MEE5114 Advanced Control for Robotics

Tutorial Lecture
Brief Introduction to Drake

Luyao Zhang and Wei Zhang

SUSTech Insitute of Robotics
Department of Mechanical and Energy Engineering
Southern University of Science and Technology, Shenzhen, China

Outline

- Overview
- Drake Concept
- Drake Simulation
- Common Utilities
- Optimization in Drake

Overview

- Drake is a powerful C++/Python toolbox for robotics started by the Robot Locomotion Group at the MIT and maintained by Toyota Research Institute.
- Include modeling of dynamic systems and simulation
- Provide abundant tools for solving optimization problems, controller design, sensor modeling etc.

Drake Concept

- **System**
  - Matlab Simulink-like style
  - Have input/output ports that could be connected with other systems
  - You can derive your system from **pydrake.systems.framework.LeafSystem**
    Consider the system

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

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

Drake Concept

```python
from pydrake.systems.framework import BasicVector, LeafSystem
# Define the system.
class SimpleContinuousTimeSystem(LeafSystem):
    def __init__(self):
        LeafSystem.__init__(self)
        self.DeclareContinuousState(1) # One state variable.
        self.DeclareVectorOutputPort("y", BasicVector(1),
                                     self.CopyStateOut)
                                     # One output.
    # xdot(t) = -x(t) + x^3(t)
    def DoCalcTimeDerivatives(self, context, derivatives):
        x = context.get_continuous_state_vector().GetAtIndex(0)
        xdot = -x + x**3
        derivatives.get_mutable_vector().SetAtIndex(0, xdot)
    # y = x
    def CopyStateOut(self, context, output):
        x = context.get_continuous_state_vector().CopyToVector()
        output.SetFromVector(x)
```

Drake Concept

- System
  - You can also load the dynamic system from a URDF file.

```
    plant = MultibodyPlant(time_step=1e-4)
    Parser(plant).AddModelFromFile( FindResourceOrThrow(
    "drake/manipulation/models/iiwa_description/sdf/
    iiwa14_no_collision.sdf"))
    plant.WeldFrames(plant.world_frame(), plant.
    GetFrameByName("iiwa_link_0"))
    plant.Finalize()

```

  -

Drake Concept

- Diagram
  - A **Diagram** consists of several smaller **Systems**
  - Use the **DiagramBuilder** class to **AddSystem()s** and to **Connect()** input ports to output ports or to expose them as inputs/output of the diagram
  - Call **Build()** to generate the new **Diagram** instance

```
builder = DiagramBuilder()
# First add the pendulum.
pendulum = builder.AddSystem(PendulumPlant())
pendulum.set_name("pendulum")

controller = builder.AddSystem(PidController(kp=[10.], ki=[1.], kd=[1.]))
controller.set_name("controller")

# Now "wire up" the controller to the plant.
builder.Connect(pendulum.get_state_output_port(), controller.get_input_port_estimated_state())
builder.Connect(controller.get_output_port_control(), pendulum.get_input_port())

```

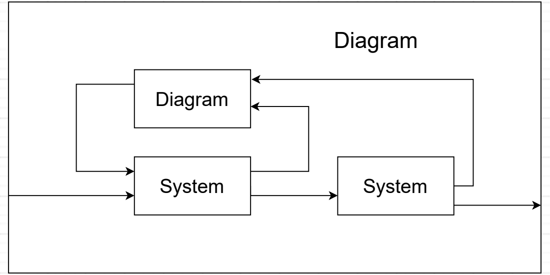
Drake Concept

```
# Make the desired_state input of the controller an input to the diagram.
builder.ExportInput(controller.get_input_port_desired_state())

# Log the state of the pendulum.
logger = LogOutput(pendulum.get_state_outpu t_port(), builder)
logger.set_name("logger")

diagram = builder.Build()
diagram.set_name("diagram")
```

Drake Concept

- **Context**
  - Store data of the time, states, inputs and system parameters, such kinematic model, mass, inertia, CoM position, etc.
  - A **System** or **Diagram** know how to create an instance of a **Context**

```
    context = diagram.CreateDefaultContext()

```



Drake Simulation

- **SceneGraph**
  - Serves as the nexus for all geometry (and geometry-based operations) in a **Diagram**, such as collision checking and visualization
  - The simplest way to add and wire up a **MultibodyPlant** with a **SceneGraph** in your **Diagram** is using **AddMultibodyPlantSceneGraph()**.

```
    plant, scene_graph = AddMultibodyPlantSceneGraph(builder,
     time_step=1e-4)

```

- **Simulator**
  - Run numerical integration for continuous system or state update for discrete system based on the equation of motion and environment forces
  - Write the states back to the **Diagram's** corresponding **Context**

