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

Michael F. Vignos

University of Wisconsin-Madison

# Table of Contents

# Introduction

Include a sentence or two about the purpose of this document

State that the purpose of this document is not to go into a very detailed, technical description of theory used to develop this simulation engine, but rather to give the reader a general understanding of how to use this engine, the capabilities and features of this engine, and an overview of the efforts to validate this simulation engine.

# Problem Statement

The goal of this work was to develop a 3D multibody kinematics and dynamics simulation engine within MATLAB. The theory used to develop this simulation engine was taken from the following sources:

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.

# simEngine3D Framework

As previously mentioned, all code for this simulation engine 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.

## Organization of GitHub Repository

The GitHub repository for this code is named simEngine3D-Vignos. Within this repository there are 3 main folders: *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.

## Object-Oriented Programming

The framework of simEngine3D was written using object-oriented programming. Using object-oriented programming allows for the creation of MATLAB classes, which can 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 animated the multibody system. The purpose of each class and the *plot* commands will be covered in a bit more detail in the following sections.

### multibodySystem.m

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 define the multibody system will contain all of the data stored throughout the simulation (e.g. kinematics of each body, reaction forces and torques, etc.). Additionally, since the these data are stored at each time step, a simulation can be stopped prematurely and the data contained within the instance of this class can be visualized to see what was occurring with the simulation up to the stopping time.

### body.m

An instance of this class contains all of the attributes of a body (e.g. mass, body number, kinematics, etc.) that are used when performing 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 and makes it easier to compute state information for a single body in post-processing.

### Basic Constraint Classes: CDconstraint.m, Dconstraint.m, DP1constraint.m, and DP2constraint.m

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.

### simEngine3DUtilities.m

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.).

### 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.

## Example Model Definition

Figure 2 below contains a screen shot of an example model definition within a driver script. 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. This concept of defining joints using basic constraints will be further discussed in the following section. More thorough examples of model definitions can be found in the *testExamples* folder in the *simEngine3D-Vignos* repository.

