Drake
A planning, control, and analysis toolbox for nonlinear dynamical systems

Robot Locomotion Group

December 9, 2013

Contents

1 Introduction and Goals **5**

1.1 What is Drake? 5

1.1.1 Relative to Simulink and SimMechanics 6

1.1.2 For controlling real hardware 6

1.2 Getting Started 7

2 Modeling and Simulation **9**

2.1 Modeling Input-Ouput Dynamical Systems 9

2.1.1 Writing your own dynamics 9

2.1.2 Existing Simulink Models/Blocks 17

2.2 Combinations of Systems 17

2.2.1 Coordinate frames 18

2.3 Simulation 18

2.4 Visualization 19

2.4.1 Outputing to a movie format 20

3 Analysis **21**

3.1 Fixed Points 21

3.1.1 Local Stability 21

3.1.2 Global Stability 21

3.1.3 Regions of Attraction 21

3.2 Limit Cycles 21

3.2.1 Local Stability 21

3.2.2 Regions of Attraction 21

3.3 Trajectories 21

3.3.1 Finite-time invariant regions 21

3.4 Stochastic Verification 21

4 Planning **23**

4.1 Trajectory Optimization 23

4.2 Randomized Motion Planning 23

5 Feedback Design **25**
5.1 Linear Quadratic Regulators 25
5.2 Robust Control 25
5.3 Sums-of-Squares Design 25

6 System Identification **27**

7 State Estimation **29**

8 Rigid Body Systems **31**
8.1 Forward Kinematics 31
8.2 Inverse Kinematics 31
8.3 Kinematic Planners 31
8.4 Contact Dynamics 31

9 External Interfaces **33**

A For Software Developers **35**
A.1 Code Style Guide 35
A.2 Check-In Procedures 36
A.2.1 Unit tests 37
A.2.2 Run all tests 37
A.2.3 Matlab Reports 37
A.2.4 Contributing Code 37
A.3 Version Number 37

B SolidWorks to URDF Exporting **39**
B.1 Drake-specific information for those already familiar with the export tool: 39
B.2 Export a SolidWorks Assembly to URDF 40
B.2.1 Property Manager 40
B.2.2 Joint Properties 41
B.2.3 Link Properties 42
B.2.4 The built package 43
B.3 How to organize a complicated SolidWorks assembly for export . . . 43
B.3.1 Introduction 44
B.3.2 Assemble all link components into one assembly 44
B.3.3 Home State Mates (a.k.a setting the zero position) 44
B.3.4 Setup Configurations and Display States 44
B.3.5 Skin Parts 45

Chapter 1

Introduction and Goals

1.1 What is Drake?

Drake is a toolbox written by the Robot Locomotion Group at the MIT Computer Science and Artificial Intelligence Lab (CSAIL). It is a collection of tools for analyzing the dynamics of our robots and building control systems for them in MATLAB. It deals with general nonlinear systems (including hybrid systems), but also contains specialized tools for multi-link rigid-body dynamics with contact. You might want to use Drake in your own research in order to, for example:

- Analyze the stability of your systems - e.g., by automatically computing Lyapunov functions for global or regional (region of attraction analysis) using Sums-of-Squares optimization,
- Design nonlinear feedback controllers for complicated (nonlinear, underactued) dynamical systems,
- Perform trajectory and feedback-motion planning for complicated (nonlinear, underactuated) dynamical systems, or
- Compute invariant "funnels" along trajectories (derived from your own motion planning software) for robust motion planning.
- Interface with a fast inverse kinematics library for rigid-body systems with a rich specification of kinematic constraints.

Drake also contains supporting methods for visualization, identification, estimation, and even hardware interfaces; making it our complete robotics software package. It has been used by the Robot Locomotion Group and a number of collaborators, but now we are attempting to open up the code to the broader community.

Drake is implemented using a hierarchy of MATLAB classes which are designed to expose and exploit available structure in input-output dynamical systems. While some algorithms are available for general nonlinear systems, specialized algorithms are available for polynomial dynamical systems, linear dynamical systems, etc.; many of those algorithms operate symbolically on the governing equations. The toolbox does a lot of work behind the scenes to make sure that, for instance, feedback or cascade combinations of polynomial systems remain polynomial. The toolbox also provides a parser that reads Universal Robot Description Format (URDF) files which makes it easy to define and start working with rigid-body dynamical systems. Drake uses the Simulink solvers for simulation of dynamical systems, and connects with a number of external tools (some relevant dependencies are listed below) to facilitate design and analysis.

1.1.1 Relative to Simulink and SimMechanics

Roughly speaking, MATLAB's Simulink provides a very nice interface for describing dynamical systems (as S-Functions), a graphical interface for combining these systems in very nontrivial ways, and a number of powerful solvers for simulating the resulting systems. For simulation analysis, it provides everything we need. However the S-Function abstraction which makes Simulink powerful also hides some of the detailed structure available in the equations governing a dynamical system; for the purposes of control design and analysis I would like to be able to declare that a particular system is governed by analytic equations, or polynomial equations, or even linear equations, and for many of the tools it is important to be able to manipulate these equations symbolically.

You can think of Drake as a layer built on top of the Simulink engine which allows you to defined structured dynamical systems. Every dynamical system in Drake can be simulated using the Simulink engine, but Drake also provides a number of tools for analysis and controller design which take advantage of the system structure. While it is possible to use the Simulink GUI with Drake, the standard workflow makes use of command-line methods which provide a restricted set of tools for combining systems in ways that, whenever possible, preserve the structure in the equations.

Like SimMechanics, Drake provides a number of tools for working specifically with multi-link rigid body systems. In SimMechanics, you can describe the system directly in the GUI whereas in Drake you describe the system in an XML file. SimMechanics has a number of nice features, such as integration with SolidWorks, and almost certainly provides more richness and faster code for simulating complex rigid body systems. Drake on the other hand will provide more sophisticated tools for analysis and design, but likely will never support as many gears, friction models, etc. as SimMechanics.

1.1.2 For controlling real hardware

Drake also has many interfaces which allow it to connect to other components of a robotic / control system. Inputs and outputs of the dynamical systems, or static matlab functions (e.g. for trajectory planning), can be connected directly to network interfaces. We primarily use Lightweight Communications and Marshalling (LCM) [3] to make these connections; support for other protocols (such as ROS[1]) can be added easily or accomplished via an independent network translator. This approach has been used extensively on real hardware experiments at MIT, and formed the foundation of our solution for controlling a complex humanoid for MIT's entry into the DARPA Robotics Challenge (with Drake nodes running in desktop MATLAB instances inside the real-time feedback loops)[2]. In addition to the primary MATLAB front end, a number of methods for kinematics and dynamics of rigid-body systems are also available directly as a C++ library.

