ANDY Tutorial

Robotics and Mechatronics Lab, Virginia Tech

January 2019

1. Introduction

ANDY is an efficient symbolic multibody toolbox capable of modeling nonholonomic, constrained and hybrid dynamic systems. The toolbox is inherently developed in MATLAB and does not require external add-on or extra configurations. The toolbox can provide symbolic model information that helps you to analytically analyze and design system dynamics, while it also uses numerical modeling method to efficiently calculate properties in simulations. In brief, the toolbox can be convenient and helpful if you would like to use it for:

1. Simulating and observing newly developed controllers performances.
2. Model preparation for model predictive controller or intelligent controller designs.
3. Optimizing mechanism dimensions based on kinematic and dynamic properties.

The toolbox can also be used to model Compared to Adams, V-Rep, Drake and other multibody software, the disadvantages of ANDY are:

1. Difficulties in modeling geometry based dynamic effects (line, surface collisions, etc.).
2. Slight lower numerical calculation efficiency.

The toolbox adopted Kane's method as its main modeling principle. The theoretical details of this toolbox can be found in a previous paper published by RMLAB. As the toolbox library is documented, the detailed software design will also not be discussed here. The main purpose of the tutorial is to demonstrate the proper modeling and simulation process. The exemplary cases can be found in toolbox root folder. The ***Quadruped*** Example will be used as the major guide in this tutorial. Other examples will also be used to demonstrate some of the features uncovered by the ***Quadruped*** Example.

1. Initialization and Variable Declaration

The initialization and variable declaration script can be found in Modeling.m. Initialization of a modeling case starts with the PathSetup, which includes the paths of library utilities and cleans up the cache. Be sure to check the relative path between the project folder and the library. It is also recommended to run PathSetup twice, as sometimes MATLAB appears to not being able to completely clean up the cache, which would introduce some errors not related to the toolbox library or modeling.

%% Path Setup

PathSetup;

PathSetup;

%% Declare Time Variable and System

t=sym('t','real');

System=kSystem('JoeQPed',t);

WORLD=System.Space.RootFrame;

System.Space.setPrecompile(false);

System.Model.setPrecompile(false);

In the above code, a time symbolic variable is first declared as 't'. The time variable will serve as the tag for time-differentiable vectors. In ANDY, a variable may have multiple tags added to the end of its name. For example, a\_\_dt\_1\_ indicates variable a tagged with dt (time-differentiable) with the property of 1 (1st Order). It should be noted that the time variable or any other symbolic variable used as tags should not have any tags with in their own naming.

Next, the System object is created by the class kSystem, which is the root object of the modeling project. Here, the naming of 'JoeQPed' is given along with the specified time variable as t. WORLD is defined as the world frame kFrame object, which is the same as System.Space.RootFrame. The two major properties of kSystem are System.Space and System.Model of the class kSpace and kModel respectively, which are the objects that includes and compiles the kinematic and dynamic models. Both objects has the setPrecompile method that allows configurations to select/unselect precompiled .mex functions generation. The JIT (Just-In-Time) .mex functions requires MATLAB Coder Toolbox to generate, which can significantly improve the calculation efficiency. If the Coder Toolbox is not available, set pre-compilation to false will still allow the smooth modeling of the system.

%% Declare Constant Parameters

C=System.genParam({

'l\_sh' 'Leg Shank Length' 0.3;

'l\_th' 'Leg Thigh Length' 0.3;

'l\_t' 'Tail Length' 0.6;

'b\_w' 'Torso Joint Width' 0.48;

'b\_l' 'Torso Joint Length' 0.54;

'd\_t' 'Tail Distance for Torso Center' 0.34;

'm\_sh' 'Shank Mass' 1.2289;

'm\_th' 'Thigh Mass' 0.9453;

'm\_b' 'Torso Mass' 26.9078;

'm\_t' 'Tail Mass' 1.4347;

'I\_bx' 'Torso Moment X' 0.8404;

'I\_by' 'Torso Moment Y' 0.3793;

'I\_bz' 'Torso Moment Z' 1.2125;

'g' 'Gravity Acceleration' 9.8067;

});

The constants are fixed values generated by System.genParam in the above notion. For example, in the first row, 'l\_sh' stands for the symbolic variable, 'Leg Shank Length' is the notation of this variable, and 0.3 is the fixed value. Both the symbolic variable and the notation need to be unique, otherwise, some of the variables will be overwritten. The variables will be put in a structure variable, which is defined as C here. The symbolic variable of 'l\_sh' can simply be used in the following by referring to C.l\_sh. The constant symbols will appear in the symbolic properties, which will be substituted when the kinematic and dynamic model functions are generated.

%% Declare Continuous States (Generalized Coordinates)

[qtx,qty,qtz,qrx,qry,qrz]...

=System.genCont({

'qtx' 'Body X Translational Displacement';

'qty' 'Body Y Translational Displacement';

'qtz' 'Body Z Translational Displacement';

'qrx' 'Body X Rotary Displacement';

'qry' 'Body Y Rotary Displacement';

'qrz' 'Body Z Rotary Displacement';

});

Next, the System.genCont is used to generate the continuous variables. These variables will be used as the generalized coordinates for the multibody system. While initial naming of the symbolic variables are not tagged (such as 'qtx'), the time-differentiable tags will be added to them automatically. As an example, to use the symbolic variable 'qtx', simply use qtx.dot(i)to generate an ith order variable of . Similar to before, both the naming of the symbolic variables and their notations need to be unique. This feature also applies to the following classes of variables.

%% Declare Discrete States

[p\_LF]...

=System.genDisc({

'p\_LF\_X' 'Left Fore Foot Contact Point X';

'p\_LF\_Y' 'Left Fore Foot Contact Point Y';

});

…

%% Declare Inputs

[u\_LF]...

=System.genInput({

'u\_LF\_H' 'Left Fore Leg Hip Joint Torque';

'u\_LF\_T' 'Left Fore Leg Thigh Joint Torque';

'u\_LF\_S' 'Left Fore Leg Shank Joint Torque';

});

The effects of discrete states and inputs are not distinctly different - they are both non time-differentiable. The only difference between these two is that inputs will be used to generate input Jacobian matrices. The generalized forces from inputs will also be distinguished from the other generalized forces. Discrete states on the other hand are more common used for system parameters or in hybrid systems. After generation, the symbolic variables of these states will also be put into structure objects and can be used similarly to constants.

%% Declare Nonholonomic States (Not Part of the Example)

[alphaNH]...

=System.genNHSignal({

'alphaNH' 'NHSignal alphaNH';

});

alphaNH.setExpr(qtx.dot(0));

The final class of variable is the nonholonomic state. These states are not as commonly used as the previous variables. The current examples does not have direct declarations of the nonholonomic states. Therefore, the above exemplary code shows the declaration notion. The method setExpr is used to define the 1st order time-differentiation expression of alphaNH. For nonholonomic kinematic link between frames (such as Quaternion based 3D rotation), nonholonomic states will be declared automatically.

1. Initialization and Variable Declaration

After the declaration of the symbolic variables, the kinematic modeling process is initiated, which can be found in Modeling\_Kinematics.m. The

%% Declare Coordinate Frames

…

[TailRoll,TailYaw,TailCOM]...

=System.Space.genNode({

'TailRoll';

'TailYaw';

'TailCOM';

});