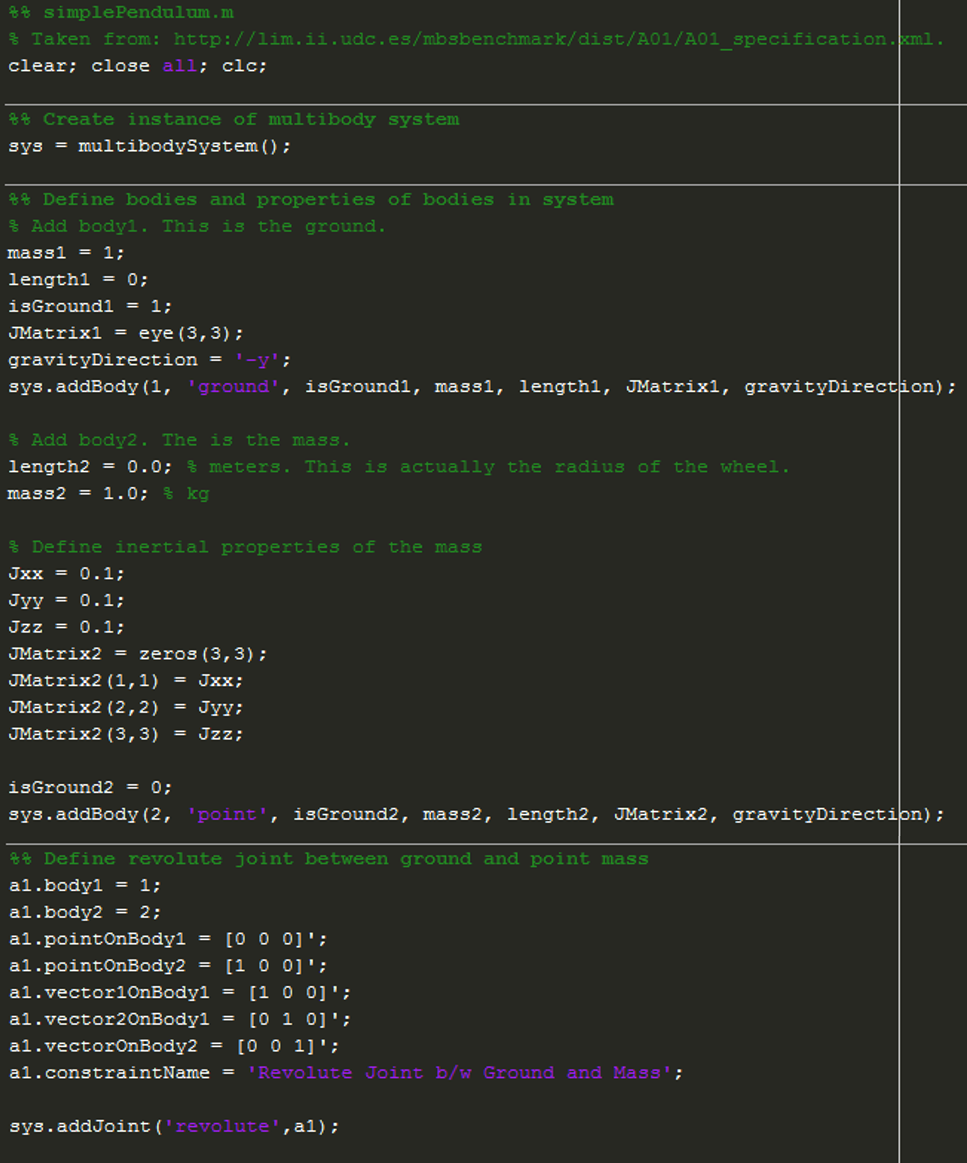


Figure 1: Example driver file showing the definition of a simple pendulum. This example contains two bodies (the ground and a point mass) and a revolute joint between them.

# System Constraints

## 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 a collection 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.

### 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.

#### Coordinate Difference (CD) Constraint

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

1. The two bodies to which the constraint is applied (i.e. body 1 and body 2)
2. A point attached to body 1 in the body 1 reference frame
3. A point v body 2 in the body 2 reference frame
4. The Cartesian coordinate that is being constrained
5. The prescribed distance between the coordinates of the two bodies

#### Distance (D) Constraint

A D constraint defines the distance between two bodies. The attributes for this constraint are as follows:

1. The two bodies to which the constraint is applied (i.e. body 1 and body 2)
2. A point attached to body 1 in the body 1 reference frame
3. A point attached to body 2 in the body 2 reference frame
4. The prescribed squared distance between the two bodies. 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 1 and one attached to body 2, to be equal to a prescribed function. 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 1 and body 2)
2. A vector attached to body 1 in the body 1 reference frame
3. A vector attached to body 2 in the body 2 reference frame
4. A function prescribing the value of the dot product between the two vectors. 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. However, in this case one vector is attached to body 1 and the second vector is defined as the vector between two points. One of these points is attached to body 1 and the other is attached to body 2. The attributes for this constraint are as follows:

1. The two bodies to which the constraint is applied (i.e. body 1 and body 2)
2. A vector attached to body 1 in the body 1 reference frame
3. A point attached to body 1 in the body 1 reference frame
4. A point attached to body 2 in the body 2 reference frame
5. A function prescribing the value of the dot product between the two vectors. If these vectors are defined to be orthogonal, this function must be zero.

### 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 2 to be orthogonal to a plane attached to body 1. The plane on body 1 is defined by two vectors attached to body 1. The attributes for this constraint are as follows:

1. The two bodies to which the constraint is applied (i.e. body 1 and body 2)
2. A vector attached to body 1 in the body 1 reference frame
3. Another vector attached to body 1 in the body 1 reference frame. This vector and the previous vector define the plane on body 1.
4. A vector attached to body 2 in the body 2 reference frame

#### Perpendicular Two (B2) Constraint

Similar to a B1 constraint, a B2 constraint constrains a vector to be orthogonal to a plane attached to body 1. However, in this case the vector is defined as the vector between a point attached to body 1 and a point attached to body 2. The attributes for this constraint are as follows:

1. The two bodies to which the constraint is applied (i.e. body 1 and body 2)
2. A vector attached to body 1 in the body 1 reference frame
3. Another vector attached to body 1 in the body 1 reference frame. This vector and the previous vector define the plane on body 1.
4. A point attached to body 1 in the body 1 reference frame
5. A point attached to body 2 in the body 2 reference frame

### Joints

To date, there are six joints that have been implemented and validated in simEngine3D. Each of these joints are constructed from a 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

1. The two bodies connected by this joint (i.e. body 1 and body 2)
2. A point attached to body 1 in the body 1 reference frame.
3. A point attached to body 2 in the body 2 reference frame. This second point will be constrained to be in the same location as the point defined in 2.

