (BDF) method) is used to compute the kinematics of the system at each time step. To date, order 1 and order 2 BDF methods have been implemented into simEngine3D and the desired order of the BDF method can be set when performing the dynamics analysis.

In addition to requiring use of a numerical integration method, solving the equations of motion at each time step requires simEngine3D to solve a nonlinear system of equations. To solve this system of equations, simEngine3D offers three different variations of the Newton-Raphson method: full Newton-Raphson, modified Newton-Raphson, and quasi Newton-Raphson. These methods differ in the form of the Jacobian of the left hand side matrix of the equations of motion used and in the number of times that the Jacobian is computed. Full Newton-Raphson uses the full Jacobian and re-computes the Jacobian at each iteration. Modified Newton-Raphson uses the full Jacobian, but only computes the Jacobian at the first iteration. Quasi Newton-Raphson uses a simplified Jacobian and only computes the Jacobian at the first iteration. Similar to choosing the order of the BDF method, the desired variation of the Newton-Raphson method can be specified when performing a dynamics analysis in simEngine3D.

6.7 Performing Kinematics, Inverse Dynamics, and Dynamics Analyses

Prior to performing a kinematics, dynamics, or inverse dynamics analysis the user must define all bodies, constraints, and external forces and torques in the system. These analyses can then be performed by calling the desired methods within the *multibodySystem* class (Fig 20-22).

```
%% Perform kinematics analysis
timeStart = 0;
timeEnd = 10;
timeStep = 10^-2;
displayFlag = 1;
sys.kinematicsAnalysis(timeStart, timeEnd, timeStep, displayFlag);
```

Figure 20: Example of a kinematics analysis being called in a driver file.

```
%% Perform inverse dynamics analysis
timeStart = 0;
timeEnd = 10;
timeStep = 10^-2;
displayFlag = 1;
sys.inverseDynamicsAnalysis(timeStart, timeEnd, timeStep, displayFlag);
```

Figure 21: Example of an inverse dynamics analysis being called in a driver file.

```
%% Perform dynamics analysis
timeStart = 0;
timeEnd = 10;
timeStep = 10^-2;
order = 2;
displayFlag = 1;
velocityConstraintViolationFlag = 0;
method = 'quasiNewton';
sys.dynamicsAnalysis(timeStart, timeEnd,timeStep, order, method, ...
    displayFlag, velocityConstraintViolationFlag);
```

Figure 22: Example of a dynamics analysis being called in a driver file.

7. Validation Efforts

Validation of simEngine3D has been performed primarily through comparison of simulation results to examples from *Computer Aided Kinematics and Dynamics of Mechanical Systems* and through comparison to benchmark problems (taken from: http://lim.ii.udc.es/mbsbenchmark/). These efforts have allowed for validation of the joints and of the external force and torque components currently implemented in simEngine3D. The following is a discussion of each of the validation problems, including the purpose of each problem, the results of the simulation, and comparison to previously validated results.

7.1. Validation of Model Components

7.1.1. Joints

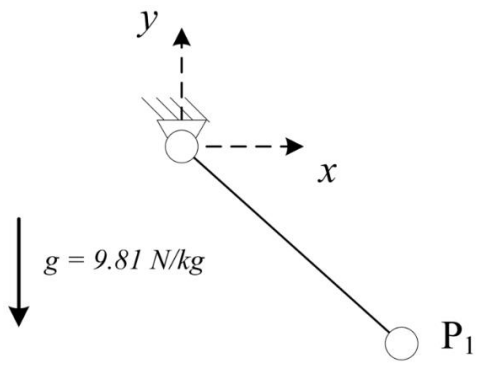
Simple Pendulum with Revolute Joint

Source of example:
http://lim.ii.udc.es/mbsbenchmark/dist/A01/A01_specification.xml.

Purpose of validation:
This problem served to validate the revolute joint implemented in simEngine3D. Additionally, this problem also allowed for comparison to a common benchmark problem.

Brief overview of validation problem:
The problem is a simple pendulum that contains a point mass connected to the ground via a revolute joint.

Image of mechanism:



Simulation Parameters:
Start time = 0
End time = 20
Step-size = 10^{-2}
BDF order = 2
Variation of Newton-Raphson = quasi Newton-Raphson

Comparison of results to previously validated results or to analytical results:

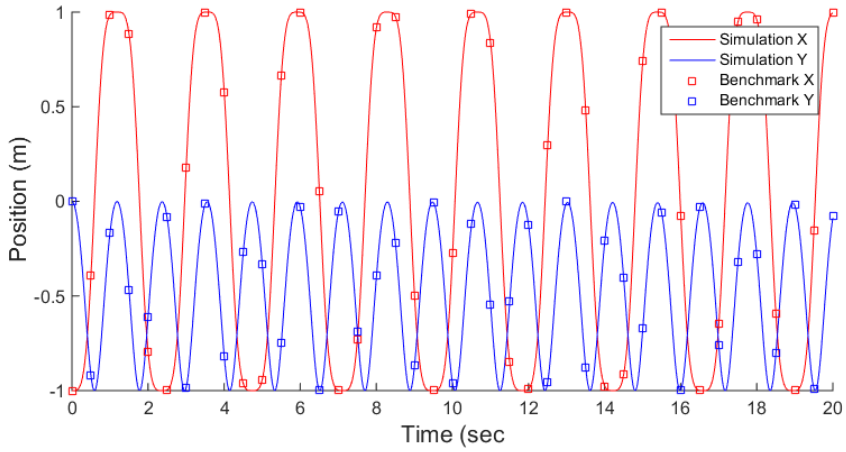


Figure 23: Comparison of simulated simple pendulum results to previously validated results for a simple pendulum.

Simple Pendulum with Cylindrical Joint

Source of example:

http://lim.ii.udc.es/mbsbenchmark/dist/A01/A01_specification.xml.

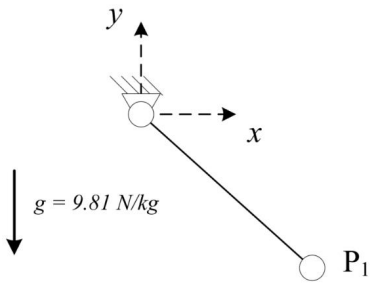
Purpose of validation:

The purpose of this example was to validate the implementation of the cylindrical joint.

Brief overview of validation problem:

This problem consists of a simple pendulum with a point mass connected to the ground a cyliandrical joint. To validate the rotation of the cylindrical joint, the x and y position of the point mass throughout the simulation was compared to the position for a simple pendulum with a revolute joint. To validate the translation of the cylindrical joint, an external force of 1 N was applied to the pendulum mass in the z-direction and it was confirmed that this 1 N load induced a 1 m/s^2 acceleration to the mass. This confirmed that the cylindrical was translating as it should.

Image of mechanism:



Simulation Parameters:

Start time = 0

End time = 20

Step-size = 10^{-2}

BDF order = 2

Variation of Newton-Raphson = quasi Newton-Raphson

Comparison of results to previously validated results or to analytical results:

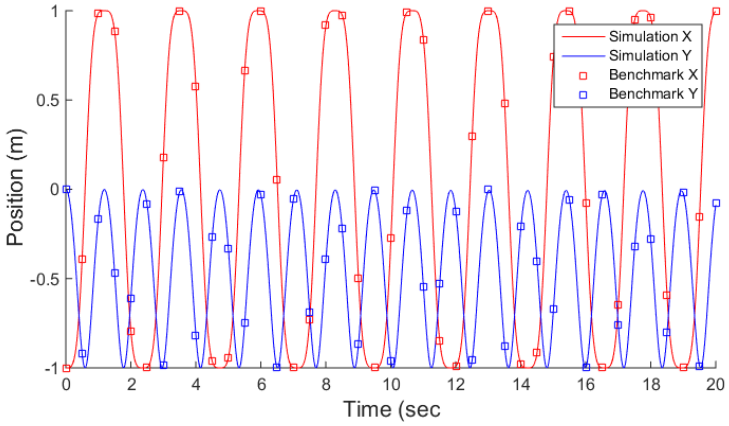


Figure 24: Comparison of results for a simulated simple pendulum with a cylindrical joint to previously validated results.

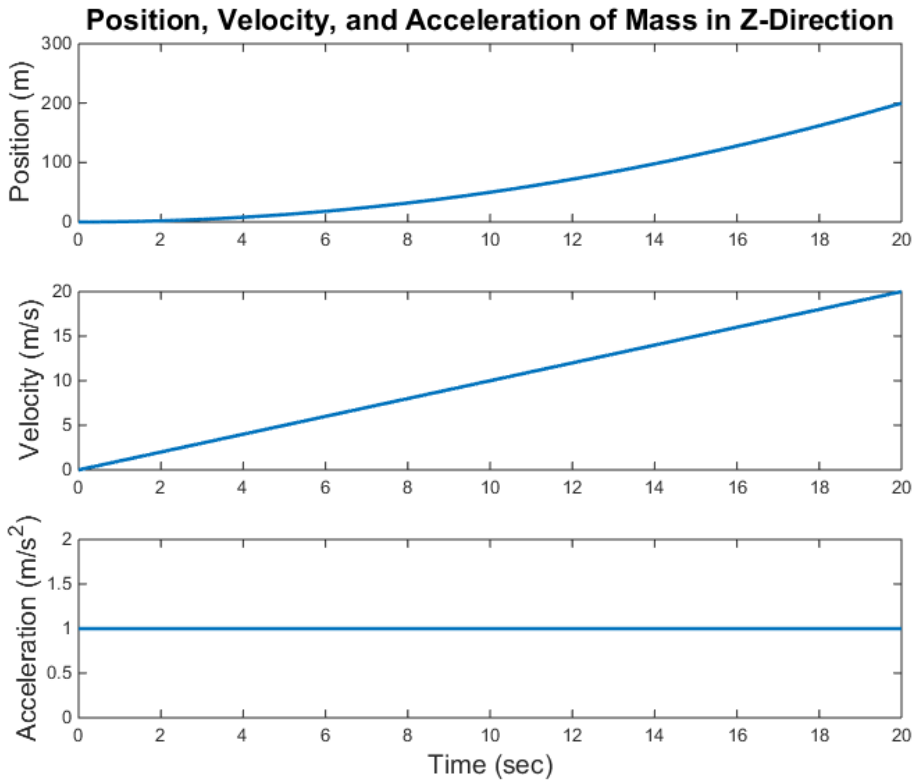


Figure 25: Kinematics of the mass of the simple pendulum in the z-direction. This behavior is caused by a 1N point load being applied to the mass in the z-direction.

Block on Inclined Translation Joint

Source of example:

No specific source for this example. This is just a simple example with an analytical solution to validate the translational joint.

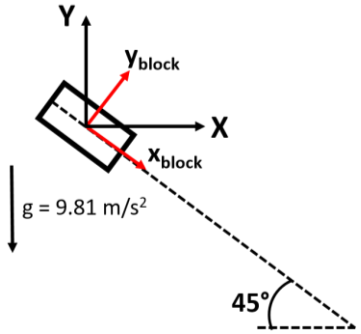
Purpose of validation:

The purpose of this example was to validate the implementation of the translational joint in simEngine3D.

