RobotLib Documentation

Robot Locomotion Group

April 29, 2012

Contents

Chapter 1

Installation and QuickStart

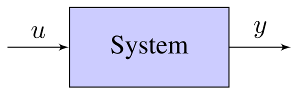
Chapter 2

Modeling and Simulation

The fundamental object in RobotLib is a dynamical system. Robots, controllers, state estimators, etc. are all instances of dynamical systems. Algorithms in RobotLib 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 RobotLib 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 RobotLib can be instantiated in a number of different ways. A system can be described by a block of C code (see for instance section 2.1.3), but many of the algorithms implemented in RobotLib 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.

2.1.1 Universal Robot Description Format (URDF)

2.1.2 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:

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

In RobotLib, 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*

$$\dot{x} = -x + x^3,$$
$$y = x.$$

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

```
classdef SimpleCTExample < RobotLibSystem
  methods
    function obj = SimpleCTExample()
      % call the parent class constructor:
      obj = obj@RobotLibSystem(...
         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 RobotLib 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 RobotLib using the following code:

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

Event-based hybrid systems

Stochastic Systems

Important Special Cases

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

2.1.3 Existing Simulink Models/Blocks

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

2.1.4 Combinations of Systems

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.

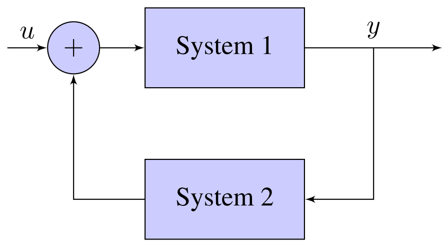


Figure 2.1: Feedback combination

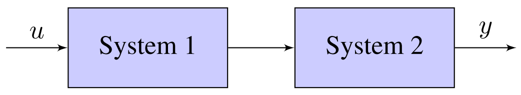


Figure 2.2: Cascade combination

2.2 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 2.3).

Example 4 *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 2.3

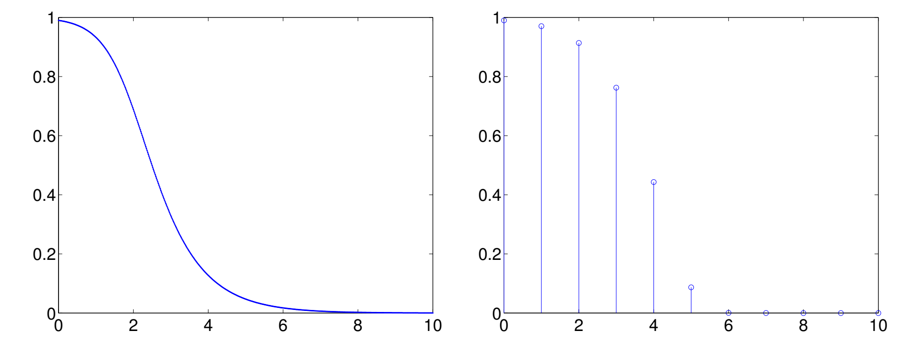


Figure 2.3: Simulation of `SimpleCTSystem` on the left, and `SimpleDTSystem` on the right.

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, RobotLib 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; RobotLib 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. For SmoothRobotLibSystems, you can optionally use the MATLAB's `ode45` suite of solvers using the method `simulateODE`.

2.3 Visualization

Playback

Use ball bouncing as an example.

Cascading. Use the realtime block.

2.3.1 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

Chapter 4

Planning

Chapter 5

Feedback Design

Chapter 6

System Identification

Chapter 7

State Estimation

Chapter 8

External Interfaces

Controlling real robots.

Bibliography

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