#### Revolute Joint Attributes

1. The two bodies connected by this joint (i.e. body 1 and body 2)
2. A point attached to body 1 in the body 1 reference frame.
3. A point attached to body 2 in the body 2 reference frame. This second point will be constrained to be in the same location as the point defined in 2.
4. A vector attached to body 1 in the body 1 reference frame. This vector is the 1st vector used to define the plane in which the revolute joint moves.
5. A second vector attached to body 1 in the body 1 reference frame. This vector is the 2nd vector used to define the plane in which the revolute joint moves.
6. A vector attached to body 2 in the body 2 reference frame. This vector is orthogonal to the plane on body 1 (i.e. the plane defined by the vectors defined in 4 and 5). In other words, this is the vector about which the revolute joint rotates.

#### Cylindrical Joint Attributes

1. The two bodies connected by this joint (i.e. body 1 and body 2)
2. A point attached to body 1 in the body 1 reference frame that is along the translational axis of body 1.
3. A point attached to body 2 in the body 2 reference frame that is along the translational axis of body 2.
4. A vector attached to body 1 in the body 1 reference frame. 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 1 in the body 1 reference frame. 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 2 in the body 2 reference frame. This vector is orthogonal to the plane on body 1 (i.e. the plane defined by the vectors defined in 4 and 5).

#### Translational Joint Attributes

1. The two bodies connected by this joint (i.e. body 1 and body 2).
2. A point attached to body 1 in the body 1 reference frame that is along the translational axis of body 1.
3. A point attached to body 2 in the body 2 reference frame that is along the translational axis of body 2.
4. A vector attached to body 1 in the body 1 reference frame. 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 1 in the body 1 reference frame. 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 2 in the body 2 reference frame. 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 2 in the body 2 reference frame. This vector is parallel to the plane on body 1 (i.e. the plane defined by the vectors defined in 4 and 5) and perpendicular to the vector defined in 4.

#### Universal Joint Attributes

1. The two bodies connected by this joint (i.e. body 1 and body 2)
2. A point attached to body 1 in the body 1 reference frame
3. A point attached to body 2 in the body 2 reference frame. This second point will be constrained to be in the same location as the point defined in 2.
4. A vector attached to body 1 in the body 1 reference frame that defines the rotation axis of this joint for body 1.
5. A vector attached to body 2 in the body 2 reference frame that defines the rotation axis of this joint for body 2. This vector is orthogonal to the vector defined in 4.

#### Revolute-Cylindrical Joint Attributes

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 a distance constraint.

1. The two bodies connected by this joint (i.e. body 1 and body 2)
2. A point attached to body 1 in the body 1 reference frame that is along the translational axis of body 1.
3. A point attached to body 2 in the body 2 reference frame that is along the translational axis of body 2.
4. A vector attached to body 1 in the body 1 reference frame that defines the rotational axis of the revolute portion of the joint.
5. A vector attached to body 2 in the body 2 reference frame that is along the translational axis of this joint. This vector is orthogonal to the vector defined in 4.
6. A vector attached to body 2 in the body 2 reference frame. 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.
7. A second vector attached to body 2 in the body 2 reference frame. 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.

## 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 with respect to time of the free degree of freedom of this joint.

# 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.

## 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 (FIGURE). 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 (Figure 1). However, the user must properly specify the body mass and the gravity direction to ensure proper magnitude and direction of the gravity force.

## 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 (FIGRURE). 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 (FIGURE).

## 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 (EQUATION SHOWING HOW TO COMPUTE MAGNITUDE AND DIRECITON OF FORCE). 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 the actuator function equal to zero (FIGURE).

# 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.

## 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, simEngine3D contains a function (*A2p()*) that converts an orientation matrix to Euler parameters. 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 *(FIGURE SHOWING USE OF SETINITIALPOSE)*.