Brief overview of validation problem:

This problem is a block on a 45° inclined plane. The inclined plane is modeled as a translational joint passing through the block's center of mass. Based on the analytical solution, we expect the acceleration of the block in the y-direction to be -4.905 m/s^2 and the acceleration in the x-direction to +4.905 m/s^2.

Image of mechanism:



Simulation Parameters:

Start time = 0

End time = 5

Step-size = 10^{-2}

BDF order = 2

Variation of Newton-Raphson = quasi Newton-Raphson

Comparison of results to previously validated results or to analytical results:

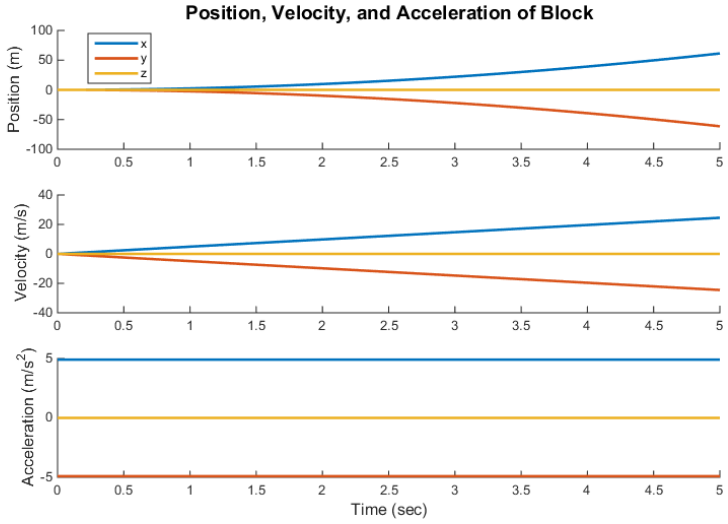


Figure 26: Kinematics produced by simulating a block sliding down an inclined plane.

Slider-Crank Mechanism

Source of example:

Chapter 12.2 of *Computer Aided Kinematics and Dynamics of Mechanical Systems*

Purpose of validation:

After validation of the revolute and translational joints, this purpose of simulating this mechanism was to validate the spherical and revolute-cylindrical joints implemented in simEngine3D.

Brief overview of validation problem:

This problem consists of two different simulations of a spatial slider-crank mechanism. The first is a dynamics analysis of the mechanism with a constant angular velocity of 2π rad/sec applied to the crank. For this simulation, the kinematics of the slider and the driving torque required to maintain this constant angular velocity were plotted and compared to the textbook results. The second is a dynamics analysis with an initial angular velocity of 6 rad/sec applied to the crank. For this analysis, the kinematics of the slider were plotted and compared to the textbook results. For both simulations, the length of the connecting rod was set to 0.3 m.

Image of mechanism:

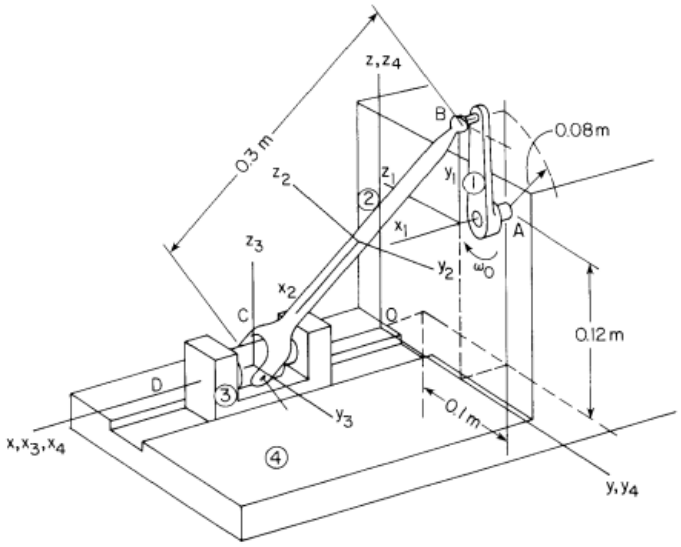


Figure 10.2.1 Spatial slider–crank.

Simulation Parameters:

NOTE: Same for both simulations

Start time = 0

End time = 1

Step-size = 10^{-3}

BDF order = 2

Variation of Newton-Raphson = quasi Newton-Raphson

Comparison of results to previously validated results or to analytical results:

Dynamics analysis with constant angular velocity applied to crank:

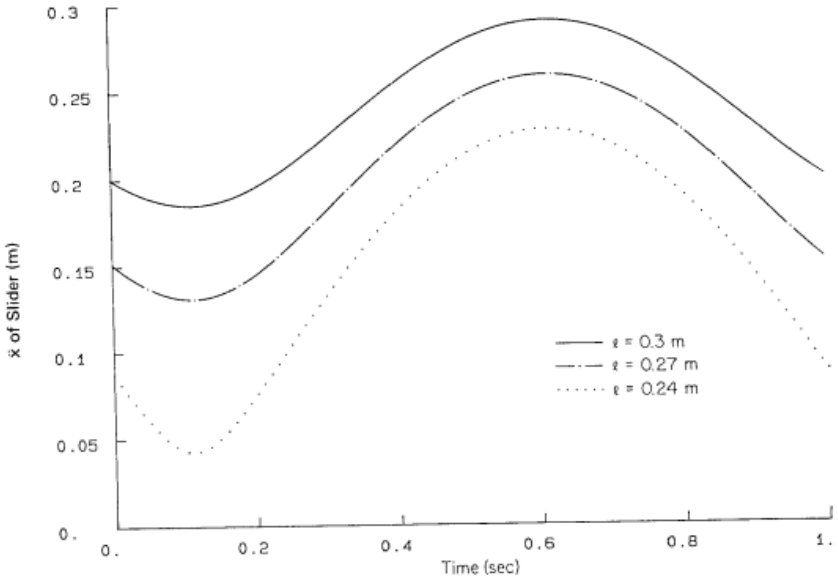


Figure 10.2.3 x of slider versus time.

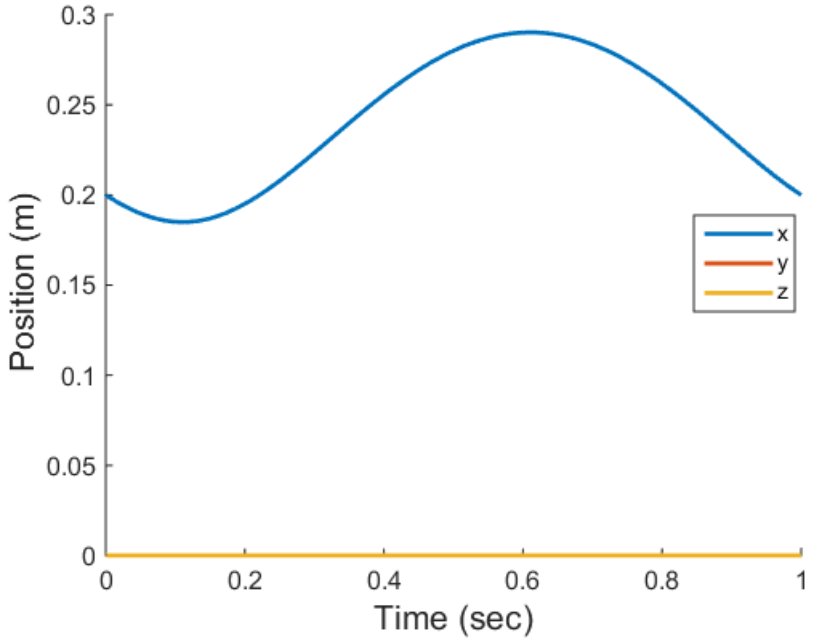


Figure 27: (Top) Textbook results of the x-position of the slider of a spatial slider-crank mechanism for a connecting rod length (l) of 0.3 m. (Bottom) Simulated results of the same mechanism.

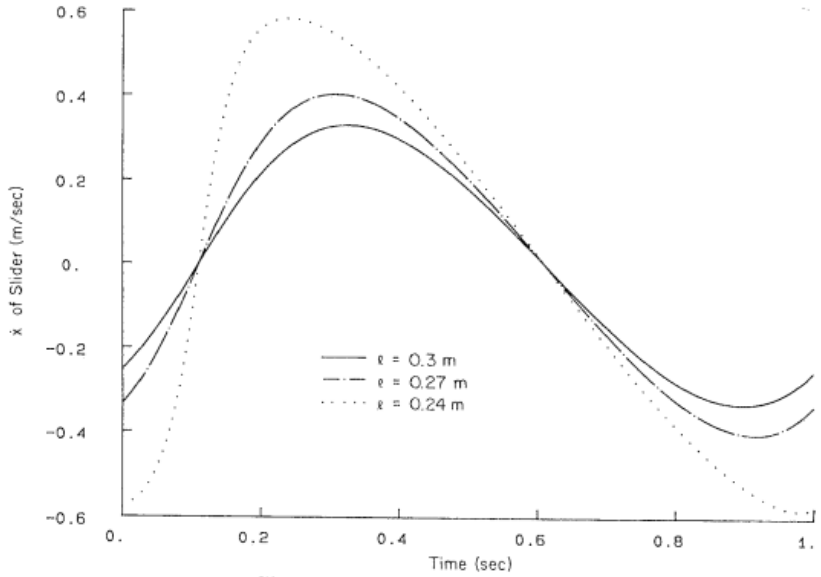


Figure 10.2.4 $\dot{x}$ of slider versus time.

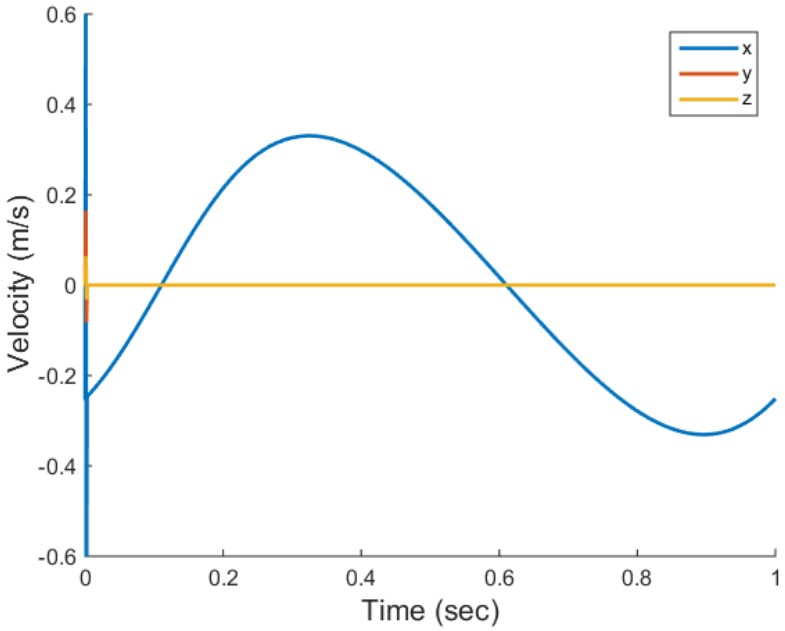


Figure 28: (Top) Textbook results of the velocity of the slider of a spatial slider-crank mechanism for a connecting rod length (l) of 0.3 m. (Bottom) Simulated results of the same mechanism.