1.2 Getting Started

To download and install Drake, please follow the "QuickStart" instructions available at

`http://drake.mit.edu/documentation`

This page also contains links to detailed documentation of the methods and classes in Drake (autogenerated using Doxygen), and additional information including a FAQ and an online discussion forum. Please let us know about any issues you have, and of course about any success stories!

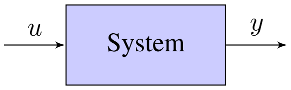
Chapter 2

Modeling and Simulation

The fundamental object in Drake is a dynamical system. Robots, controllers, state estimators, etc. are all instances of dynamical systems. Algorithms in Drake operate on dynamical systems. This chapter introduces what you need to know to instantiate the dynamical systems that you are interested in, and to simulate and visualize their outputs.

2.1 Modeling Input-Ouput Dynamical Systems

Dynamical systems in Drake are represented by their dynamics in state-space form, with x denoting the state vector. In addition, every dynamical system can have an input vector, u, and an output vector y.



As we will see, dynamical systems in Drake can be instantiated in a number of different ways. A system can be described by a block of C code (see for instance section 2.1.2), but many of the algorithms implemented in Drake operate symbolically on the governing equations of the dynamical systems. For this reason, the preferred approach is to use the Drake URDF interface (for rigid body dynamics, see Chapter 8) or alternatively to describe your dynamics in MATLAB code by deriving from the `DrakeSystem` interface class.

2.1.1 Writing your own dynamics

In this section, we will describe how you can write your own dynamics class by deriving from the `DrakeSystem` interface class.

Continuous-Time Systems

Consider a basic continuous-time nonlinear input-output dynamical system described by the following state-space equations:

$$\begin{aligned} \dot{x} &= f(t,x,u), \\ y &= g(t,x,u). \end{aligned}$$

In Drake, you can instantiate a system of this form where $f()$ and $g()$ are anything that you can write into a MATLAB function. This is accomplished by deriving from the `DrakeSystem` class, defining the size of the input, state, and output vectors in the constructor, and overloading the `dynamics` and `output` methods, as illustrated by the following example:

Example 1 (A simple continuous-time system) *Consider the system*

$$\begin{aligned} \dot{x} &= -x + x^3, \\ y &= x. \end{aligned}$$

This system has zero inputs, one (continuous) state variable, and one output. It can be implemented in Drake using the following code:

```
classdef SimpleCTExample < DrakeSystem
  methods
    function obj = SimpleCTExample()
      % call the parent class constructor:
      obj = obj@DrakeSystem(...
         1, ... % number of continuous states
         0, ... % number of discrete states
         0, ... % number of inputs
         1, ... % number of outputs
         false, ... % because the output does not depend on u
         true);  % because the dynamics and output do not depend on t
    end
    function xdot = dynamics(obj,t,x,u)
      xdot = -x+x^3;
    end
    function y=output(obj,t,x,u)
      y=x;
    end
  end
end
```

Discrete-Time Systems

Implementing a basic discrete-time system in Drake is very analogous to implementing a continuous-time system. The discrete-time system given by:

$$\begin{aligned} x[n+1] &= f(n,x,u), \\ y[n] &= g(n,x,u), \end{aligned}$$

can be implemented by deriving from `DrakeSystem` and defining the `update` and `output` methods, as seen in the following example.

Example 2 (A simple discrete-time system) *Consider the system*

$$x[n+1] = x^3[n],$$
$$y[n] = x[n].$$

This system has zero inputs, one (discrete) state variable, and one output. It can be implemented in Drake using the following code:

```
classdef SimpleDTExample < DrakeSystem
  methods
    function obj = SimpleDTExample()
      % call the parent class constructor:
      obj = obj@DrakeSystem(...
         0, ... % number of continuous states
         1, ... % number of discrete states
         0, ... % number of inputs
         1, ... % number of outputs
         false, ... % because the output does not depend on u
         true);  % because the update and output do not depend on t
    end
    function xnext = update(obj,t,x,u)
      xnext = x^3;
    end
    function y=output(obj,t,x,u)
      y=x;
    end
  end
end
```

Mixed Discrete and Continous Dynamics

It is also possible to implement systems that have both continuous dynamics and discrete dynamics. There are two subtleties that must be addressed. First, we'll denote the discrete states as x_d and the continuous states as x_c, and the entire state vector $x = [x_d^T, x_c^T]^T$. Second, we must define the timing of the discrete dynamics update relative to the continuous time variable t; we'll denote this period with Δ_t. Then a mixed system can be written as:

$$\dot{x}_c = f_c(t,x,u),$$
$$x_d(t+t') = f_d(t,x,u), \quad \forall t \in \{0, \Delta_t, 2\Delta_t, ...\}, \forall t' \in (0, \Delta_t]$$
$$y = g(t,x,u).$$

Note that, unlike the purley discrete time example, the solution of the discrete time variable is defined for all t. To implement this, derive from `DrakeSystem` and implement the `dynamics`, `update`, `output` methods for $f_c()$, $f_d()$, and $g()$, respectively. To define the timing, you must also implement the `getSampleTime` method. Type `help DrakeSystem/getSampleTime` at the MATLAB prompt for more information. Note that currently Drake only supports a single DT sample time.

Example 3 (A mixed discrete- and continuous-time example) *Consider the system*

$$x_1(t+t') = x_1^3(t), \quad \forall t \in \{0,1,2,..\}, \forall t' \in (0,1],$$
$$\dot{x}_2 = -x_2(t) + x_2^3(t),$$

which is the combination of the previous two examples into a single system. It can be implemented in Drake using the following code:

```
classdef SimpleMixedCTDTExample < DrakeSystem
  methods
    function obj = SimpleMixedCTDTExample()
      % call the parent class constructor:
      obj = obj@DrakeSystem(...
         1, ... % number of continuous states
         1, ... % number of discrete states
         0, ... % number of inputs
         2, ... % number of outputs
         false, ... % because the output does not depend on u
         true);  % because the update and output do not depend on t
    end
    function ts = getSampleTime(obj)
      ts = [[0;0], ...   % continuous and discrete sample times
        [1;0]];         % with dt = 1
    end
    function xdnext = update(obj,t,x,u)
      xdnext = x(1)^3;       % the DT state is x(1)
    end
    function xcdot = dynamics(obj,t,x,u);
      xcdot = -x(2)+x(2)^3;  % the CT state is x(2)
    end
    function y=output(obj,t,x,u)
      y=x;
    end
  end
end
```

Systems w/ Constraints

Nonlinear input-output systems with constraints can also be defined. There are two distinct types of constraints supported: state constraints that can be modeled as a function $\phi(x) = 0$ and input constraints which can be modeled as $u_{min} \le u \le u_{max}$. For instance, we would write a continuous-time system with state and input constraints as:

$$\dot{x} = f(t,x,u), \quad y = g(t,x,u),$$
$$\text{subject to } \phi(x) = 0, u_{min} \le u \le u_{max}.$$

These two types of constraints are handled quite differently.