Common Utilities

- Rigid-Body Motions
  - Drake provides the **RotationMatrix** class and the **RigidTransform** class to describe rigid-body motions.

```
    # demonstrate the way to multiply two rotation matrices
    R_Ggrasp0 =
    RotationMatrix.MakeXRotation(np.pi/2.0).multiply(RotationMatrix.MakeZRotation(np.pi/2.0))

    # construct a RigidTransform from
    # a rotation matrix and a position vector
    X_Ggrasp0 = RigidTransform(R_Ggrasp0, p_Ggrasp0)
    # Take the inverse RigidTransform
    X_0Ggrasp = X_Ggrasp0.inverse()
    X_WGgrasp = X_W0.multiply(X_0Ggrasp)

    # convert the rotation matrix to the angle axis
    X_GgraspGpregrasp = RigidTransform([0, -0.08, 0])
    angle_axis = X_GprepickGpreplace.rotation().ToAngleAxis()

```

Common Utilities

- Forward Kinematics

```
    # First, get the link by name,
    # and then evaluate the link pose in the world frame
    B_O = plant.GetBodyByName(link_name, model_instance)
    X_WO = plant.EvalBodyPoseInWorld(plant_context, B_O)

```

Common Utilities

- Jacobian

```
    G = plant.GetBodyByName(link_name).body_frame()
    W = plant.world_frame()
    # Compute the spatial velocity Jacobian of the point Gp
     fixed in the frame G
    # measured in Frame W expressed in Frame W.
    # [0,0,0] is the position vector from Go (frame G's origin)
     to point Gp (regarded as fixed to G), expressed in frame G.
    J_G = self._plant.CalcJacobianSpatialVelocity(
              plant_context, JacobianWrtVariable.kV,
              G, [0,0,0], W, W)

```

$$\mathrm{a} = \mathrm{J}\dot{\mathrm{v}} + \dot{\mathrm{J}}\mathrm{v}$$

- The method CalcBiasSpatialAcceleration() computes $\dot{\mathrm{J}}\mathrm{v}$.

Common Utilities

- Dynamics

$$\mathbf{M(q)\dot{v} + C(q, v)v = \boldsymbol{\tau}_g + B\boldsymbol{\tau} + J_c^T F_{ext}}$$

```
context = plant.CreateDefaultContext()
# update the plant according to
# the current   generalized position and velocity
plant.SetPositions(context,q)
plant.SetVelocities(context,v)
# calculate the mass matrix
M = plant.CalcMassMatrixViaInverseDynamics(context)
# compute the bias term C(q, v)v containing Coriolis,
centripetal, and gyroscopic effects
Cv = plant.CalcBiasTerm(context)
# compute the generalized forces due to the gravity field
tauG = plant.CalcGravityGeneralizedForces(context)
B = plant.MakeActuationMatrix()

```

Common Utilities

- Trajectory Generation
  - The **PiecewisePolynomial** class is used to handle polynomials.
  - You need to specify the break points and the values at the break points.

```
    # times and X_G store a series of break points and gripper poses in the world frame
    traj = PiecewisePolynomial.FirstOrderHold(
            [times["initial"], times["prepick"]],
            np.vstack([X_G["initial"].translation(),
            X_G["prepick"].translation()]).T)

    traj.AppendFirstOrderSegment(times["pick_start"],
                                  X_G["pick"].translation())
    # Quaternion Interpolation
    traj = PiecewiseQuaternionSlerp()
    traj.Append(times["initial"], X_G["initial"].rotation())

```

Optimization in Drake

- Provide an user-friendly interface to formulate and solve optimization problems
- Support the symbolic form for constraints and cost functions
- Categories of optimization problems that Drake can solve
  - Linear programming
  - Quadratic programming
  - Second-order cone programming
  - Nonlinear nonconvex programming
  - Semidefinite programming
  - Sum-of-squares programming
  - Mixed-integer programming (mixed-integer linear programming, mixed-integer quadratic programming, mixed-integer second-order cone programming)
  - Linear complementarity problem

Optimization in Drake

- Code snippet to show the way to formulate and solve the optimization problem

```python
"""
Solves a simple optimization problem
       min x(0)^2 + x(1)^2
subject to x(0) + x(1) = 1
           x(0) <= x(1)
"""
from pydrake.solvers.mathematicalprogram import Solve
# Set up the optimization problem.
prog = MathematicalProgram()
x = prog.NewContinuousVariables(2)
prog.AddConstraint(x[0] + x[1] == 1)
prog.AddConstraint(x[0] <= x[1])
prog.AddCost(x[0] **2 + x[1] ** 2)