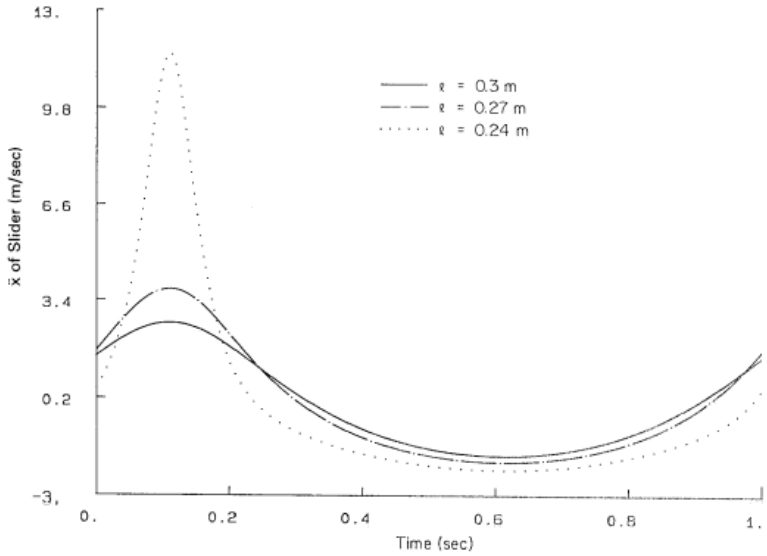


Figure 10.2.5 $\ddot{x}$ of slider versus time.

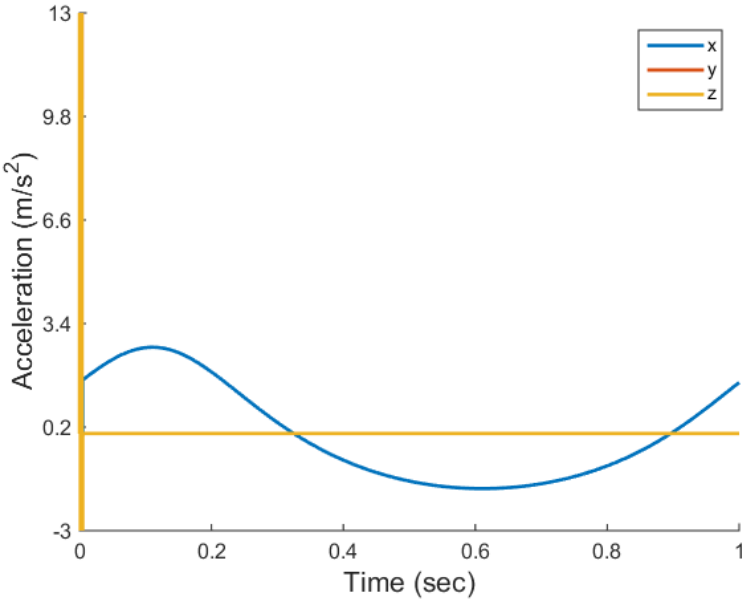


Figure 29: (Top) Textbook results of the acceleration of the slider of a spatial slider-crank mechanism for a connecting rod length (l) of 0.3 m. (Bottom) Simulated results of the same mechanism.

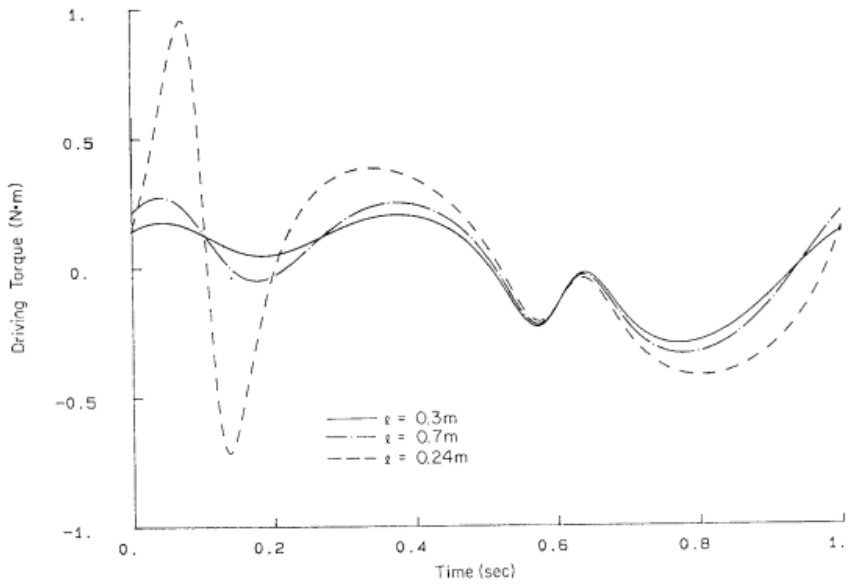


Figure 12.2.1 Driving torque for a spatial slider–crank mechanism.

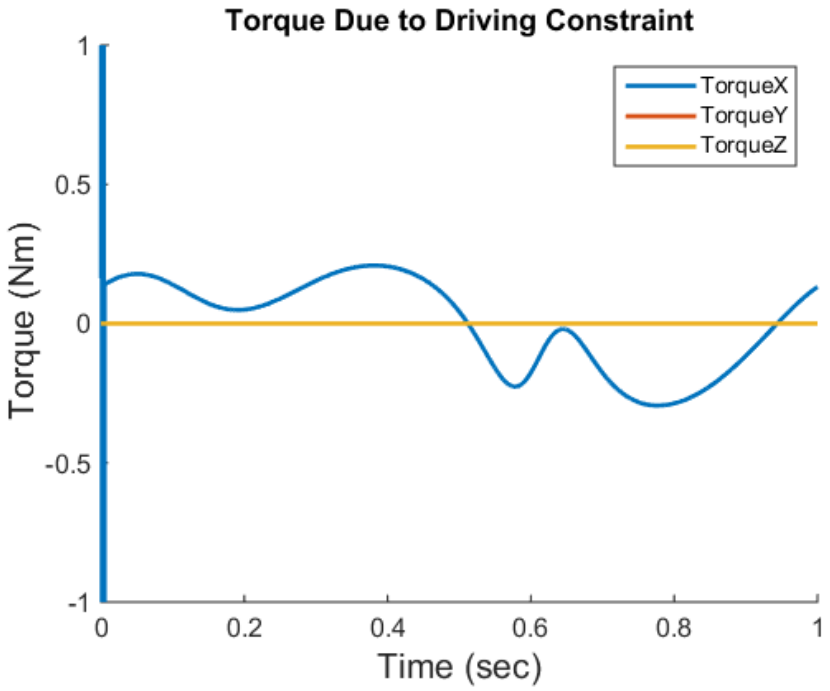


Figure 30: (Top) Textbook results for the driving torque required to maintain a constant crank angular velocity of 2π rad/sec for a connecting rod length (l) of 0.3 m. (Bottom) Simulated results of the same mechanism.

Dynamics analysis with initial angular velocity of 6 rad/sec applied to crank:

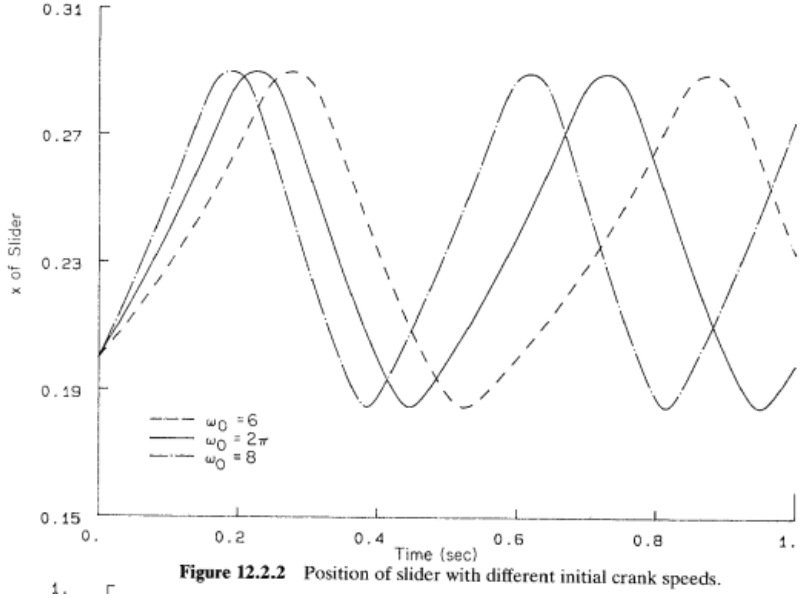


Figure 12.2.2 Position of slider with different initial crank speeds.

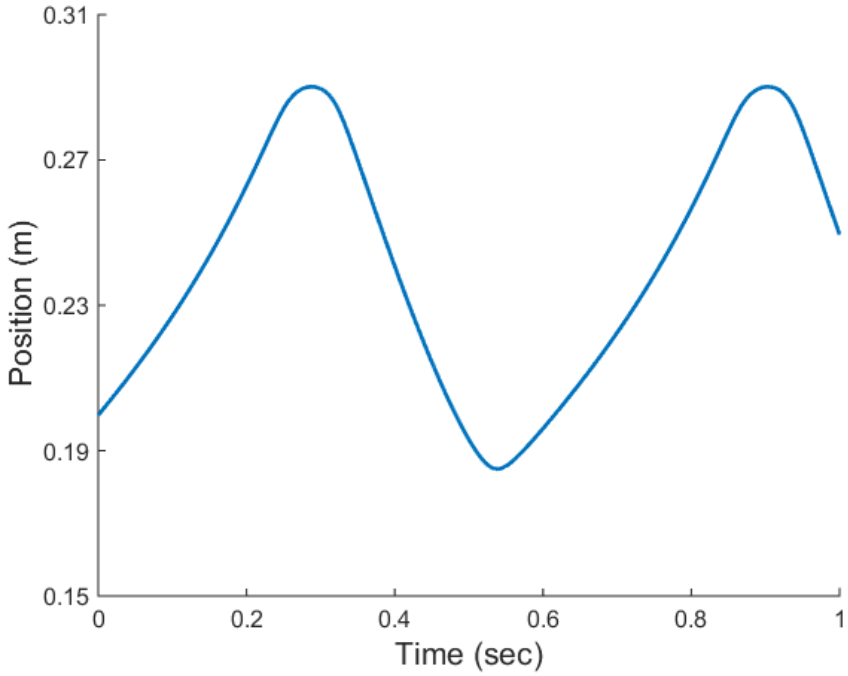


Figure 31: (Top) Textbook results for the x-position of the slider in a spatial slider-crank mechanism with an initial crank angular velocity of 6 rad/sec. (Bottom) Simulated results of the same mechanism.

