CHAPTER 6

Classes

MATLAB provides a framework for object-oriented programming. MATLAB created a new framework in 2008a, although the old one is still available. The new framework will be familiar to those of you who program in Python and C++.

Basically, objects contain both data and the operations that work on the data in one package. Classes conceptually derive from the `struct` which only contains data. Combining data with the code that operates on the data can lead to more reliable software. Once you create a class, you can create new classes that inherit from the old class, without changing the old class adding functionality. A new class can inherit from multiple classes. Subclasses get you out of the habit of adding flags to change the functionality of a block of code and data. The subclass usually adds features that are not available in the original class (i.e., the code/data conglomerate).

In this chapter, we will give an example of a class for state space systems. We will create two subclasses, one that is for a continuous system and one that is for discrete systems. This is in line with the many examples in this book.

6.1 Object-Oriented Programming

Object-oriented programming can be thought of as a method for the software designer to impose restrictions on how the software is used by a programmer. In compiled software, the restrictions are imposed at compile time rather than when the software is executed. This, hopefully, catches many errors. In MATLAB, and other scripting languages, the restrictions are imposed whenever you use the function.

Generally, software has moved from unfettered access to memory to more restrictions. In FORTRAN (MATLAB was originally written in FORTRAN), all variables were pointers. You could pass any variable of any size to a function, and the function only saw the first spot in memory. The old linear algebra package, LAPACK, would have you pass the sizes of each variable. Or you could do all sorts of interesting programming, taking advantage of that feature, much to the detriment of anyone wanting to use your code. MATLAB was designed to solve many of the problems of using LAPACK. A really fun thing you could use was the `COMMON` block. This is perfectly good FORTRAN code.

M. Paluszek and S. Thomas, *MATLAB Recipes*,
https://doi.org/10.1007/978-1-4842-6124-8_6

```
COMMON i1 i2

COMMON x
```

Everything starting with an `i` was an integer. Everything else was float. If you did this, you were mapping `i1` and `i2` into `x`. Confusing?

Problems would arise in practice when multiple people used the same code base and changed `COMMON` blocks without letting other programmers know. Languages like C required all data to be type defined, but even that wasn't enough as software became more complex. Object-oriented programming was devised to catch as many problems as possible at the compile stage.

An object is a conglomeration of data and operations on that data. Classes evolved from structures. A MATLAB structure is

```
s = struct('controlInput',[],'state',[],'stateOutput','data',[])
```

This organizes your variables with names that indicate their purpose. Now your function can take as an input `s`:

```
q = ControlFunction(s);
```

rather than

```
stateOutput = ControlFunction(controlInput,state,data);
```

However, even with the structure, we still have to know that ControlFunction takes `s` as an argument. Object-oriented programming helps in this regard.

A class is a definition of the object. An object is an instantiation of the class. The operations are often called methods. At the very least, we want to be able to add data to the object and read data from the object. So the minimal class is

1. Data
2. Input methods
3. Output methods

The input and output methods control access to the data in the object. You might not want the user of the object to be able to change all of the data, and you might not want a user to have access to all of the data. For example, you might have constants in the object that are fundamental to the functioning of the object that you don't want users to ever change.

After this minimal object definition, you then can add methods that operate on the data in the object. These methods are just functions. This leads to the concept of overloading when a function can have one name but operate on different classes. For example, you could create a member function `Add` for your double class.

```
a = 1;
b = 2;
c = Add(a,b);
```

And then create a member function for your image class.

```
a = imread('a');
b = imread('b');
c = Add(a,b);
```

So you don't need names like `AddDouble` and `AddImage`.

6.2 State Space Systems Base Class

Problem

We want to create a class for state space systems.

Solution

Create the base class for state space systems.

How It Works

A state space system consists of four matrices that connect the system inputs to the system outputs. In between are the states of the systems. The states are dynamical quantities that can change with time. The inputs are external inputs and the outputs are what we see outside. Two versions of a state space system are the continuous and discrete. The continuous system is

$$\dot{x}(t) = ax(t) + bu(t) \tag{6.1}$$
$$y = cx(t) + du(t) \tag{6.2}$$

and the discrete system is

$$x_{k+1} = ax_k + bu_k \tag{6.3}$$
$$y_k = cx_k + du_k \tag{6.4}$$

t is the time and k is the step. A step usually means a value taken at a fixed interval of time. Define a time vector:

```
t = linspace(0,1000,101);
```

If you use this vector, your step is every 10 time units.

Both systems have the vectors x, y, and u and the matrices a, b, c, and d. Our base class will just involve the vectors and matrices along with their names.