## 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 (EQUATION SHOWING OPTIMIZATION PROBLEM) that aims to find a consistent initial pose that is as close as possible to the user-specified initial pose. 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()*.

## 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 (FIGURE).

### Case 2: Initial velocity known for a subset of the bodies.

In this case, the user can compute and set the initial velocity for the rest of the bodies in the system using the function *computeAndSetIntialVelocities()* and providing the body numbers with known velocities and the initial velocities of these bodies as inputs to this function (FIGURE). This function then computes the initial velocity for the rest of the bodies, such that these velocities satisfy the velocity-level constraints (EQUATION).

### Case 3: Initial velocity not known for any of the bodies.

This case is handled in a similar fashion to the previous case in that the velocity of each of the bodies is computed and set using the function *computeAndSetIntialVelocities()*. However, 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 (FIGURE). Initial velocities that satisfy the velocity-level constraints are then computed for all bodies in the system (EQUATION).

## Kinematics Analysis

A kinematics analysis is used to compute the position, velocity, and acceleration of each body in a multibody system throughout a simulation. Prior to beginning a kinematics analysis, the user must begin with a multibody 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 constrain only 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 (EQUATION). 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 (EQUATIONS). 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*.

## 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 also 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.

## 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 (EQUATION). , which in turn 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 positions, velocities, and accelerations of the system at each timestep. 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 to compute the time-evolution of a system, solving the equations of motion across a discretized time grid also requires simEngine3D to solve a nonlinear system of equations at each time step. 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 their iteration matrix and in the number of times that the iteration matrix is computed. Similar to choosing the order of the BDF method, the desired variation of the Newton-Raphson method can be specified when performing the dynamics analysis.

### Full Newton-Raphson

The full Newton-Raphson method is typically the slowest to converge in that

### Modified Newton-Raphson

### Quasi Newton-Raphson

# 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 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.

## Validation of Model Components

### Joints

#### Simple Pendulum with Revolute Joint

**Source of example:**

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

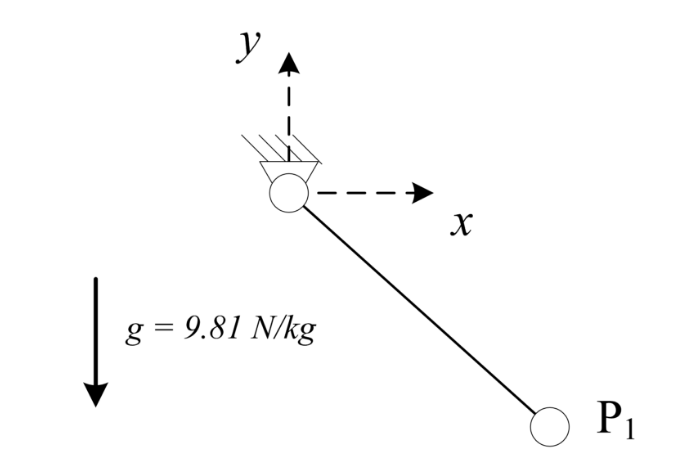
**Purpose of validation:**

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

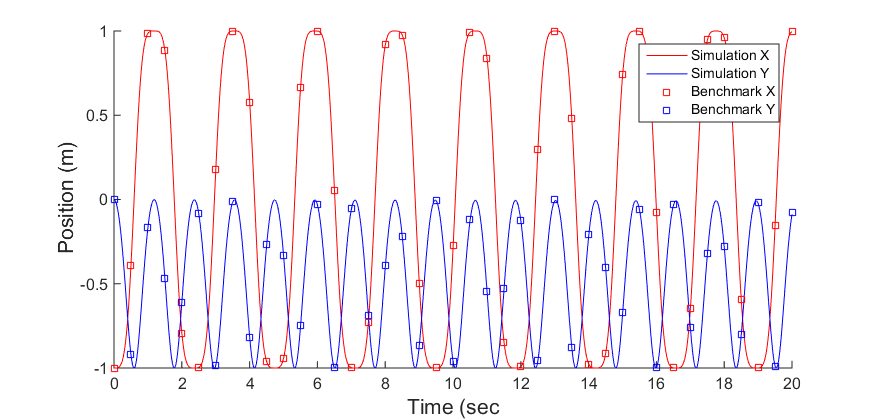
**Brief overview of validation problem:**

The problem is a simple pendulum that consists of a point mass connected to the ground via a revolute joint.