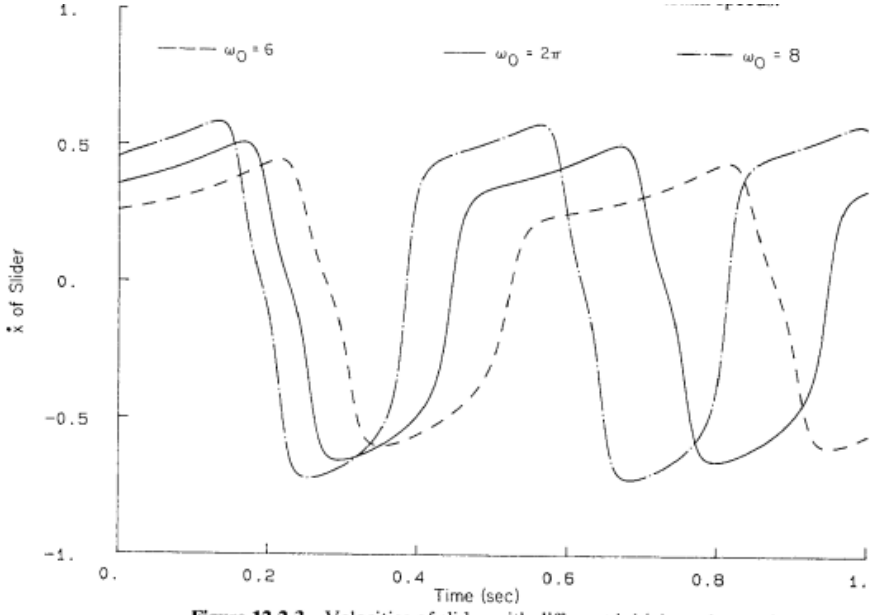


Figure 12.2.3 Velocities of slider with different initial crank speeds.

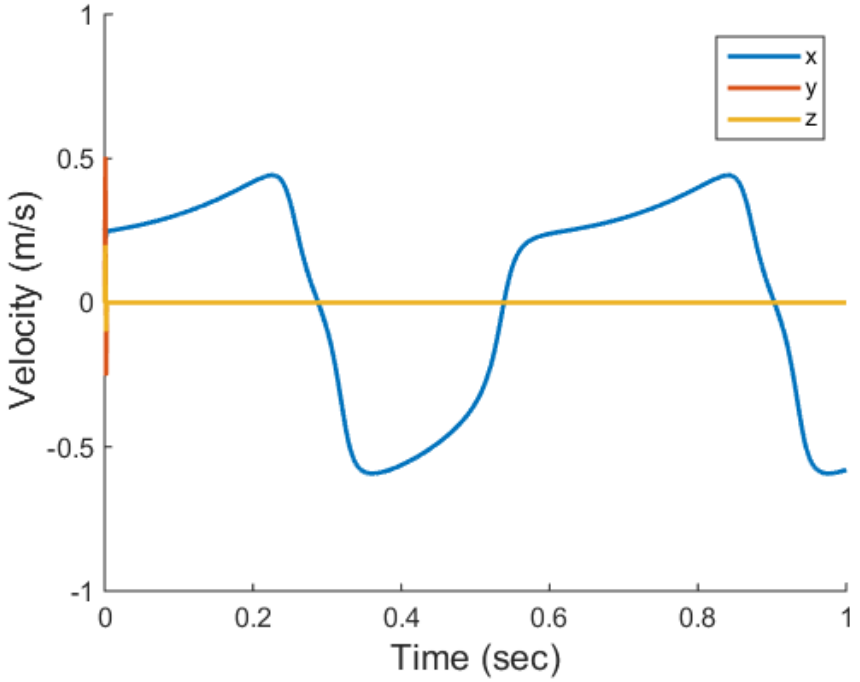


Figure 32: (Top) Textbook results for the velocity of the slider in a spatial slider-crank mechanism with an initial crank angular velocity of 6 rad/sec. (Bottom) Simulated results of the same mechanism.

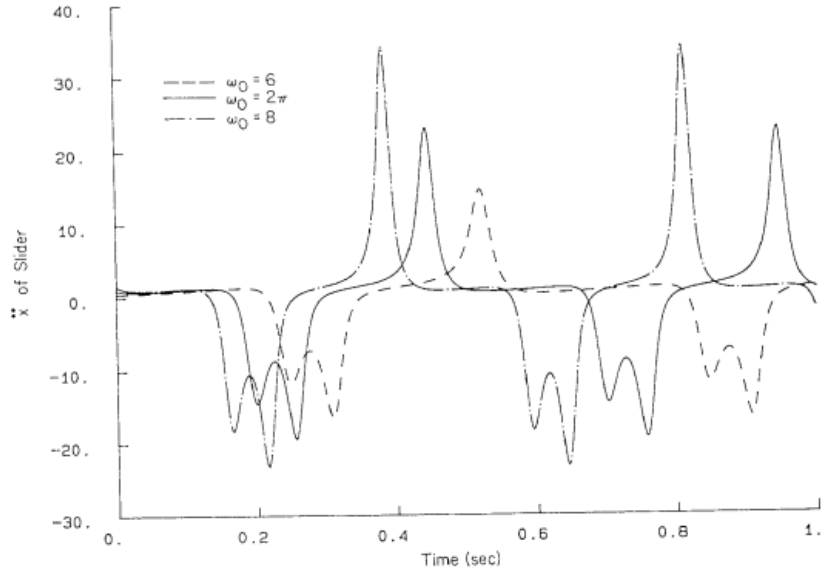


Figure 12.2.4 Accelerations of slider with different initial crank speeds.

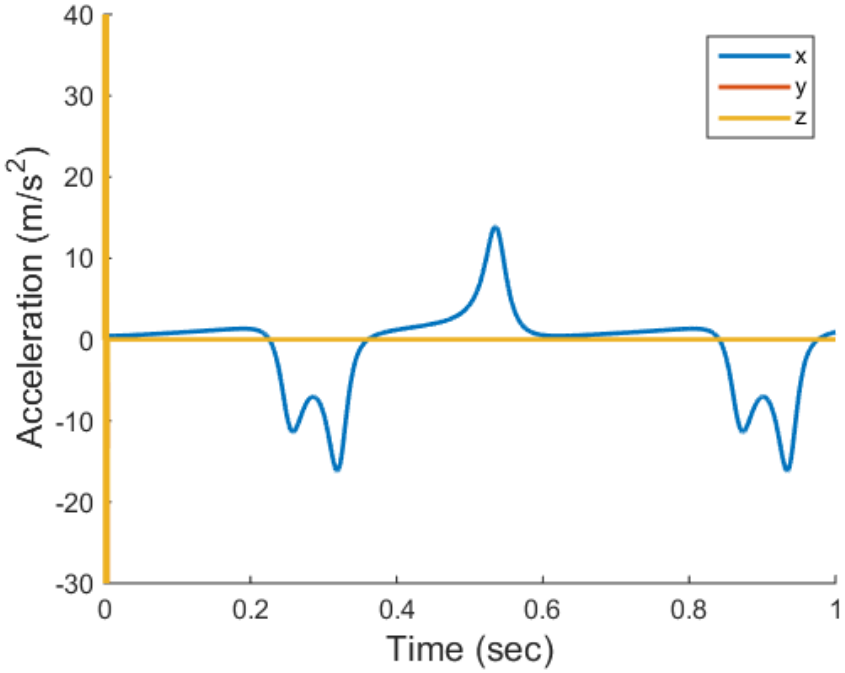


Figure 33:: (Top) Textbook results for the acceleration of the slider in a spatial slider-crank mechanism with an initial crank angular velocity of 6 rad/sec. (Bottom) Simulated results of the same mechanism.

Four Bar Mechanism

Source of example:

Chapter 12.3 of *Computer Aided Kinematics and Dynamics of Mechanical Systems*

Purpose of validation:

Prior to simulating this mechanism, the spherical and revolute joints were validated. Therefore, the purpose of simulating this four-bar mechanism was to validate the universal joint and the ability to provide a constant torque to a body.

Brief overview of validation problem:

This problem consisted of two different simulations of a four-bar mechanism. The first simulation was a dynamics analysis with a constant angular velocity of π applied to the first link (the wheel). For validation, the position of point B on the first link, the position and velocity of the rocker, the required driving torque applied to link 1 to maintain this constant angular velocity, and the reaction force at point A on the first link were compared to the textbook results. The second simulation was a dynamics analysis with a constant counterclockwise torque of 10 Nm applied to the first link. For validation, the position and velocity of the rocker were compared to the textbook results.

Image of mechanism:

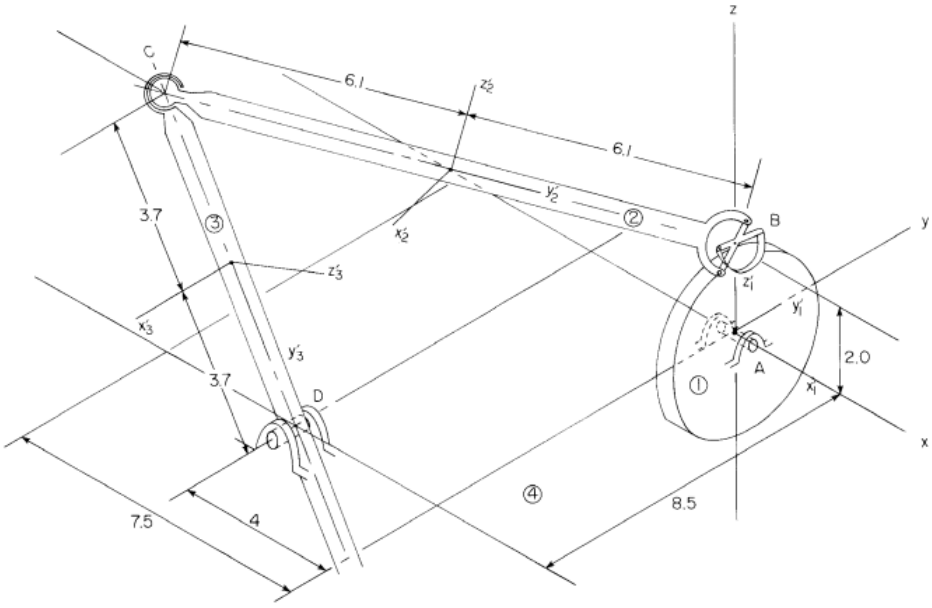


Figure 10.3.1 Spatial four-bar mechanism, model 1.