If we were using functional programming, as opposed to object-oriented programming, we would create the structure:

```
s = struct('a',[],'b',[],'c',[],'d',[],'x',[],'y',[],'u',[],'xName',{},
    'yName',{},'uName';
```

We'd then create a set of functions to operate on this structure. There are a couple of problems with this approach. The first is that the arrays have specific sizes. If we have n states, m inputs, and p outputs, then a is n by n, b is n by m, c is p by n, and d is p by m. Another problem is that the name fields don't restrict the matrix dimensions which can lead to bugs. Also, there is nothing that says `'xName'` is a cell array. Another issue in MATLAB is that anyone can change the structure on the fly, leading to more issues when sharing software, or even using your old software!

To create a class, select "Class" from the New pull-down in the command window.

```
classdef untitled
  %UNTITLED Summary of this class goes here
  %   Detailed explanation goes here

  properties
    Property1
  end

  methods
    function obj = untitled(inputArg1,inputArg2)
      %UNTITLED Construct an instance of this class
      %   Detailed explanation goes here
      obj.Property1 = inputArg1 + inputArg2;
    end

    function outputArg = method1(obj,inputArg)
      %METHOD1 Summary of this method goes here
      %   Detailed explanation goes here
      outputArg = obj.Property1 + inputArg;
    end
  end
end
```

This provides a good starting framework for the class. There is a method to create an instance of the class, the function that is `untitled`. Internally, you see that `obj` is a data structure. `properties` are the data stored in the class. `methods` are operations that work on the data stored in the class. We create the `StateSpace` class to have only data. It does input

validation so that once you have created the class, all the matrices are the right sizes. The class definition is

StateSpace.m

```
classdef StateSpace
  % StateSpace Dynamical state space class
  %  This class contains the matrices and vectors for a state space
  %  system.
```

The properties, that is, the data, are in the next block of code.

```
  properties (Access = public)
    a(:,:) double
    b(:,:) double
    c(:,:) double
    d(:,:) double
    xN(1,:) cell
    uN(1,:) cell
    yN(1,:) cell
    x(:,:) double
    u(:,:) double
    y(:,:) double
  end

  properties (Access = protected)
    n(1,1) double
    m(1,1) double
    p(1,1) double
  end
```

We made `n,m,p` private to restrict its visibility to subclasses. There are many possible properties. Each set goes with its own block. If it were `private`, subclasses could not see it. You would use `private` if you didn't want people who are deriving subclasses to have access to that property.

We use property validation by specifying

```
    a(:,:) double
    b(:,:) double
    c(:,:) double
    d(:,:) double
    xN(1,:) cell
```

If we don't set the property correctly, we will get

```
>> s.xN = 1
Error setting property 'xN' of class 'StateSpace':
Invalid data type. Value must be cell or be convertible to cell.
```

However, if we do, we will get

```
>> s.a = 'mike'

s =

  StateSpaceDiscrete with properties:

     a: [109 105 107 101]
     b: [2x1 double]
     c: [1 0]
     d: 0
    xN: {'r'  'v'}
    uN: {'u'}
    yN: {'y'}
     x: [2x1 double]
     u: 0
     y: 0
```

It happily makes `a` a 1-by-4 array. This is because `char` is numeric. For example, in the char "Mike", each character is an integer, which is of course numeric. A more sophisticated property validation is possible. You should use as much property validation as you deem necessary for your class.

The remaining code is the class constructor.

```
26      function obj = StateSpace(a,b,c,d,xN,uN,yN)
27        % StateSpace Construct an instance of this class
28        %   Checks all of the sizes
29        obj.a   = a;
30        obj.n   = size(a,1);
31        obj.b   = b;
32        [n,m]   = size(b);
33        if(n ~= obj.n)
34          error('b must have as many rows as a');
35        end
36        obj.m = m;
37
38        [p,n]     = size(c);
39        if(n ~= obj.n)
40          error('c must have as many columns as a');
41        end
42        obj.c   = c;
43        obj.p   = p;
44
45        [p,m]     = size(d);
46        if(p ~= obj.p)
47          error('d must have as many rows as c');
48        end
49        if(m ~= obj.m)
50          error('d must have as many columns as b');
51        end
```

```
52        obj.d   = d;
53
54        if( nargin > 4 )
55          n = length(xN);
56          if( n ~= obj.n )
57            error('xN must have as many strings as the rows of a');
58          end
59          obj.xN  = xN;
60
61          m = length(uN);
62          if( m ~= obj.m )
63            error('uN must have as many strings as the columns of b');
64          end
65          obj.uN  = uN;
66
67          p = length(yN);
68          if( n ~= obj.n )
69            error('yN must have as many strings as the rows of c');
70          end
71          obj.yN  = yN;
72        else
73          for k = 1:obj.n
74            obj.xN{k} = sprintf('%d',k);
75          end
76          for k = 1:obj.p
77            obj.yN{k} = sprintf('%d',k);
78          end
79          for k = 1:obj.m
80            obj.uN{k} = sprintf('%d',k);
81          end
82        end
83
84        obj.x = zeros(n,1);
85        obj.u = zeros(m,1);
86        obj.y = zeros(p,1);
```