**Image of mechanism:**



**Comparison of results to previously validated results:**



#### Simple Pendulum with Cylindrical Joint

**Source of example:**

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

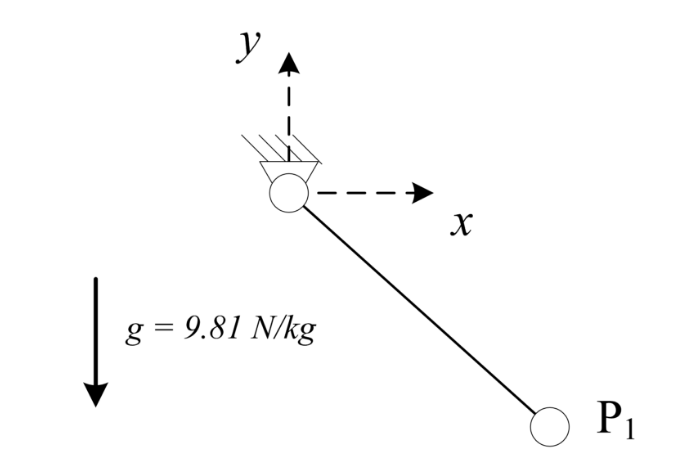
**Purpose of validation:**

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

**Brief overview of validation problem:**

This problem consisted of a simple pendulum with a point mass connected to the ground by a cylindrical 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/s2 acceleration to the mass. This confirmed that the cylindrical was translation as it should.

**Image of mechanism:**



**Comparison of results to previously validated results:**

#### Block on Incline Plane with Translation Joint

**Source of example:**

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

**Purpose of validation:**

**Brief overview of validation problem:**

**Image of mechanism:**

**Comparison of results to previously validated results:**

#### Slider-Crank Mechanism

**Source of example:**

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

**Purpose of validation:**

**Brief overview of validation problem:**

**Image of mechanism:**

**Comparison of results to previously validated results:**

#### Four Bar Mechanism

**Source of example:**

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

**Purpose of validation:**

**Brief overview of validation problem:**

**Image of mechanism:**

**Comparison of results to previously validated results:**

### Force and Torque Components

#### Spring-Mass-Damper

**Source of example:**

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

**Purpose of validation:**

**Brief overview of validation problem:**

**Image of mechanism:**

**Comparison of results to previously validated results:**

#### Flyball Governor

**Source of example:**

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

**Purpose of validation:**

**Brief overview of validation problem:**

**Image of mechanism:**

**Comparison of results to previously validated results:**

## Comparison to Benchmark Problems

#### Simple Pendulum

**Source of example:**

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

**Purpose of validation:**

**Brief overview of validation problem:**

**Image of mechanism:**

**Comparison of results to previously validated results:**

#### N-Four Bar Mechanism

**Source of example:**

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

**Purpose of validation:**

**Brief overview of validation problem:**

**Image of mechanism:**

**Comparison of results to previously validated results:**

# 

Things to do still:

~~Implement simple pendulum~~

~~Implement N-bar mechanism~~

~~Implement method to prescribe initial velocities~~

~~Implement example to validate a cylindrical joint (could maybe be a simple pendulum with a cylindrical joint instead of a revolute joint??)~~

Implement method to prescribe angular velocity at a joint, rather than using a DP1 constraint. A DP1 constraint is plagued with too many singularity issue.

~~Implement method to remove redundant constraints??~~ Not sure how robust my method is.

Implement ability to read model parameters from a file???

~~Flyball governor mechanism?~~

Bricard’s mechanism?

Andrew’s mechanism?

Things done:

~~Implemented ability to prescribe all basic constraints, all intermediate constraints, and all joints discussed in class.~~

~~Implemented ability to perform kinematics, inverse dynamics, and dynamics analyses.~~

~~Validated all joints~~

~~Validated ability to prescribe constant torque  
Implemented kinematics, inverse dynamics, and dynamics analyses~~

~~Implemented different methods of computing iteration matrix~~

~~Implemented various test cases validated by book and by group that created the benchmark problems~~

~~Implemented method to prescribe initial velocities for dynamics analysis.~~