Input constraints are designed to act in the same way that an actuator limit might work for a mechanical system. These act as a saturation nonlinearity system attached to the input, where for each element:

$$y_i = \begin{cases} u_{max,i} & \text{if } u_i > u_{max,i} \\ u_{min,i} & \text{if } u_i < u_{min,i} \\ u_i & \text{otherwise.} \end{cases}$$

The advantage of using the input limits instead of implementing the saturation in your own code is that the discontinuity associated with hitting or leaving a saturation is communicated to the solver correctly, allowing for more efficient and accurate simulations. Input constraints are set by calling the `setInputLimits` method.

State constraints are additional information that you are providing to the solver and analysis routines. They should be read as "this dynamical system will only ever visit states described by $\phi(x) = 0$". Evaluating the dynamics at a vector x for which $\phi(x) \neq 0$ may lead to an undefined or non-sensible output. Telling Drake about this constraint will allow it to select initial conditions which satisfy the constraint, simulate the system more accurately (with the constraint imposed), and restrict attention during analysis to the manifold defined by the constraints. However, *the state constraints function should not be used to enforce an additional constraint that is not already imposed on the system by the governing equations.*. The state constraint should be simply announcing a property that the system already has, if simulated accurately. Examples might include a passive system with a known total energy, or a four-bar linkage in a rigid body whos dynamics are written as a kinematic tree + constraint forces. State constraints are implemented by overloading the `stateConstraints` method *and* by calling `setNumStateConstraints` to tell the solver what to expect in that method.

implement example of passive pendulum with and without state constraints, showing the additional accuracy.

Event-Driven Systems

Drake supports systems that are modeled as smooth, discrete- or continuous- time systems which transition between discrete modes based on some event. Simulink calls these models "State Machines". Examples include a walking robot model which undergoes a discrete impulsive collision event (and possibly a change to a different model) when a foot hits the ground, or the switching controller for swinging up the underactuated double pendulum (Acrobot) which switches from an energy-shaping swing-up controller to a linear balancing controller at the moment when the state arrives in a prespecified neighborhood around the upright. An example event-driven system is illustrated in Figure 2.1. Note that the internal mode dynamics could also have discrete-time dynamics or mixed discrete- and continuous-time dynamics.

Event-driven systems in Drake are described using the language from the Hybrid Systems community. Transitions between individual modes are described by a *guard* function, denoted by $\phi(t, x, u)$ in Figure 2.1, which triggers a transition out of the current mode when $\phi \leq 0$. The dynamics of the transition are given by the *reset* function, $x^+ = \Delta(t, x^-, u)$. Event-driven systems are constructed by creating (or inheriting from) an empty `HybridDrakeSystem`, then populating the system with modes (nodes in the graph) by calling `addMode`, and populating the system with transitions (edges in the graph) by calling `addTransition`. Note that it is often useful to create guard

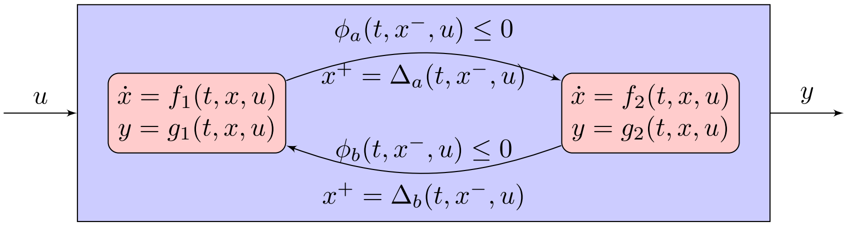


Figure 2.1: Example of a continuous-time event-driven system.

out of a logical combination of smooth guards (e.g. $x(1) > 0$ and $x(2) < .5$); to accomplish this you should use the `andGuard` and `notGuard` methods. The output of a `HybridDrakeSystem` is the output of the active mode.

Example 4 (The bouncing ball: an event-driven system example) *The dynamics of a vertically bouncing ball can be described by a* `HybridDrakeSystem` *with a single continuous mode to model the flight of the ball through the air and a discrete collision event at the moment that the ball hit the ground. This can be accomplished with the following two classes:*

```
classdef BallFlightPhasePlant < DrakeSystem

  methods
    function obj = BallFlightPhasePlant()
      obj = obj@DrakeSystem(...
        2, ... % number of continuous states
        0, ... % number of discrete states
        0, ... % number of inputs
        1, ... % number of outputs
        false, ... % not direct feedthrough
        true); % time invariant
    end

    function xdot = dynamics(obj,t,x,u)
      xdot = [x(2); -obj.g];  % qddot = -g; x=[q,qdot]
    end

    function y = output(obj,t,x,u)
      y = x(1); % height of the ball
    end
  end

  properties
    g = 9.81;  % gravity
  end
end
```

```
classdef BallPlant < HybridDrakeSystem
```

```
  methods
    function obj = BallPlant()
      obj = obj@HybridDrakeSystem(...
        0, ...  % number of inputs
        1);     % number of outputs

      % create flight mode system
      sys = BallFlightPhasePlant();
      obj = setInputFrame(obj,sys.getInputFrame);
      obj = setOutputFrame(obj,sys.getOutputFrame);
      [obj,flight_mode] = addMode(obj,sys);  % add the single mode

      g1=inline('x(1)-obj.r','obj','t','x','u');  % q-r<=0
      g2=inline('x(2)','obj','t','x','u'); % qdot<=0
      obj = addTransition(obj, ...
        flight_mode, ...           % from mode
        andGuards(obj,g1,g2), ...  % q-r<=0 & qdot<=0
        @collisionDynamics, ...    % transition method
        false, ...                 % not direct feedthrough
        true);                     % time invariant
    end

    function [xn,m,status] = collisionDynamics(obj,m,t,x,u)
      xn = [x(1); -obj.cor*x(2)];      % qdot = -cor*qdot

      if (xn(2)<0.01) status = 1; % stop simulating if ball has stopped
      else status = 0; end
    end

  end

  properties
    r = 1;  % radius of the ball
    cor = .8;  % coefficient of restitution
  end
end
```

Stochastic Systems

Drake also provides limited support for working with stochastic systems. This includes continuous-time stochastic models of the form

$$\begin{aligned} \dot{x}(t) &= f(t, x(t), u(t), w(t)) \\ y(t) &= g(t, x(t), u(t), w(t)), \end{aligned}$$

where $w(t)$ is the vector output of a random process which generates Gaussian white noise. It also supports discrete-time models of the form

$$\begin{aligned} x[n+1] &= f(n, x[n], u[n], w[n]) \\ y[n] &= g(n, x[n], u[n], w[n]), \end{aligned}$$

where $w[n]$ is Gaussian i.i.d. noise, and mixed continuous- and discrete-time systems analagous to the ones described in Section 2.1.1. These are quite general models, since nearly any distribution can be approximated by a white noise input into a nonlinear dynamical system. Note that for simulation purposes, any continuous-time white noise, $w(t)$, is approximated by a band-limited white noise signal.