These methods are all fully implemented in the code. We add one to compute the eigenvalues since this is common to all state space systems.

```
89      function e = Eig(obj)
90      %Eig Get the eigenvalues
91        e = eig(obj.a);
92      end
```

We can then create a double integrator using our class.

```
>> s = StateSpace([0 1;0 0],[0;1],[1 0],1,{'r' 'v'},{'u'},{'y'})

s =

  StateSpace with properties:
```

```
      a: [2x2 double]
      b: [2x1 double]
      c: [1 0]
      d: 1
     xN: {'r'  'v'}
     uN: {'u'}
     yN: {'y'}
      x: [2x1 double]
      u: 0
      y: 0
```

`n,m,p` are not listed.

The first argument to every member class is `obj`. You don't pass this as an argument; it is implicit in the member function call.

The eigenvalues are

```
>> s.Eig

ans =

     0
     0
```

which is what we expect.

6.3 State Space Systems Discrete Class

Problem

We want to create a class to propagate discrete time state space systems.

Solution

Create a subclass of `StateSpace` and add a step propagator and a general propagator.

How It Works

We make `StateSpaceDiscrete` a subclass of `StateSpace` in the first line with the `>` operator.

StateSpaceDiscrete.m

```
1  classdef StateSpaceDiscrete<StateSpace
```

If you have multiple super classes, list multiple superclasses `superclass1 & superclass2 & superclass3`, for example:

```
1  classdef automobile>RigidBody>GroundVehicle>FourWheels
```

The constructor just passes the inputs to the super class.

```
6       function obj = StateSpaceDiscrete(a,b,c,d,xN,uN,yN)
7         if( nargin == 0 )
8          a  = [];
9          b  = [];
10         c  = [];
11         d  = [];
12         xN = {};
13         yN = {};
14         uN = {};
15        end
16        obj@StateSpace(a,b,c,d,xN,uN,yN);
17      end
```

We add two methods to propagate the discrete time class.

```
19      function y = Propagate(obj)
20        %Propagate Propagates the state space system
21        %   Propagates the state space system
22        n      = size(obj.u,2);
23        y      = zeros(size(obj.x,n));
24        y(1)   = obj.c*obj.x;
25        for k = 2:n
26          y(k)   = obj.c*obj.x + obj.d*obj.u(:,k-1);
27          obj.x = obj.a*obj.x + obj.b*obj.u(:,k-1);
28        end
29      end
30
31      function y = Step(obj,n)
32        %Step Applies a step to the state space system
33        %   Generates the step response. Only the first value of u is
              used.
34        y      = zeros(obj.p,n);
35        y(1)   = obj.c*obj.x + obj.d*obj.u(:,1);
36        for k = 2:n
37          y(k)   = obj.c*obj.x + obj.d*obj.u(:,1);
38          obj.x = obj.a*obj.x + obj.b*obj.u(:,1);
39        end
40      end
```

6.4 Using the State Space Class

Problem

We want to create a script to use the state space class.

Solution

Create a script to propagate the continuous subclass of StateSpace.

How It Works

We create a script to use both propagation methods. We assign values to `u` by using the dot operator, just like any structure. We don't need to write setter methods.

DiscretePropagate.m

```
%% Demonstrate using methods of a subclass.

a     = [0 1;0 0];
b     = [0;1];
dT    = 0.03;

% Convert to discrete time
[a,b] = C2DZOH(a,b,dT);

% Step response
s   = StateSpaceDiscrete(a,b,[1 0],0,{'r' 'v'},{'u'},{'y'});
s.u = 1;
y   = s.Step(100);

PlotSet(1:length(y),y,'x label','Step','y label','y',...
        'figure title','Sub Class Step');

% Pulse Response
s.u = [zeros(1,20) ones(1,30) zeros(1,50)];

y   = s.Propagate;

PlotSet(1:length(y),y,'x label','Step','y label','y',...
       'figure title','Sub Class Pulse');
```

`method` uses whatever `u` is in the object when you used `Propagate`. You can add help using `%` just below the method names and at the top.

```
>> help StateSpace
  StateSpace Dynamical state space class
   This class contains the matrices and vectors for a state space
   system.

    Documentation for StateSpace

>> help StateSpace.Eig
 Eig Get the eigenvalues
```