Simulation Parameters:

Dynamics analysis with constant angular velocity:

Start time = 0

End time = 2.5

Step-size = 10^{-2}

BDF order = 2

Variation of Newton-Raphson = quasi Newton-Raphson

Dynamics analysis with constant counterclockwise torque:
Start time = 0
End time = 2
Step-size = 10^{-2}
BDF order = 2
Variation of Newton-Raphson = quasi Newton-Raphson

Comparison of results to previously validated results or to analytical results:
Dynamics analysis with constant angular velocity:

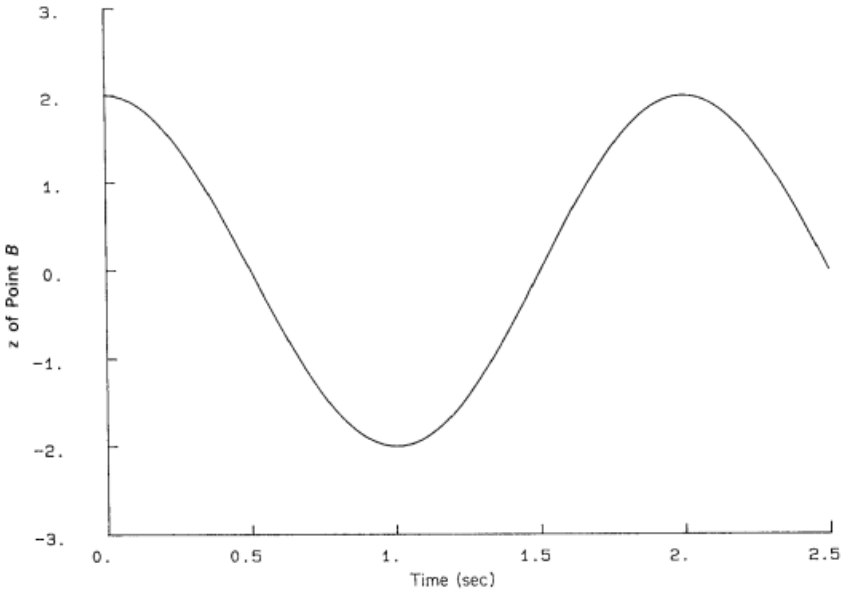


Figure 10.3.3 z coordinate of point B versus time, model 1.

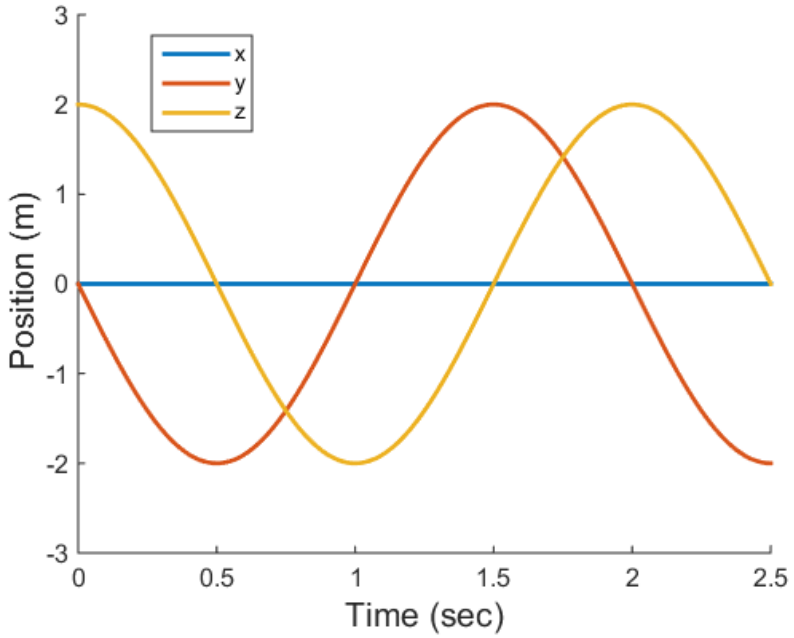


Figure 34: (Top) Textbook results for the position of point B in the four-bar mechanism when a constant angular velocity of π rad/sec is applied to the first link. (Bottom) Simulated results of the same mechanism.

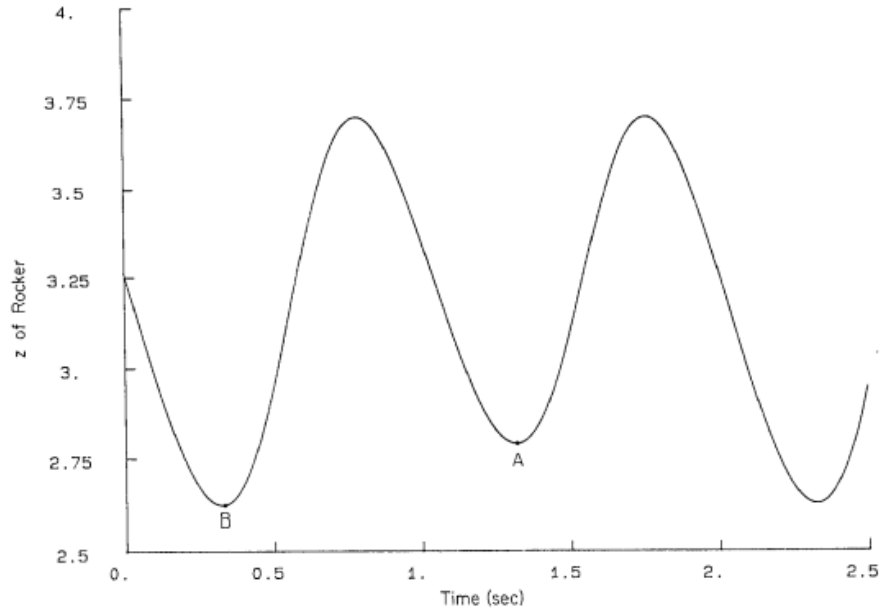


Figure 10.3.4 z coordinate of body 3 versus time, model 1.

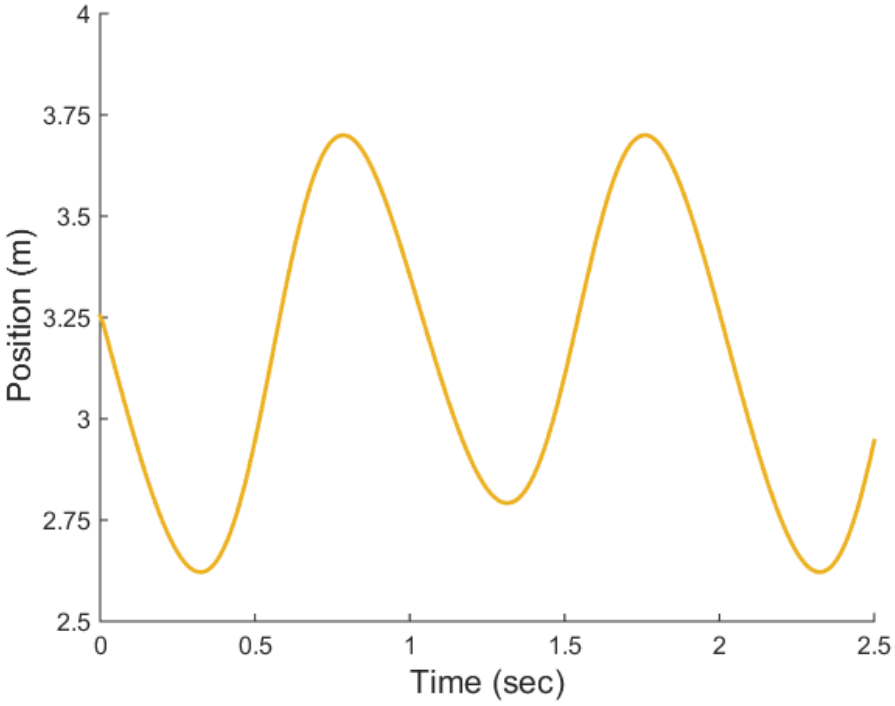


Figure 35: (Top) Textbook results for the position of the rocker in the four-bar mechanism when a constant angular velocity of π rad/sec is applied to the first link. (Bottom) Simulated results of the same mechanism.

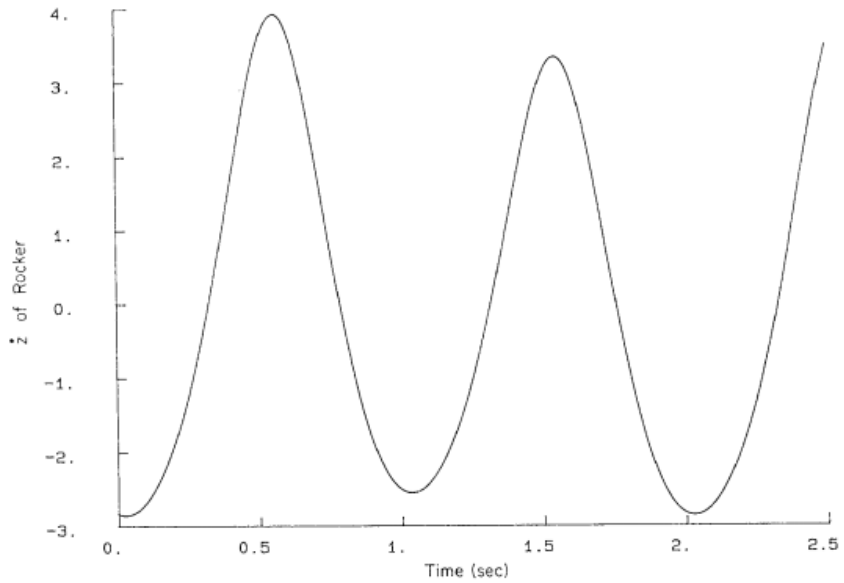


Figure 10.3.5 z velocity of body 3 versus time, model 1.

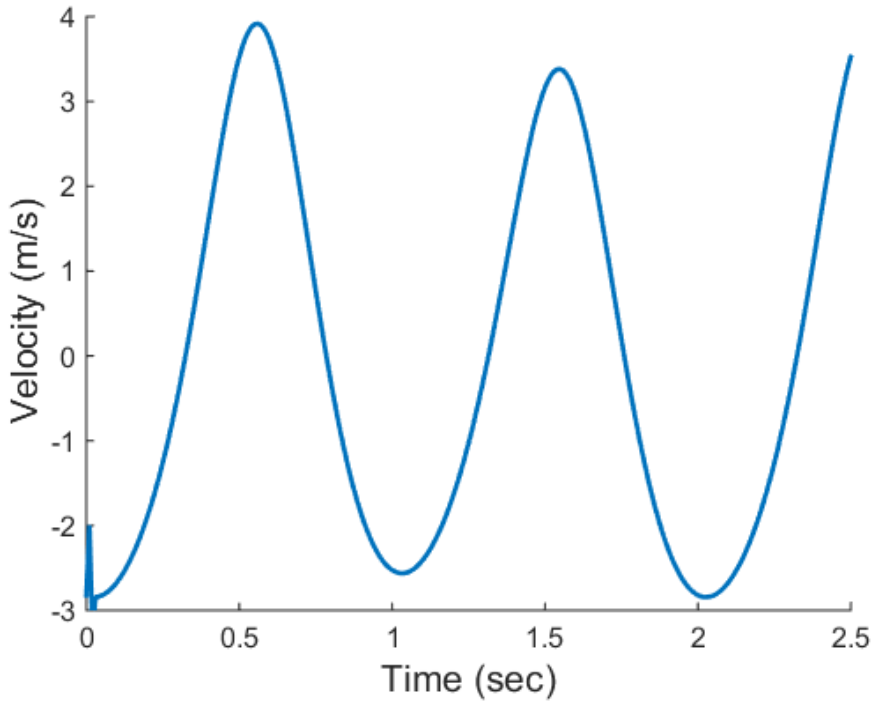


Figure 36: (Top) Textbook results of the velocity of the rocker in the four-bar mechanism when a constant angular velocity of π rad/sec is applied to the first link. (Bottom) Simulated results of the same mechanism.