Stochastic models can be implemented in Drake by deriving from `StochastiDrakeSystem` and implementing the `stochasticDynamics`, `stochasticUpdate`, and `stochasticOutput` methods.

Example 5 (A simple continuous-time stochastic system) *Consider the system*

$$\begin{aligned} \dot{x} &= -x + w, \\ y &= x. \end{aligned}$$

This system has zero inputs, one (continuous) state variable, and one output. It can be implemented in Drake using the following code:

```
classdef LinearGaussianExample < StochasticDrakeSystem

  methods
     function obj = LinearGaussianExample
       obj = obj@StochasticDrakeSystem(...
          1, ... % number of continuous states
          0, ... % number of discrete states
          0, ... % number of inputs
          1, ... % number of outputs
          false, ...  % not direct feedthrough
          true, ...   % time invariant
          1, ... % number of noise inputs
          .01);  % time constant of w(t)
     end

     function xcdot = stochasticDynamics(obj,t,x,u,w)
       xcdot = -x + w;
     end

     function y = stochasticOutput(obj,t,x,u,w);
       y=x;
     end
   end
end
```

Important Special Cases

There are many special cases of dynamical systems with structure in the dynamics which can be exploited by our algorithms. Special cases of dynamical systems implemented in Drake include

- Second-order systems, given by $\ddot{q} = f(t, q, \dot{q}, u)$, $y = g(t, q, \dot{q}, u)$, and $\phi_1(q) = 0$ and $\phi_2(q, \dot{q}) = 0$.

- Rigid-body systems, governed by the manipulator equations,

$$H(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = Bu + \frac{\partial \phi_1}{\partial q}^T \lambda_1 + \frac{\partial \phi_2}{\partial \dot{q}}^T \lambda_2$$
$$\phi_1(q) = 0, \quad \phi_2(q,\dot{q}) = 0$$

  where λ_1 and λ_2 are forces defined implicitly by the constraints.

- (Rational) polynomial systems, given by $e(x)\dot{x} = f(t,x,u)$, $y = g(t,x,u)$, subject to $\phi(x) = 0$, where $e(), f(), g()$, and $\phi()$ are all polynomial.

also discrete time

- Linear time-invariant systems, given by

$$\dot{x}_c = A_c x + B_c u,$$
$$x_d[n+1] = A_d x + B_d u,$$
$$y = Cx + Du,$$

  and $\phi(x) = \{\}$.

These special cases are implemented by classes derived from `DrakeSystem`. You should always attempt to derive from the deepest class in the hierarchy that describes your system.

In some cases it is possible to convert between these derived classes. For example, it is possible to convert a rigid-body system to a rational polynomial system (e.g., by changing coordinates through a stereographic projection). Methods which implement these conversions are provided whenever possible.

xcubed example again, but this time deriving from polynomialsystem

describe user gradients

2.1.2 Existing Simulink Models/Blocks

Although most users of Drake never open a Simulink GUI, Drake is built on top of the MATLAB Simulink engine. As such, Drake systems can be used in any Simulink model, and any existing Simulink block or Simulink model (an entire Simulink diagram) which has a single (vector) input and single (vector) output can be used with the Drake infrastructure.

Example of using a simulink model

2.2 Combinations of Systems

Drake actually currently zaps the input on a feedback system. consider changing that behavior; otherwise update this diagram

Whenever possible, structure in the equations is preserved on combination. A polynomial system that is feedback combined with another polynomial system produces a new polynomial system. However, if a polynomial system is feedback combined with a Simulink Block, then the new system is a DynamicalSystem, but not a PolynomialSystem.

A combination of two systems should return a system of the type that is the least common ancestor in the class hierarchy. There are two exceptions: combinations with a hybrid system stay hybrid, and combinations with a stochastic system stay stochastic.

Stochastic Hybrid Systems are not implemented yet, but it wouldn't be hard.

Make sure that feedback and cascade handle all of the cases described above (especially blocks with different sample times, hybrid systems, etc)

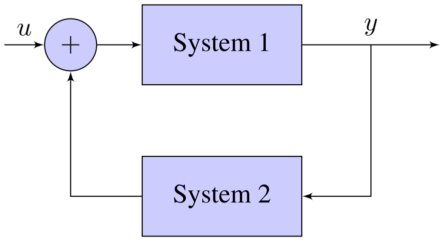


Figure 2.2: Feedback combination

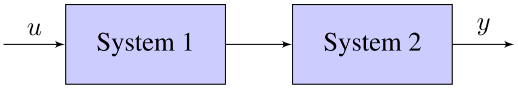


Figure 2.3: Cascade combination

2.2.1 Coordinate frames

Must match to allow combination. Hybrid modes automatically inherit the input and output coordinate frame of the hybrid system; the hybrid system does not have a coordinate frame for the state.

2.3 Simulation

Once you have acquired a DynamicalSystem object describing the dynamics of interest, the most basic thing that you can do is to simulate it. This is accomplished with the `simulate` method in the DynamicalSystem class, which takes the timespan of the simulation as a 1x2 vector of the form `[t0 tf]` and optionally a vector initial condition as input. Type `help DynamicalSystem/simulate` for further documentation about simulation options.

Every `simulate` method outputs a instance of the `Trajectory` class. To inspect the output, you may want to evaluate the trajectory at any snapshot in time using `eval`, plot the output using `fnplt`, or hand the trajectory to a visualizer (described in Section 2.4).

decide once and for all if this is the state trajectory or the output trajectory. it should be the output trajectory, with the state trajectory available as an optional extra output.