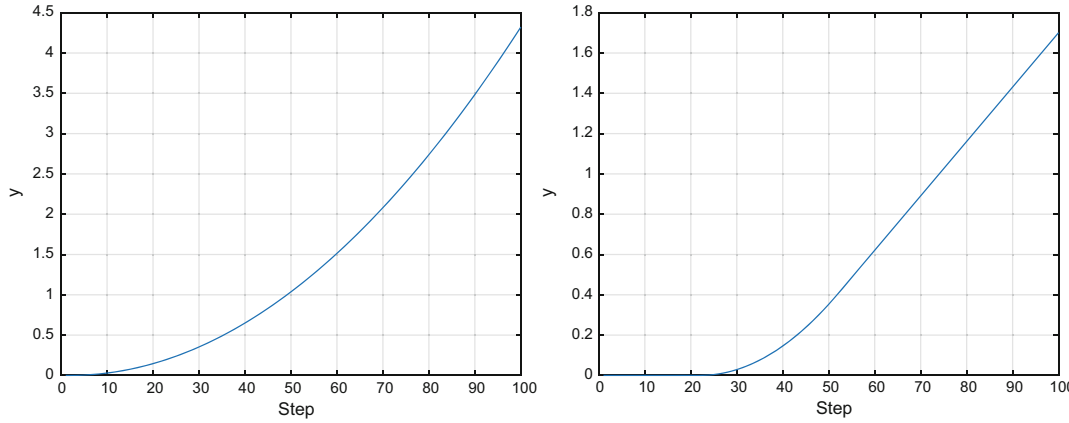

Figure 6.1: *Propagated states. The step response is on the left. The pulse response is on the right.*

The resulting plots are shown in Figure 6.1.

You can see how compact the code is. `StateSpaceDiscrete` can handle any linear time-invariant (i.e., the state space matrices are constant) discrete time state space system.

6.5 Using a Mocking Framework

Problem

We want to test an incomplete class for which other needed classes are unavailable.

Solution

Use a mocking framework which is a framework that allows us to interface to an incomplete class, that is, a "mock" class.

How It Works

We create a class to test a function that calls a class that does not exist or is unavailable. In this case, we have a function `Drag`:

Drag.m

```
function drag = Drag(DensityModel,h,v,s)
drag = 0.5*DensityModel.LookUpDensity(h)*s*v^2;
end
```

that uses the density class `DensityModel`. When you are testing `Drag`, the density table is not yet available. Create the class for `DensityModel` with an abstract method.

DensityModel.m

```
classdef DensityModel

  methods(Abstract,Static)

    rho = LookUpDensity(altitude);

  end

end
```

Now write the test class using the `matlab.unittest.TestCase` and `matlab.mock.TestCase` superclasses. The `mock` framework allows us to fake the existence of the density model. The `unittest` framework allows us to evaluate the results of the test.

DragTest.m

```
classdef DragTest < matlab.unittest.TestCase & matlab.mock.TestCase
    methods(Test)
        function NegativeDensity(testCase)
          [stubDensity,densityModelBehavior]=createMock(testCase,?
              DensityModel);
          testCase.assignOutputsWhen(densityModelBehavior.LookUpDensity
              (-1),-1);
          d = Drag(stubDensity,-1,1,2);
          testCase.verifyLessThan(d,0);
        end
    end
end
```

The question mark, `?`, is used to get a metaclass of `DensityModel`. In this code snippet, the Drag function is called with an altitude of `h= -1`, a velocity of `v=1`, and a surface area of `s=2`. It is given a "stubDensity" class that is created just for the purpose of this test, using the mock framework `createMock`.

We create a mock object, `stubDensity`. The method is implemented with `density ModelBehavior.LookUpDensity(0)`. When we pass a negative altitude, we get a negative density and negative drag.

Run the test.

```
>> results = runtests('DragTest');
Running DragTest
.
Done DragTest
__________

>> table(results)

ans =
```

```
  1x6 table

   Name                              Passed  Failed  Incomplete Duration
       Details
___________________                  ______  _____   __________ ________
    ______
{'DragModelTest/NegativeDensity'} true    false   false      0.04766 {1x1
     struct}
```

This verifies that we get the correct behavior from `Drag`.

Summary

This chapter has demonstrated how to use MATLAB classes and mocking frameworks in classes. Table 6.1 lists the code developed in the chapter.

Table 6.1: *Chapter Code Listing*

File	Description
`StateSpace`	State space dynamical system class
`StateSpaceDiscrete`	Subclass of `StateSpace` for discrete systems
`DragTest`	A test class for `Drag` and `DensityModel`
`Drag`	A test class for `Drag`
`DensityModel`	A placeholder class for `DensityModel`