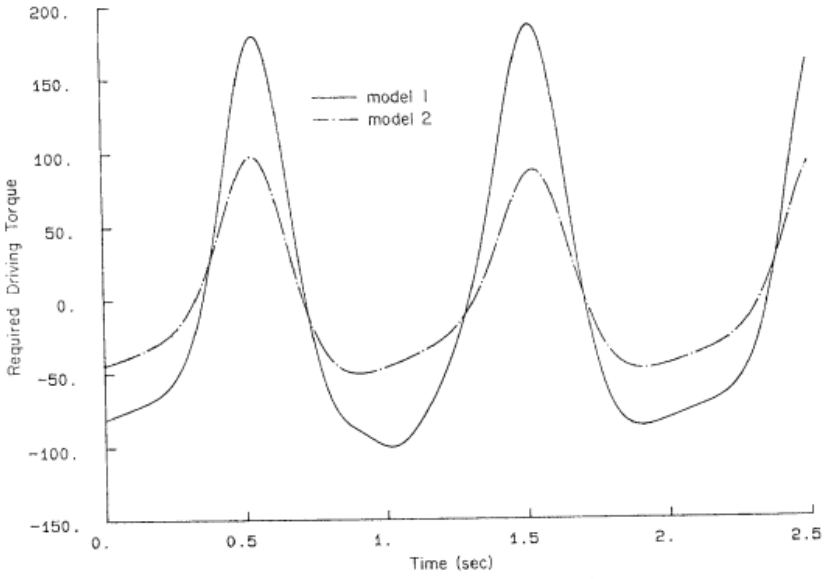


Figure 12.3.1(a) Driving torque at point *A*.

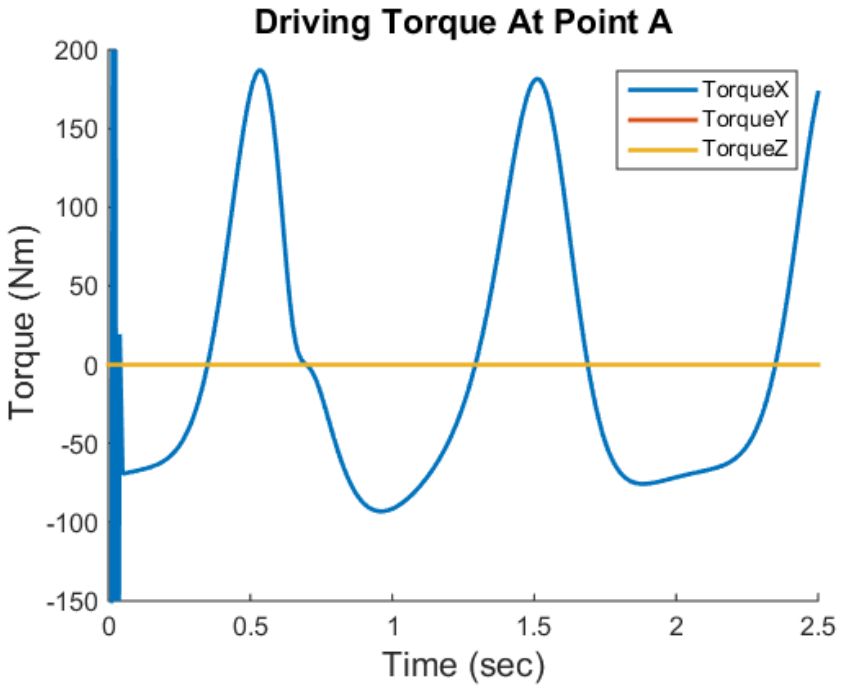


Figure 37: (Top) Textbook results for the driving torque applied at point A of the first link in the four-bar mechanism required to maintain a constant angular velocity of π rad/sec of the first link. (Bottom) Simulated results of the same mechanism.

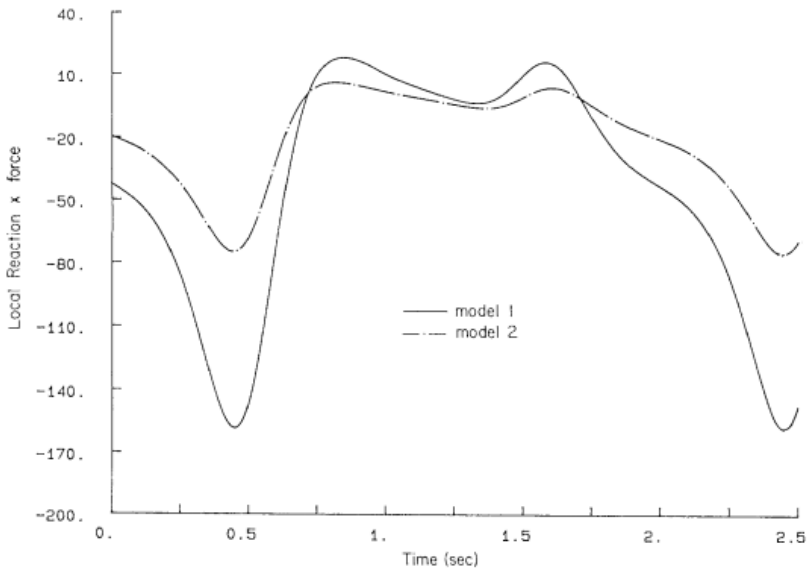


Figure 12.3.1(b) Reaction x force at point A.

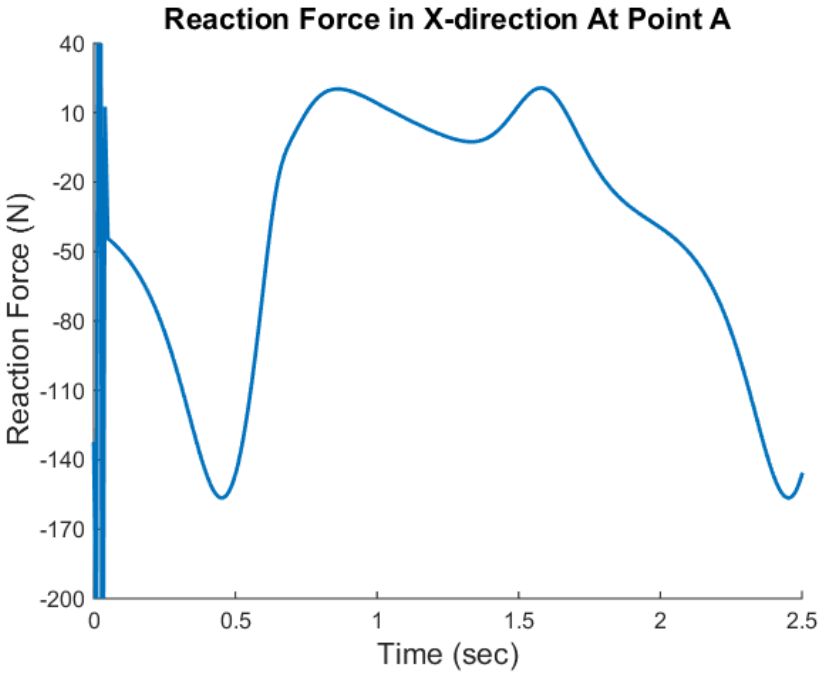


Figure 38: (Top) Textbook results for the x-component of the reaction force at point A in the four-bar mechanism when a constant angular velocity of π rad/sec is applied to the first link. (Bottom) Simulated results of the same mechanism.

Dynamics analysis with constant counterclockwise torque:

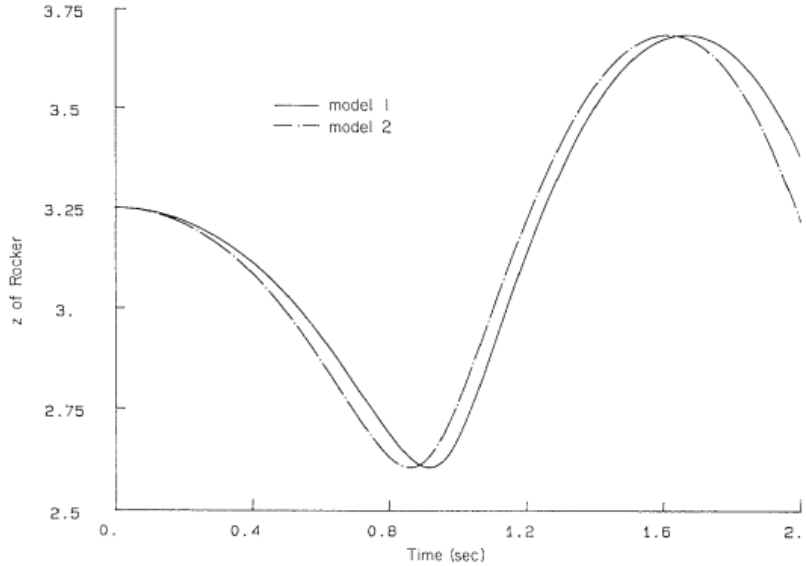


Figure 12.3.2(a) Position of centroid of link *CD*.

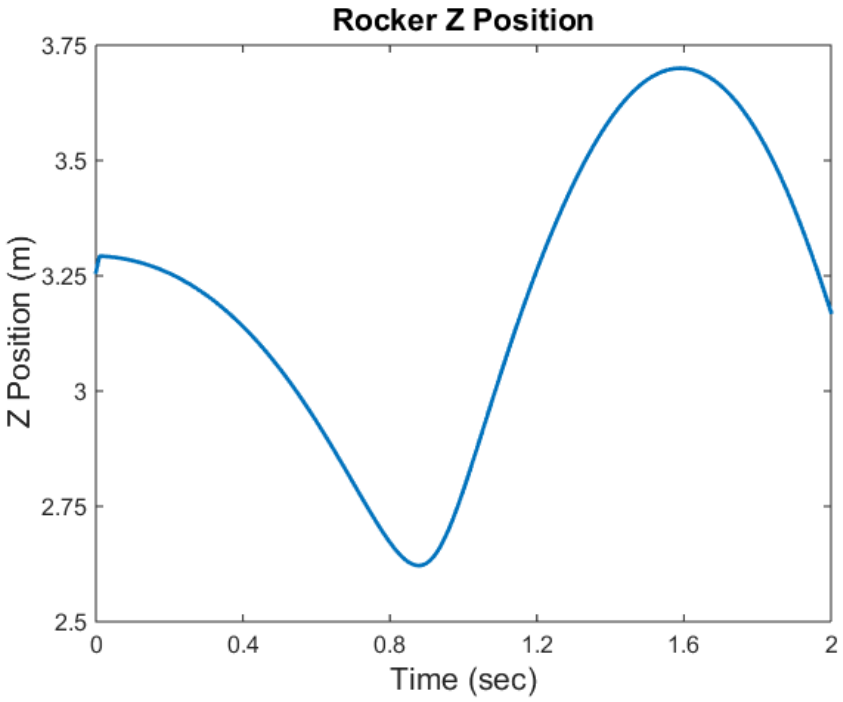


Figure 39: (Top) Textbook results for the z-position of the rocker in the four-bar mechanism when a constant torque 10 Nm is applied to the first link. (Bottom) Simulated results of the same mechanism.

