Drake
A Nonlinear Dynamics and Control Toolbox

Robot Locomotion Group

May 25, 2012

Contents

Chapter 1

Introduction and Goals

1.1 What is Drake?

1.1.1 Relative to Simulink and SimMechanics

Roughly speaking, MATLAB's 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 More than just MATLAB

1.1.3 As a component in R.O.S.

1.1.4 For controlling real hardware

Chapter 2

Installation and QuickStart

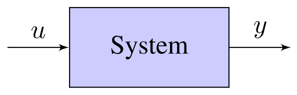
Chapter 3

Modeling and Simulation

The fundamental object in is a dynamical system. Robots, controllers, state estimators, etc. are all instances of dynamical systems. Algorithms in 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.

3.1 Modeling Input-Ouput Dynamical Systems

Dynamical systems in 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 can be instantiated in a number of different ways. A system can be described by a block of C code (see for instance section 3.1.3), but many of the algorithms implemented in operate symbolically on the governing equations of the dynamical systems. For this reason, the preferred approach is (for rigid body dynamics) to use the RobotLib URDF interface or alternatively to describe your dynamics in MATLAB code by deriving from the RobotLibSystem interface class.

3.1.1 Universal Robot Description Format (URDF)

3.1.2 Writing your own dynamics

Not every system of interest can be described by the URDF interface. For example, for some simple model systems, it makes more sense to type in the few lines of MATLAB code to describe the system. In other cases, such as modeling a aircraft, there may be terms required in the dynamics (such as aerodynamic forces) that are not programmed into the rigid-body URDF interface. Similarly, if you need to write your own control system or state estimator, you will need to write your own code. In this section, we will describe how you can write your own dynamics class by deriving from the RobotLibSystem 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 , 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 RobotLibSystem 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 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 is very analogous to implementing a continuous-time system. The discrete-time system given by:

$$x[n+1] = f(n,x,u),$$
$$y[n] = g(n,x,u),$$

can be implemented by deriving from `RobotLibSystem` 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 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 RobotLibSystem 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 RobotLibSystem/getSampleTime` at the MATLAB prompt for more information. Note that currently 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 using the following code:

```
classdef SimpleMixedCTDTExample < RobotLibSystem
  methods
    function obj = SimpleMixedCTDTExample()
      % call the parent class constructor:
      obj = obj@RobotLibSystem(...
         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 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

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 pre-specified neighborhood around the upright. An example event-driven system is illustrated in Figure 3.1. Note that the internal mode dynamics could also have discrete-time dynamics or mixed discrete- and continuous-time dynamics.

Event-driven systems in 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 3.1, which triggers a transition out of the current mode when $\phi \leq 0$. The dynamics of the transition are given by the transition function, $x^+ = \Delta(t, x^-, u)$. Event-driven systems are constructed by creating (or inheriting from) an empty `HybridRobotLibSystem`, then populating the system with modes

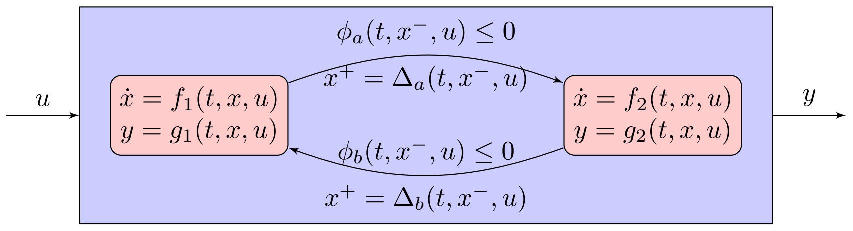


Figure 3.1: Example of a continuous-time event-driven system.

(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 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 `HybridRobotLibSystem` 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* `HybridRobotLibSystem` *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,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

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 3.1.2. 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 by deriving from StochastiRobotLib-System and implementing the `stochasticDynamics`, `stochasticUpdate`, and `stochasticOutput` methods.

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

$$\dot{x} = -x + w,$$
$$y = x.$$

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

```
classdef LinearGaussianExample < StochasticRobotLibSystem

  methods
     function obj = LinearGaussianExample
       obj = obj@StochasticRobotLibSystem(...
          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 RobotLib include

- Smooth systems. Any mixture of continuous and discrete time dynamics, where the functions governing the dynamics $(f(), g(), ...)$ are smooth functions. Smooth functions are functions that are continuous with derivatives that exist everywhere and are smooth[1].
- Piecewise-smooth systems.