Example 6 *Use the following code to instantiate the `SimpleCTExample`, simulate it, and plot the results:*

```
>> sys = SimpleCTExample;
>> traj = simulate(sys, [0 10], .99);
>> fnplt(traj);
```

The arguments passed to `simulate` set the initial time to $t_0 = 0$, final time to $t_f = 5$, and initial condition to $x_0 = .99$. The code looks the same for the `SimpleDTExample`:

```
>> sys = SimpleDTExample;
>> traj = simulate(sys, [0 10], .99);
>> fnplt(traj);
```

These generate the plots in Figure 2.4

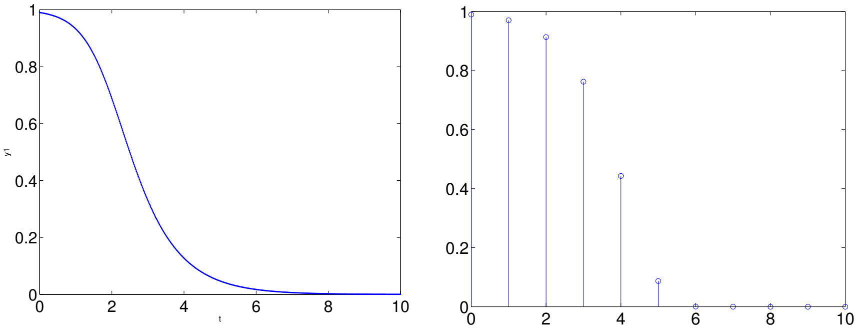


Figure 2.4: Simulation of `SimpleCTSystem` on the left, and `SimpleDTSystem` on the right.

The `simulate` method sets the input, u, to the system to zero. In order to simulate with an input tape, you should `cascade` a Trajectory object describing $u(t)$ with the system, then simulate. If you do not specify the initial conditions on your call to `simulate`, then the `getInitialState()` method is called on your DynamicalSystem object. The default `getInitialState` method sets $x_0 = 0$; you should consider overloading this method in your DynamicalSystem class. In special cases, you may need to set initial conditions based on the initial time or input - in this case you should overload the method `getInitialStateWInput`.

By default, Drake uses the Simulink backend for simulation, and makes use of a number of Simulink's advanced features. For example, input limits can cause discontinuities in the derivative of a continuous time system, which can potenially lead to inaccuracy and inefficiency in simulation with variable-step solvers; Drake avoids this by registering "zero-crossings" for each input saturation which allow the solver to handle the derivative discontinuity event explicitly, without reducing the step-size to achieve the accuracy tolerance. You can change the Simulink solver parameters using the `setSimulinkParam` method. For `DrakeSystems`, you can optionally use the MATLAB's `ode45` suite of solvers using the method `simulateODE`, which can also understand (most of) the Simulink solver parameters.

2.4 Visualization

Playback

Use ball bouncing as an example.

Cascading. Use the realtime block.

2.4.1 Outputing to a movie format

Chapter 3

Analysis

3.1 Fixed Points

3.1.1 Local Stability

3.1.2 Global Stability

3.1.3 Regions of Attraction

3.2 Limit Cycles

3.2.1 Local Stability

3.2.2 Regions of Attraction

3.3 Trajectories

3.3.1 Finite-time invariant regions

3.4 Stochastic Verification

Chapter 4

Planning

4.1 Trajectory Optimization

4.2 Randomized Motion Planning

Chapter 5

Feedback Design

5.1 Linear Quadratic Regulators

5.2 Robust Control

5.3 Sums-of-Squares Design

Chapter 6

System Identification

Coming soon.

Chapter 7

State Estimation

Coming soon.

Chapter 8

Rigid Body Systems

Overview of specific methods for kinematic chains.

8.1 Forward Kinematics

8.2 Inverse Kinematics

8.3 Kinematic Planners

8.4 Contact Dynamics

Chapter 9

External Interfaces

This chapter will explain how to interface with real robots. Coming soon.

Appendix A

For Software Developers

A.1 Code Style Guide

This section defines a style guide which should be followed by all code that is written in Drake. Being consistent with this style will make the code easier to read, debug, and maintain. The section was inspired by the C++ style guide for ROS: `http://www.ros.org/wiki/CppStyleGuide`. It makes use of the follow shortcuts for naming schemes:

- `CamelCased`: The name starts with a capital letter, and has a capital letter for each new word, with no underscores.
- `camelCased`: Like CamelCase, but with a lower-case first letter
- `under_scored`: The name uses only lower-case letters, with words separated by underscores.
- `Under_scored`: The name starts with a capital letter, then uses `under_score`.
- `ALL_CAPITALS`: All capital letters, with words separated by underscores.

Note: Some of the files in the repository were written before this style guide. If you find one, rather than trying to change it yourself, log a bug in bugzilla.

- In General:
  - Robot Names are `CamelCased`.
- In Java:
  - Class names (and therefore class filenames/directories) are `CamelCased`
  - Methods names are `camelCased`
  - Variable names are `under_scored`
  - Member variables are `under_scored` with a leading `m_` added

  - Global variables are `under_scored` with a leading `g_` added
  - Constants are `ALL_CAPITALS`
  - Every class and method should have a brief ”javadoc” associated with it.
  - All java classes should be in packages relative to the locomotion svn root, e.g.:
    `package drake.examples.Pendulum;`
    `package robots.compassTripod;`

- In MATLAB:
  - All of the above rules hold, except:
  - Member variables need not start with `m_` since the requirement that they are referenced with obj.var makes the distinction from local variables clear
  - Variable names that describe a matrix (instead of vector/scalar) are `Under_scored`.
  - Calls to MATLAB class member functions use `obj = memberFunc(obj,...)`.
  - All methods begin by checking their inputs (e.g. with `typecheck.m`).

- In C++:
  - All of the above rules still hold.
  - Filenames for `.cpp` and `.h` files which define a single class are `CamelCased`.
  - Filenames for `.cpp` and `.h` files which define a single method are `camelCased`.
  - Filenames for any other `.cpp` and `.h` files are `under_scored`.

- In LCM:
  - LCM types are `under_scored` with a leading `lcmt_` added. If the type is specific to a particular robot, then it begins with `lcmt_robotname_`.
  - Channel names are `under_scored`, and ALWAYS begin with `robotname_`. *Although robotnames are `CamelCased`, their use in LCM channels and types should be all lowercase*
  - Variable names in LCM types follow the rules above.

A.2 Check-In Procedures

This section defines the requirements that must be met before anything is committed to the main branch (`trunk`) of the Drake repository.

A.2.1 Unit tests

Whenever possible, add test files (in any subdirectory test) any code that you have added or modified. These take a little time initially, but can save incredible amounts of time later.

A.2.2 Run all tests

Before committing anything to the repository, the code must pass all of the unit tests. Use the following script to check: `drake/runAllTests.m`

A.2.3 Matlab Reports

There are a number of helpful matlab reports, that can be run using: `Desktop>Current Directory`, then `Action>Reports` (the Action menu is the gear icon)

Before a commit, your code should pass the following reports:

- Contents report
- Help report (with all but the Copyright options checked)

and you should run the M-Link Code Check report to look for useful suggestions.