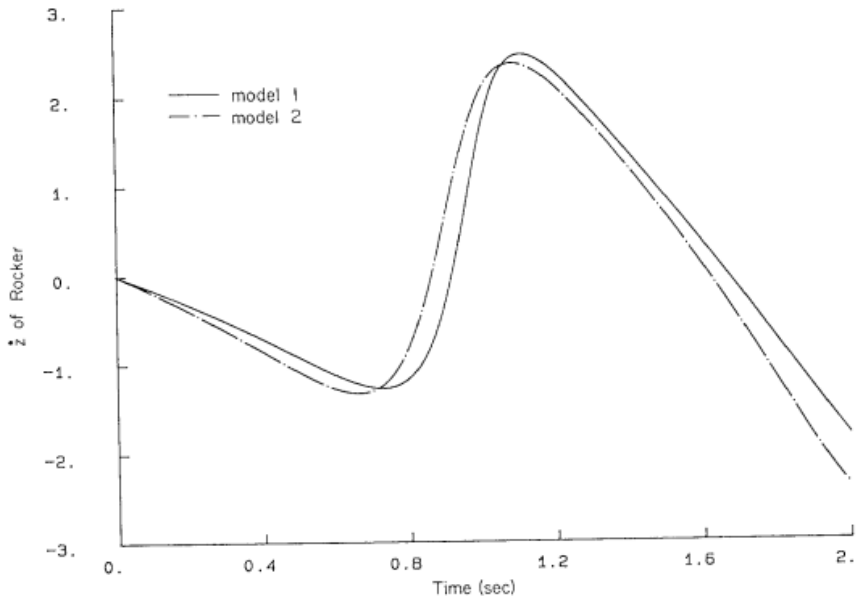


Figure 12.3.2(b) Velocity of centroid of link *CD*.

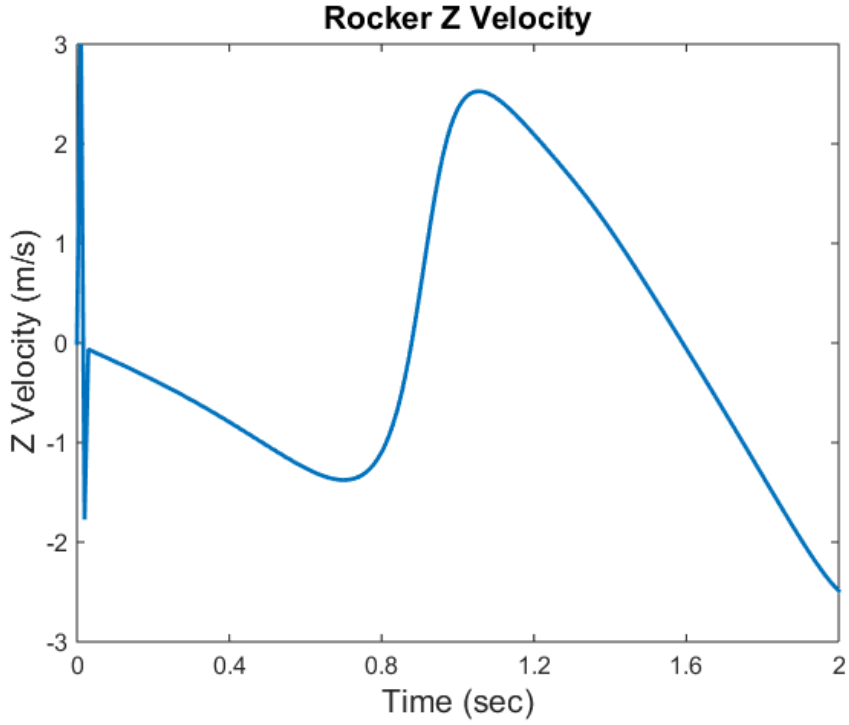


Figure 40: (Top) Textbook results for the z-velocity of the rocker in the four-bar mechanism when a constant torque 10 Nm is applied to the first link. (Bottom) Simulated results of the same mechanism.

7.1.2. Force and Torque Component Validation

Spring-Mass-Damper

Source of example:

No specific source for this example. This mechanism is a commonly seen mass-spring-damper system.

Purpose of validation:

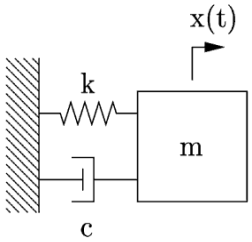
To validate the translational-spring-damper-actuator (TSDA) force element.

Brief overview of validation problem:

This problem is a classic mass-spring-damper system in which a spring-damper element is attached to a mass that translates along a frictionless surface. A translational joint was used to allow translation along the global x-axis. The analytical solution of the mass displacement was computed by solving the second-order ODE that represents this system using ODE45 in MATLAB. The parameters of the mass-spring-damper are as follows:

mass = 1 kg; k = 100 N/m; c = 2 kg/s; resting length = 0 m;

Image of mechanism:



Simulation Parameters:

Start time = 0

End time = 4

Step-size = 10^{-2}

BDF order = 2

Variation of Newton-Raphson = quasi Newton-Raphson

Comparison of results to previously validated results or to analytical results:

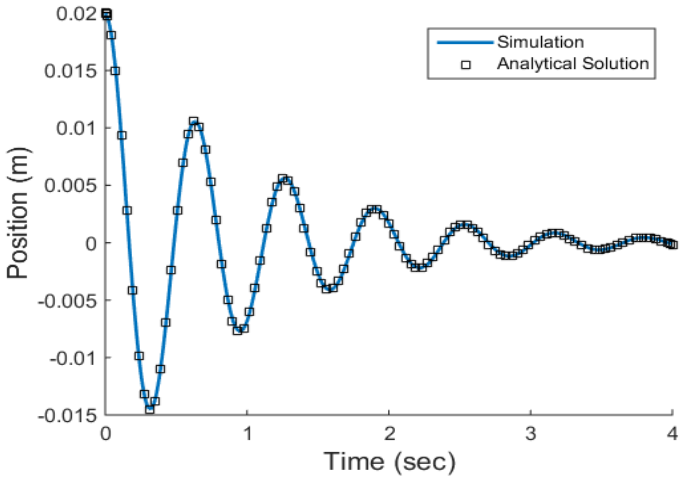


Figure 41: Comparison of analytical results to simulated results for a mass-spring-damper system

Flyball Governor

Source of example:

Chapter 12.6 of *Computer Aided Kinematics and Dynamics of Mechanical Systems*

Purpose of validation:

The purpose of simulating this flyball governor mechanism was to validate the ability to provide a variable torque to a body during a simulation.

Brief overview of validation problem:

This problem is a flyball governor mechanism that consists of 4 bodies (not including the ground), a TSDA force element, and two variable torques applied to the shaft (torques not shown in figure). One of the variable torques is a function of time and the other depends on the position of the collar. A definition of these variable torques can be seen in chapter 12.6 of *Computer Aided Kinematics and Dynamics of Mechanical Systems* or in the functions *Te.m* and *Ts.m* within the *flyballGovernorFromTextbook* folder. The simulation of this mechanism consists of a dynamics analysis with an initial velocity of 11.0174 rad/sec applied to the shaft. For validation, the y-location of the collar and the angular velocity of the shaft were compared to the textbook solutions.

Image of mechanism:

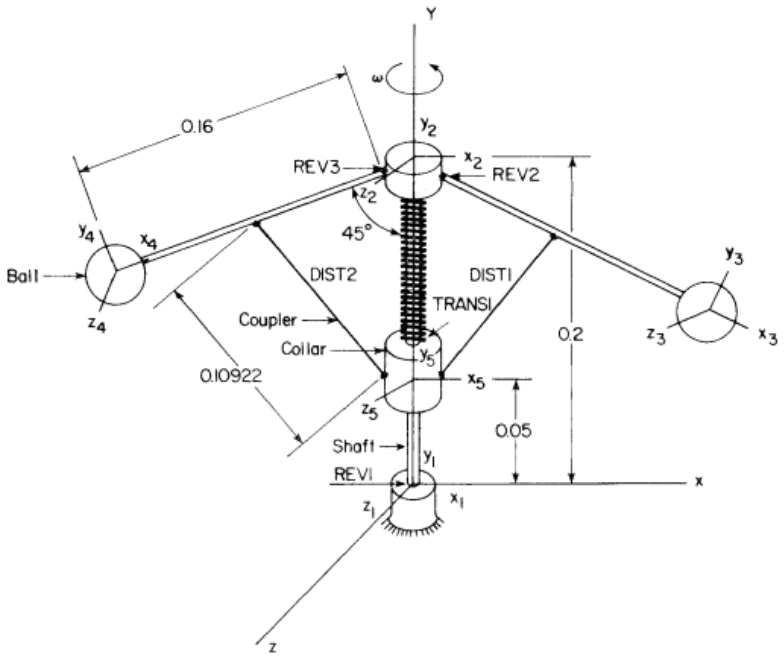


Figure 12.6.1 Governor mechanism.

Simulation Parameters:

Start time = 0

End time = 10

Step-size = 10^{-3}

BDF order = 2

Variation of Newton-Raphson = quasi Newton-Raphson

Comparison of results to previously validated results or to analytical results:

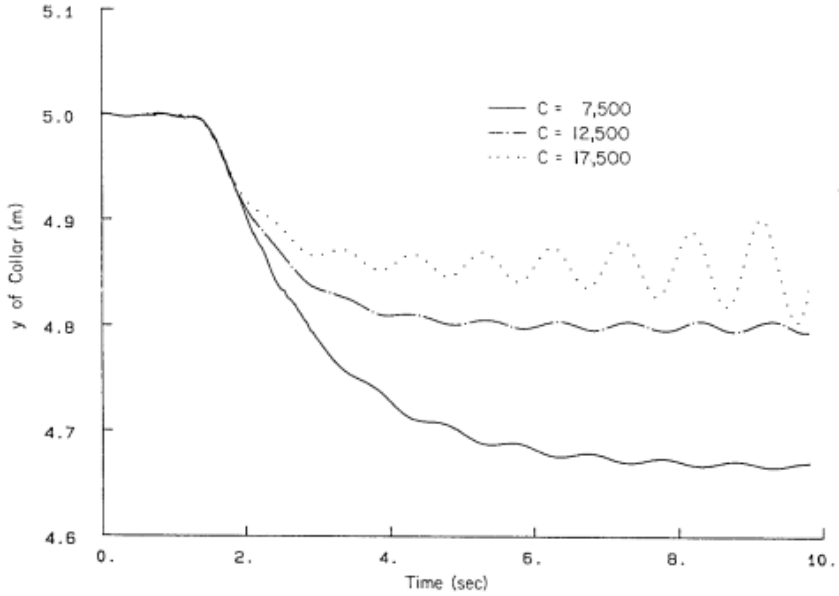


Figure 12.6.5 y of collar versus time ($k = 1000$).