```

Optimization in Drake

```
    # Now solve the optimization problem.
    result = Solve(prog)

    # print out the result.
    print("Success? ", result.is_success())
    # Print the solution to the decision variables.
    print('x* = ', result.GetSolution(x))
    # Print the optimal cost.
    print('optimal cost = ', result.get_optimal_cost())
    # Print the name of the solver that was called.
    print('solver is: ', result.get_solver_id().name())

```

The TRI team has provided well-written notebooks, and you can refer to them via the following link.
https://mybinder.org/v2/gh/RobotLocomotion/drake/nightly-release?filepath=tutorials/mathematical_program.ipynb

Optimization in Drake

- Linear Programming
  - The mathematical formulation of a general LP is

$$\min_x \; c^T x + d$$
$$\text{subject to } Ax \leq b$$

  - Add linear cost

```
prog = MathematicalProgram()
# Add two decision variables x[0], x[1].
x = prog.NewContinuousVariables(2, "x")
# Add a symbolic linear expression as the cost.
cost1 = prog.AddLinearCost(x[0] + 3 * x[1] + 2)
# c^T x + d
# We add a linear cost 3 * x[0] + 4 * x[1] + 5 to prog by
    specifying the coefficients
# [3., 4] and the constant 5 in AddLinearCost
cost3 = prog.AddLinearCost([3., 4.], 5., x)
```

Optimization in Drake

- Linear Programming
  - Add linear constraint
    - Bounding box constraint. A lower/upper bound on the decision variable: $lower \leq x \leq upper$.
    - Linear equality constraint: $Ax = b$.
    - Linear inequality constraint: $lower <= Ax <= upper$

```python
prog = MathematicalProgram()
x = prog.NewContinuousVariables(2, "x")
y = prog.NewContinuousVariables(3, "y")
# Add a linear inequality constraint
linear_constraint = prog.AddLinearConstraint(
    A=[[2., 3., 0], [0., 4., 5.]],
    lb=[-np.inf, 1],
    ub=[2., 3.],
    vars=np.hstack((x, y[2])))
# Add a bounding box constraint -1 <= x[0] <= 2, 3 <= x[1] <= 5
bounding_box3 = prog.AddBoundingBoxConstraint([-1, 3], [2, 5], x)
# Add a linear equality constraint 4 * x[0] + 5 * x[1] == 1
linear_eq3 = prog.AddLinearEqualityConstraint(np.array([[4, 5]]), np.array([1]), x)
```

Optimization in Drake

- Quadratic Programming
  - A (convex) quadratic program has the following form

$$\min_x 0.5x^T Qx + b^T x + c$$
$$\text{s.t } Ex \leq f,$$

    where ‘Q‘ is a positive semidefinite matrix.
  - Support different kinds of quadratic cost

```
# Add the cost x[0]*x[0] + x[0]*x[1] + 1.5*x[1]*x[1] + 2*x[0]
    + 4*x[1] + 1
cost3 = prog.AddQuadraticCost(Q=[[2, 1], [1, 3]], b=[2, 4], c
    =1, vars=x)
# Adds the cost (x - [1;2])' * Q * (x-[1;2])
cost4 = prog.AddQuadraticErrorCost(Q=[[1, 2],[2, 6]],
    x_desired=[1,2], vars=x)
# |Ax-b|^2
# Adds the squared norm of (x[0]+2*x[1]-2, x[1] - 3) to the
    program cost.
cost5 = prog.AddL2NormCost(A=[[1, 2], [0, 1]], b=[2, 3], vars
    =x)
```

Optimization in Drake

- Quadratic Programming
  - A (convex) quadratic program has the following form

$$\min_x 0.5x^T Qx + b^T x + c$$
$$\text{s.t } Ex \leq f,$$

    where ‘Q‘ is a positive semidefinite matrix.
  - Drake supports different kinds of quadratic cost.

```
# Add the cost x[0]*x[0] + x[0]*x[1] + 1.5*x[1]*x[1] + 2*x[0] + 4*x[1] + 1
cost3 = prog.AddQuadraticCost(Q=[[2, 1], [1, 3]], b=[2, 4], c=1, vars=x)
# Adds the cost (x - [1;2])' * Q * (x-[1;2])
cost4 = prog.AddQuadraticErrorCost(Q=[[1, 2],[2, 6]], x_desired=[1,2], vars=x)
# |Ax-b|^2
# Adds the squared norm of (x[0]+2*x[1]-2, x[1] - 3) to the program cost.
cost5 = prog.AddL2NormCost(A=[[1, 2], [0, 1]], b=[2, 3], vars=x)
```

  - To add linear constraints into quadratic program, please refer to the previous slide.