A.2.4 Contributing Code

If you don't have write permissions to the repository, then please make sure that your changes meet the requirements above, then email a patch to Russ by running

```
svn diff > my_patch.diff
```

in your main Drake directory, then email the diff file.

A.3 Version Number

Version number has format W.X.Y.Z where

- W= major release number
- X = minor release number
- Y = development stage*
- Z = build

* Development stage is one of four values:

- 0 = alpha (buggy, not for use)
- 1 = beta (mostly bug-free, needs more testing)

- 2 = release candidate (rc) (stable)
- 3 = release

Z (build) is optional. This is probably not needed but could just refer to the revision of the repo at the time of snapshot. Numbered versions should be referenced via tags.

Appendix B

SolidWorks to URDF Exporting

The ROS community has provided a plugin to Solidworks that exports a solid model in the Universal Robot Description Format (URDF).

`http://www.ros.org/wiki/sw_urdf_exporter`

The ROS wiki has an excellent tutorial on how to use the Solidworks to URDF plugin:

`http://www.ros.org/wiki/sw_urdf_exporter/Tutorials`

This section is comprised of two tutorials pulled from the above wiki, with Drake-specific recommendations scattered throughout in *italics*. Preceding them (immediately below) is a brief summary of just these specific recommendations:

B.1 Drake-specific information for those already familiar with the export tool:

- Ensure that each link is in the proper "zeroed" position before running the export tool. Using temporary mates to specify this Home State is recommended.

- Defining your own reference geometry coordinate systems and axes of rotations is highly recommended to ensure you are familiar with each link's X,Y, and Z axes as well as its expected (positive) direction of motion. Axes of rotation should typically have positive values.

- Only the STL files in the meshes folder and the URDF in the robots folder are used by Drake. Manual edits are needed after the export, such as updating the`<visual>` and `<collision>` file paths in the URDF to point to the appropriate places on the file system.

- The STL files generated need to be transformed into .obj files before they can be used by Drake. This can be done with the Meshlab command:

  ```
  meshlabserver -i infile.STL -o outfile.obj -om vn
  ```

  The resulting .obj files should be referenced in the `<visual>` tags of the URDF.

- Multiple parts and assemblies can be included in one link in the URDF. All parts for one link will get welded together, with the resultant meshes and mass/inertia properties as if all included parts were one rigid body.

B.2 Export a SolidWorks Assembly to URDF

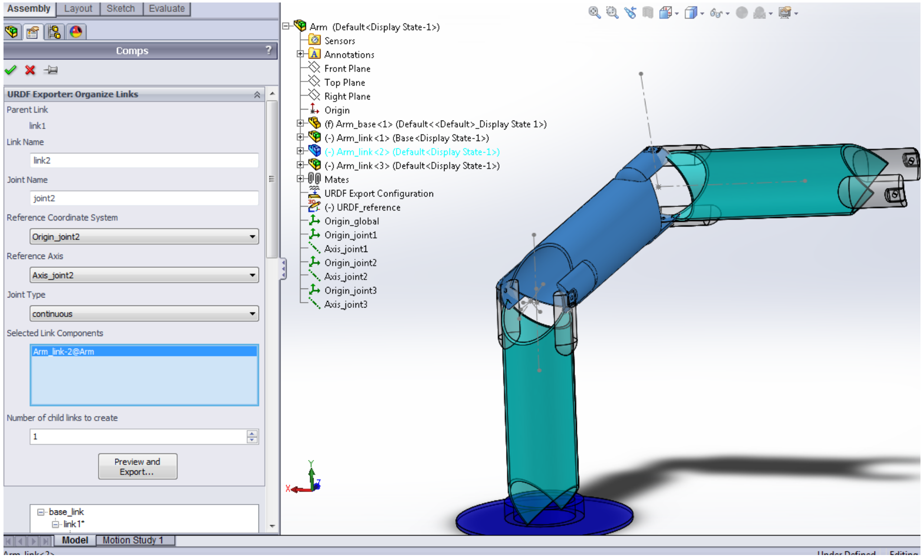
Description: This tutorial covers the process of exporting a SolidWorks Assembly to URDF using the SolidWorks to URDF Exporter

Tutorial Level: BEGINNER
Open your assembly in SolidWorks.

1. Set the position of your joints as you would like them exported
2. Click "File>Export to URDF"

B.2.1 Property Manager

The exporter first brings up a property manager page for you to configure your URDF Export. You will need to configure each link and build the tree manually. After configuring this for the first time, the tool will save your configuration with the assembly itself. You should then be able to skip this step afterwards.

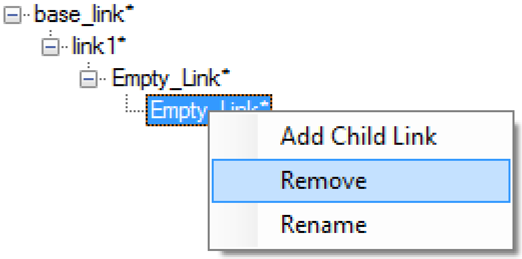

URDF Exporter: Organize Links
Parent Link: link1
Link Name: link2
Joint Name: joint2
Reference Coordinate System: Origin_joint2
Reference Axis: Axis_joint2
Joint Type: continuous
Selected Link Components: Arm_link-2@Arm
Number of child links to create: 1

Assembly
Layout
Sketch
Evaluate
Comps
Preview and Export...
base_link
link1*
Model
Motion Study 1
Arm (Default<Display State-1>)
Sensors
Annotations
Front Plane
Top Plane
Right Plane
Origin
(f) Arm_base<1> (Default<<Default>_Display State 1>)
(-) Arm_link<1> (Base<Display State-1>)
(-) Arm_link<2> (Default<Display State-1>)
(-) Arm_link<3> (Default<Display State-1>)
Mates
URDF Export Configuration
(-) URDF_reference
Origin_global
Origin_joint1
Axis_joint1
Origin_joint2
Axis_joint2
Origin_joint3
Axis_joint3

For each link, you need to give it a unique name, give it a unique joint name, select the components (assemblies or parts) in SolidWorks that are to be associated with that link and add the necessary number of children. If you have reference coordinate systems or axes you would rather the tool to use, you may select those from the list. You can also manually specify what each joint type is. *Both your custom reference coordinate systems and manual joint types are highly recommended so the user maintains* *control over the orientation of the child axes.* Each URDF has only one base link. The joint configurations are disabled for this one link because there doesn't exist a joint to a higher level link. You only need to name the link (if you don't want to call it base_link), select its components and create its child links.