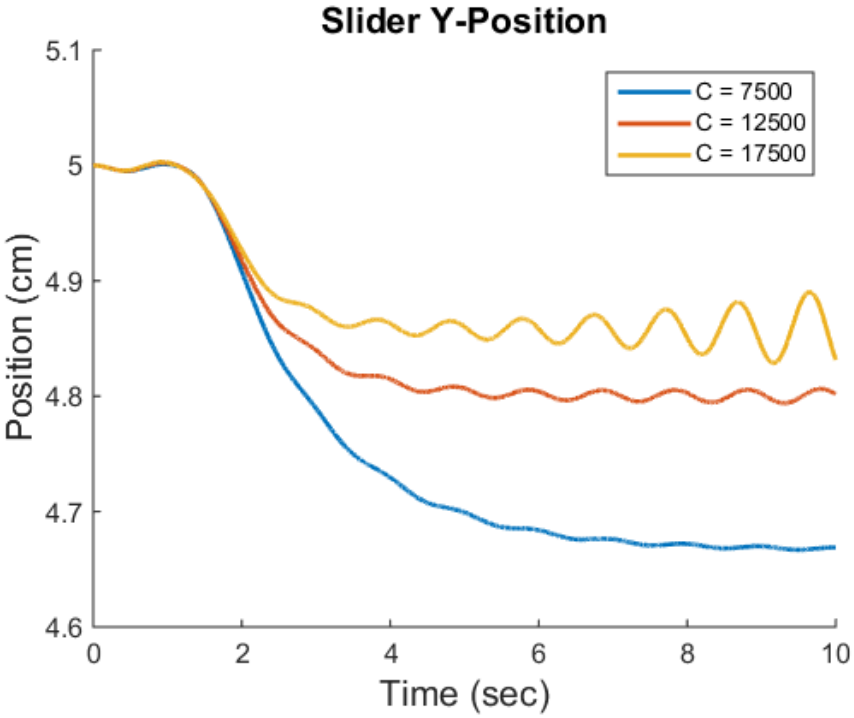


Figure 42: (Top) Textbook results for the y-position of the slider in the flyball governor mechanism with three different variable torques (different values of C) applied to the axis. (Bottom) Simulated results of the same mechanism.

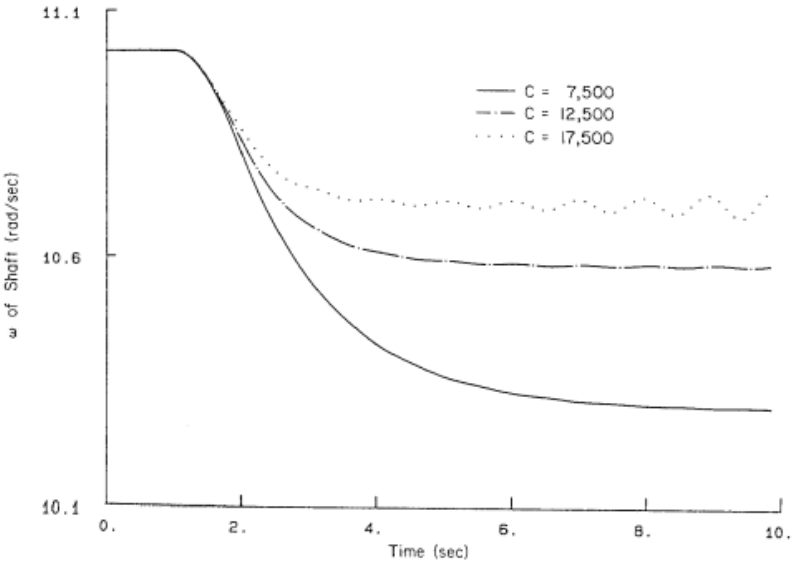


Figure 12.6.6 ω of shaft versus time ($k = 1000$).

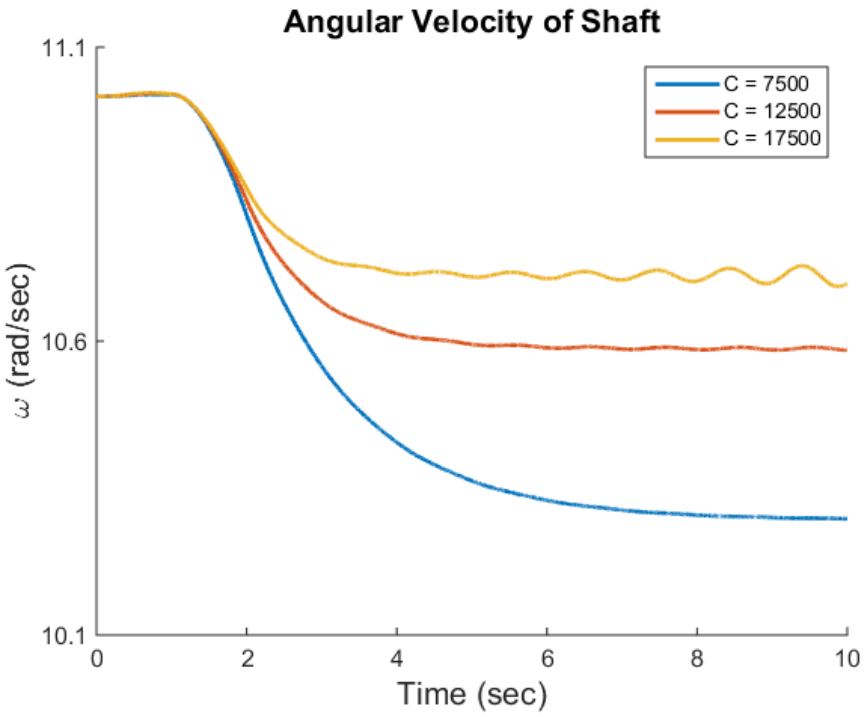


Figure 43: (Top) Textbook results for the angular velocity of the axis in the flyball governor mechanism with three different variable torques (different values of C) applied to the axis. (Bottom) Simulated results of the same mechanism.

7.2. Comparison to Benchmark Problems

Simple Pendulum with Revolute Joint

Simulation of this mechanism was already discussed at the beginning of section 7. However, it is also listed here because, in conjunction with validation the revolute joint, this mechanism is also a common benchmark problem.

N-Four Bar Mechanism

Source of example:

http://lim.ii.udc.es/mbsbenchmark/dist/A02/A02_specification.xml

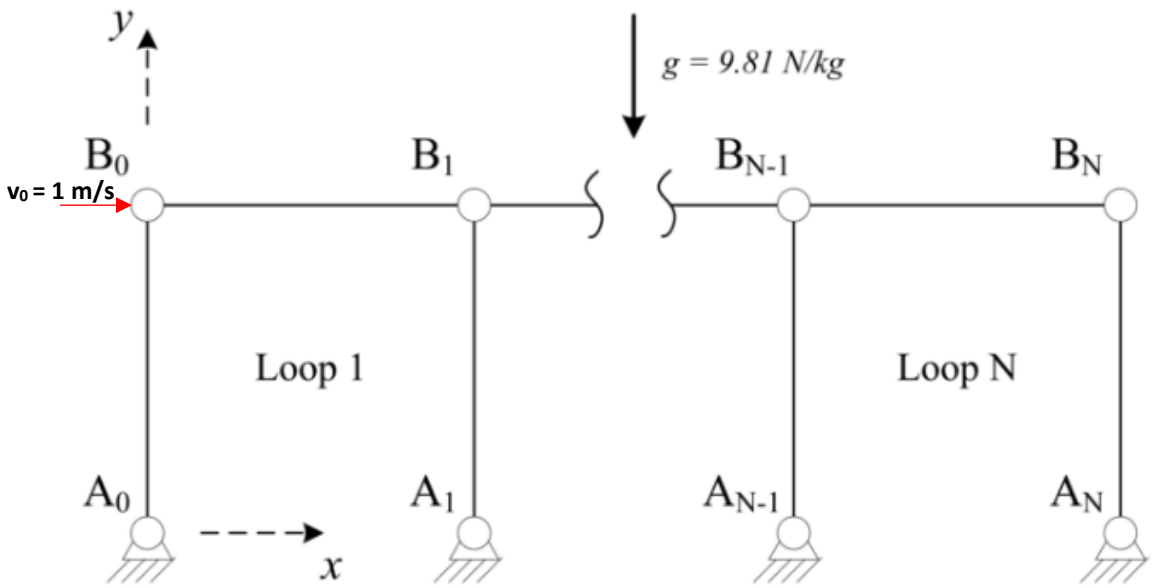
Purpose of validation:

The purpose of simulating this mechanism was to compare to a common benchmark problem.

Brief overview of validation problem:

This problem consists of an N-four bar mechanism, where N is the number of four bar mechanisms that are combined to create the overall system. The simulation of this mechanism consisted of a dynamic simulation with an initial velocity of 1 m/s in the +X direction for point B_0 (labeled in the below image).

Image of mechanism:



Simulation Parameters:

Start time = 0

End time = 20

Step-size = 10^{-2}

BDF order = 2

Variation of Newton-Raphson = quasi Newton-Raphson

Comparison of results to previously validated results or to analytical results:

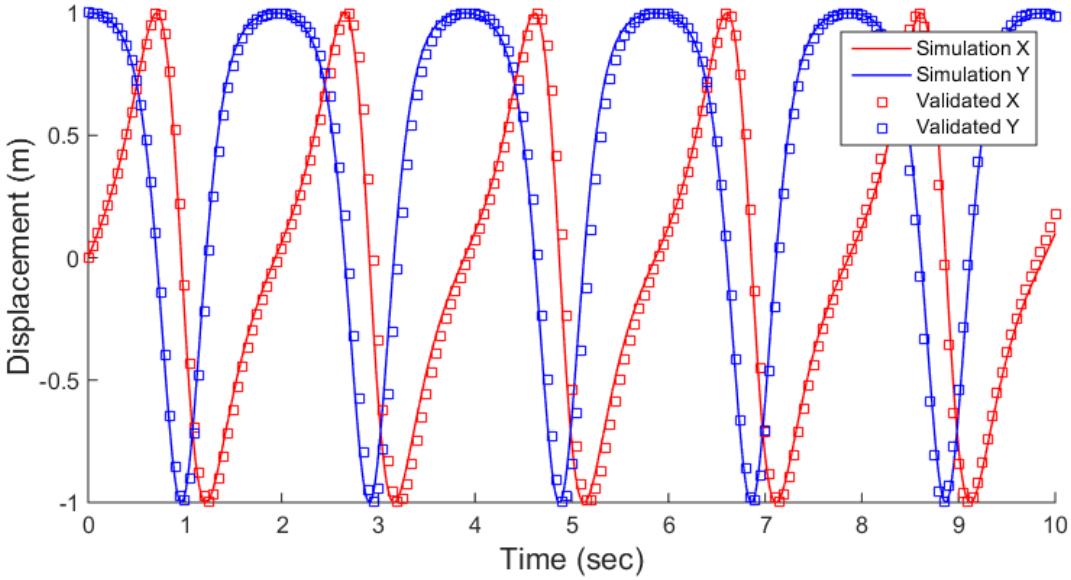


Figure 44: Previously validated results for the position of point B_0 (shown in figure) compared to simulated results for an N-four bar mechanism.