- 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.
- URDF. Derive from, but add initial conditions, etc.
- (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 $\dot{x} = Ax + Bu$, $y = Cx + Du$, and $\phi(x) = \{\}$.

also discrete time

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

Some of the algorithms available for simulation and analysis of dynamical systems need to make assumptions about the smoothness of the equations governing the dynamics of the systems they are operating on. For this reason, the methods for implementing dynamical systems are designed to force you to be explicit about when these smoothness assumptions are valid.

In some cases it is possible to convert between these derived classes. For example, converting 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.

Todo: xcubed example again, but this time deriving from polynomialsystem.

Providing User Gradients

3.1.3 Existing Simulink Models/Blocks

Although most users of RobotLib never open a Simulink GUI, is built on top of the MATLAB Simulink engine. As such, 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 infrastructure.

3.1.4 Combinations of Systems

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

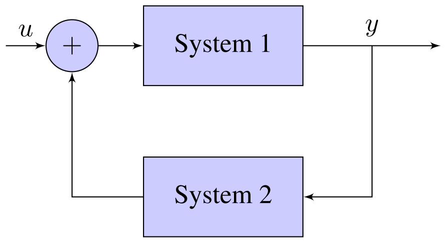


Figure 3.2: Feedback combination

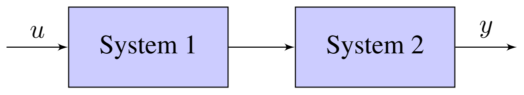


Figure 3.3: Cascade combination

a Simulink Block, then the new system is a DynamicalSystem, but not a Polynomial-System.

A combination of two systems should return a system of the type that is the least common ancestor in the class hierarchy. With 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)

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.

3.2 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 3.3).

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 3.4

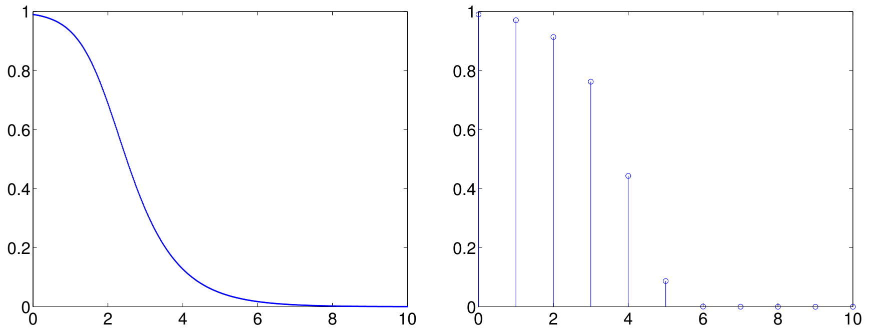


Figure 3.4: Simulation of `SimpleCTSystem` on the left, and `SimpleDTSystem` on the right.

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, 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; 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 SmoothRobotLibSystems, 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.

3.3 Visualization

Playback
Use ball bouncing as an example.
Cascading. Use the realtime block.

3.3.1 Outputing to a movie format

Chapter 4

Analysis

4.1 Fixed Points

4.1.1 Local Stability

4.1.2 Global Stability

4.1.3 Regions of Attraction

4.2 Limit Cycles

4.2.1 Local Stability

4.2.2 Regions of Attraction

4.3 Trajectories

4.3.1 Finite-time invariant regions

Chapter 5

Planning

Chapter 6

Feedback Design

Chapter 7

System Identification

Chapter 8

State Estimation

Chapter 9

External Interfaces

Controlling real robots.

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.

Bibliography

[1] Wikipedia. Smooth function — Wikipedia, the free encyclopedia, 2012. [Online; accessed 15-April-2012].