In SolidWorks, if you select components on the actual assembly display instead of the FeatureManager Design Tree, you will only be able to select parts. More likely you would rather select assemblies that represent the entire link, in which case you will have select them from the Design Tree as pictured above.

You can select multiple parts and assemblies for each resulting link in the URDF. Any parts selected for one link will get welded together, and the resultant meshes and mass/inertia properties will be exported as if all included parts were one rigid body in the configuration they are placed in when the tool is run.



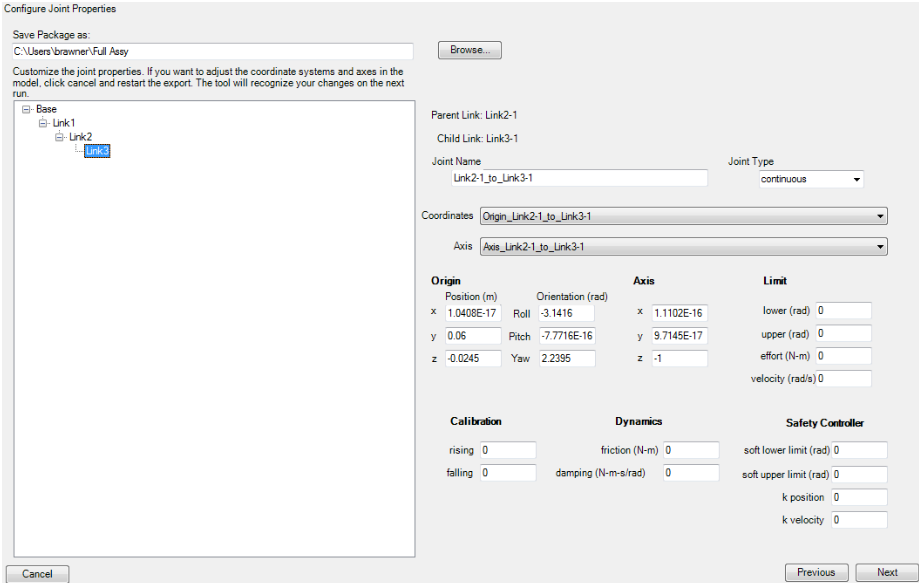
The configuration tree shows you all the links you've added. For each child link on the tree, a joint will be created to its parent link. You can select any link you've already added to change its properties. Right clicking a link will allow you to add children to or remove the link. You can also drag and drop links to re-order them. Dragging a parent link onto a child will cause the child link to switch positions with the parent link. The parent link's other children still stay with the original parent link.

Clicking the green check mark will just save your configuration and exit. Clicking the red x will allow you to exit without saving changes made to your configuration.

When you are ready to build your URDF, click "Preview and Export…". If you have specified the tool to automatically generate coordinate systems or axes, it will build them at this stage.

B.2.2 Joint Properties

Should you find that these joint origins are often not created in the most desirable location, you can change the reference coordinate systems and axes to suit your needs outside of the exporter. However, the model will work fine built as-is. The 3D Sketch is just used for temporary construction, but you may find it useful for adjusting the locations of the reference geometry. Pressing cancel on the Joint Properties page will allow you to save the names of the coordinate systems and axes to your configuration. You may then proceed to adjust the coordinate systems and axes. Restart the export process by clicking "File> Export to URDF". Ensure that the right names of coordinate systems and axes are saved for each link. The tool will no longer build them and instead refer to the reference geometry already in place.



Configure Joint Properties

Save Package as: C:\Users\brawner\Full Assy

Customize the joint properties. If you want to adjust the coordinate systems and axes in the model, click cancel and restart the export. The tool will recognize your changes on the next run.

- Base
  - Link1
    - Link2
      - Link3

Parent Link: Link2-1

Child Link: Link3-1

Joint Name: Link2-1_to_Link3-1

Joint Type: continuous

Coordinates: Origin_Link2-1_to_Link3-1

Axis: Axis_Link2-1_to_Link3-1

Origin

	Position (m)		Orientation (rad)
x	1.0408E-17	Roll	-3.1416
y	0.06	Pitch	-7.7716E-16
z	-0.0245	Yaw	2.2395

Axis

x: 1.1102E-16
y: 9.7145E-17
z: -1

Limit

lower (rad): 0
upper (rad): 0
effort (N-m): 0
velocity (rad/s): 0

Calibration

rising: 0
falling: 0

Dynamics

friction (N-m): 0
damping (N-m-s/rad): 0

Safety Controller

soft lower limit (rad): 0
soft upper limit (rad): 0
k position: 0
k velocity: 0

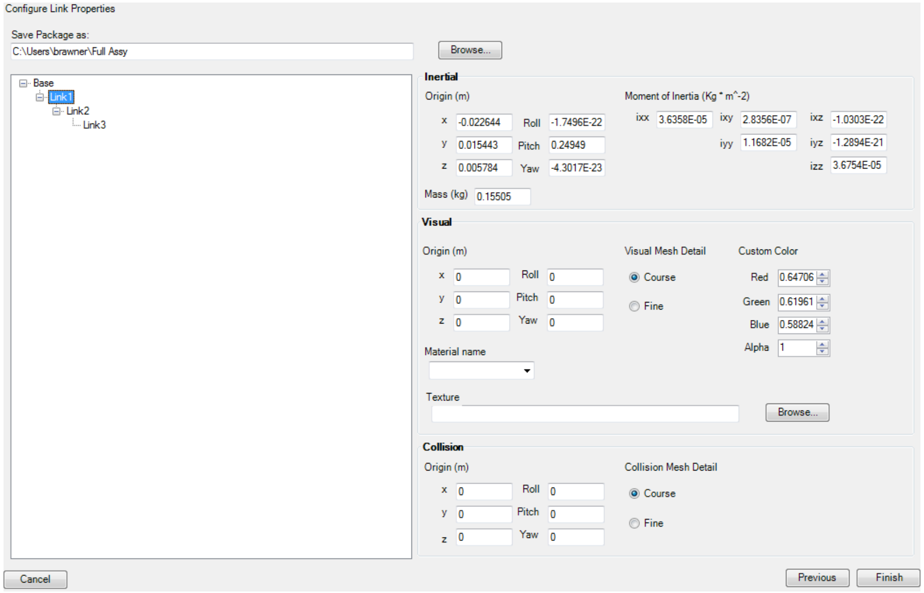
The joint page includes a non-configurable joint tree, where clicking each joint will bring up its properties on the right-hand side. You can customize the properties of any joint before exporting. These properties will not be saved however with the configuration. They will need to be re-entered each time.

Special care should be taken with the sign of the axis of rotation. Typically these should be positive numbers to obey the right-hand rule, especially if you defined your own coordinate reference geometries. Numbers for both positions and axes that are close to zero ($\sim 10^{-15}$*) can be zeroed without consequence.*

Fields that are initially blank are not required by the URDF specifications. If they are left blank they will not be written to the URDF file. If you change a property of a section that includes other required fields, but neglect to specify them, they will be filled in with 0.

B.2.3 Link Properties

This page presents a similar view to the Part Exporter discussed earlier. You can change the link properties of any link in your tree. You can add a texture, change the color, change the origins of different sections, change the moment of inertia tensor, mass, etc. These changes will also not be saved with your configuration. *Varying colors can be useful for visually identifying separate links or degrees of freedom later on. If you input the mass properties into your Solidworks files, it is recommended to leave all the inertial properties alone.*



Save Package as: C:\Users\brawner\Full Assy

Inertial

Origin (m)

x -0.022644 Roll -1.7496E-22
y 0.015443 Pitch 0.24949
z 0.005784 Yaw -4.3017E-23

Moment of Inertia (Kg * m^-2)

ixx 3.6358E-05 ixy 2.8356E-07 ixz -1.0303E-22
iyy 1.1682E-05 iyz -1.2894E-21
izz 3.6754E-05

Mass (kg) 0.15505

Visual

Origin (m)

x 0 Roll 0
y 0 Pitch 0
z 0 Yaw 0

Visual Mesh Detail
● Course
○ Fine

Custom Color
Red 0.64706
Green 0.61961
Blue 0.58824
Alpha 1

Material name

Texture

Collision

Origin (m)

x 0 Roll 0
y 0 Pitch 0
z 0 Yaw 0

Collision Mesh Detail
● Course
○ Fine

Click Finish to create your URDF package.

B.2.4 The built package

The package will contain directories for meshes, textures and robots. *For Drake, you only need the .STL files in the meshes folder and the URDF in the robots folder. These files require a little bit of manual work before they can be used by Drake. The .STL files need to be converted into .obj files using a program like Meshlab. After conversion, the `<visual>` tags in the URDF need to be changed from their ROS declaration to the appropriate .obj filenames.*

Also it is a good idea to simplify the `<collision>` geometry using primitives rather than the default meshes. Refer to the URDF documentation for the different options available for collision references. If you find that your meshes are too complicated you can refer to section B.3 or you can simplify complicated meshes by using a program like MeshLab or Blender.

Except where otherwise noted, the ROS wiki is licensed under Creative Commons Attribution 3.0.

B.3 How to organize a complicated SolidWorks assembly for export

Description: This tutorial covers how to organize your SolidWorks assembly to create a better workflow and also how to reduce the exported mesh complexity when dealing with complicated assemblies.

Tutorial Level: BEGINNER

B.3.1 Introduction

This tutorial is a collection of all techniques I can think of to help make the exporting process simpler.
Goals in simplification:

1. Keep work flow as natural as possible and integrate easily with project
2. Reduce mesh complexity without sacrificing goal #1

B.3.2 Assemble all link components into one assembly

The easiest way to organize your assembly is to have as few components as possible associated with each link. Instead, group as much as possible into a single assembly for every independent link. This suggestion is not due to limitations in the software, but instead it's more likely with a longer list that the associated components will change throughout the development process. So this practice will reduce the amount of configuration necessary before each export. It will also help ensure that the saved export configuration will remain up-to-date for everyone who opens the assembly.

Obviously this won't help if you suddenly decide your robot arm requires 6 degrees of freedom instead of 5, but as you add more components to it it should minimize tinkering around with the configuration.

B.3.3 Home State Mates (a.k.a setting the zero position)

The exporter will use the reference geometry coordinate systems you specify for the resulting URDF model. If these are tied to the child link, then setting the child link in a proper orientation before doing an export is critical to making your zero positions correct. For this case, it is recommended to maintain mates for each degree of freedom. Name these mates something recognizable (like 'Shoulder Home State') so that it is a simple manner of suppressing them for the first export process. Once the reference geometries for the joints have been created and the joint type has been saved, then you do not need to suppress the mates each time. The tool will refer to its configuration instead of trying to automatically generate them, which would be inhibited anyway by the unsuppressed home state mates. However, you should ensure the coordinate systems and axes are located properly, and their names are saved within the configuration.

B.3.4 Setup Configurations and Display States

To avoid exporting the meshes of every single nut, bolt, screw, or flux capacitor in your assembly, you can hide them before exporting. However, solely hiding them doesn't allow you to save this configuration for later exports. Create a Display State specifically for exporting. Click the configuration tab in the Property Manager and right click the bottom section where Display States are located. Add a new display state and name it something like "URDF Export" so that you and others can return to it later. From the property manager tree, hide all the subassemblies and parts you do not want displayed. Double clicking the original display state, probably 'Display State-1', unhides all the components that are hidden in the URDF Export display state. Most SolidWorks users are familiar with Configurations, but these only allow you to suppress components or features. This is not useful because the exporter will ignore all those components when calculating mass properties and your mates may break. Display States is the analogous feature but for hiding parts. SolidWorks allows you to 'Link Display States to Configuration', but this is not recommended. Despite the ability to link configurations in the main assembly with configurations in subassemblies, the Display States aren't inherited for some reason. So you'll have to bite the bullet and work in the top level assembly to hide all the subassembly components. Annoying I know, and it's SW's bug, not mine. I'm open to suggestions for how to deal with this.

B.3.5 Skin Parts

SolidWorks does not have great tools for reducing the triangle count in a mesh below their 'course' export option, which is never course enough for simulation/collision detection. Therefore when not using the Exporter, many are forced to create their own skin meshes and incorporate them into the URDF. Since our goal is to eliminate work outside of the natural work flow specifically for exporting, it's recommended to create your skin part in SolidWorks. Create a part that envelopes your entire link and that vaguely resembles its shape. Set the material on this skin part to 'Air'. It won't matter, but you might want to change the appearance. Insert it into your main assembly. Place this component over your link in the approximate location and create a 'lock' mate to another component in your assembly. Then activate your 'URDF Export' Display State and hide every component but your skin parts. Then in your default 'Display State-1' or whatever its called, hide the skin parts.

Except where otherwise noted, the ROS wiki is licensed under Creative Commons Attribution 3.0.

Bibliography

[1] The Robot Operating System (ROS). http://www.ros.org.

[2] Team mit for the darpa robotics challenge. http://drc.mit.edu, 2013.

[3] Albert S. Huang, Edwin Olson, and David C. Moore. LCM: Lightweight communications and marshalling. *International Conference on Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ*, pages 4057–4062, October 2010.