Appendix

C MATLAB® Objects

The MATLAB® programming language, known as 'M', has syntax and semantics somewhat similar to the classical computer language Fortran. In particular array indices start from one not zero, and subscripts are indicated by parentheses just like function call arguments. In early versions of MATLAB® the only data type was a two-dimensional matrix of real or complex numbers and a scalar was just a 1×1 matrix. This changed with the release of MATLAB® version 5.0 in 1997 which introduced many features that are part of the language today: structures, cells arrays and classes.

The early computer languages (Fortran, Pascal, C) are imperative languages in which the programmer describes computation in terms of *actions* that change the program's state – its data. The program is a logical procedure that takes input data, processes it, and produces output data. As program size and complexity grew the limitations of imperative programming became evident and new languages were designed to address these shortcomings.

A very powerful idea, dating from the mid 1980s, was object-oriented programming (OOP). The OOP programming model is organized around *objects* rather than *actions*. Each object encapsulates data and the functions, known as *methods*, to manipulate that object's data. The inner details of the object need not be known to the programmer using the object. The object presents a clean interface through its methods which makes large software projects easier to manage.

OOP languages support the concept of object classes. For example, we might define a class that represents a quaternion and which has methods to return the inverse of the quaternion, multiply two quaternions or to display a quaternion in a human-readable form. Our program might have a number of quaternion variables, or *objects*, and each is an *instance* of the quaternion class. Each instance has its own value, the data part of the object, but it shares the methods defined for the class.

Well known OOP languages such as C++, Java, Python and Ruby are still imperative in style but have language features to support objects. MATLAB® shares many features with these other well-known OOP languages and the details are provided in the MATLAB® documentation. The Toolboxes define a number of classes to represent robot arms, robot arm links, quaternions, robot path planners and various types of image feature. Toolbox classes are shown in bold font in the index of functions on page 554.

The use of objects provides a solution to the namespace pollution problem that occurs when using many MATLAB® toolboxes. When a MATLAB® function is invoked it is searched for in a list of directories – the MATLAB® search path. If the search path contains lots of Toolboxes from various sources the chances of two functions having the same name increases and this is problematic. If instead of functions we provide methods for objects then those method names don't occupy the function namespace, and can only be invoked in the context of the appropriate object.

Using a Class

The following illustrates some capabiltiies of the quaternion class provided as part of the Robotics Toolbox. A quaternion object is created by

```
>> q = Quaternion( rotx(0.2) );
```

which invokes the *constructor* method for the class. By convention class names begin with a capital letter. This method checks the types of arguments and computes the equivalent quaternion. The quaternion's scalar and vector components are stored within this particular object or *instance* of the quaternion class. In MATLAB® the data part of an object is referred to as its *properties*. The arguments to the constructor can be a rotation matrix (as in this case), an angle and a vector, a 4-vector comprising the scalar and vector parts, or another quaternion. The result is a new object in the workspace

```
>> about(q)
q [Quaternion] : 1x1 (88 bytes)
```

and it has the type `Quaternion`. In a program we can inquire about the type of an object

```
>> class(q)
ans =
  Quaternion
```

which returns a string containing the name of the object's class. All MATLAB® objects have a class

```
>> x = 3
>> class(x)
ans =
  double
```

and this class `double` is built in, unlike `Quaternion` which is user defined. We can test the class of an object

```
>> isa(q, 'double')
ans =
     0
>> isa(q, 'Quaternion')
ans =
     1
```

We can access the properties of the quaternion object by

```
>> q.s
ans =
    0.9950
```

which returns the value of the scalar part of the quaternion. However the Toolbox implementation of the `Quaternion` does not allow this property to be set

```
>> q.s = 0.5;
??? Setting the 's' property of the 'Quaternion' class is not allowed.
```

since the scalar and vector part should be set together to achieve some consistent quaternion value.

We can compute the inverse of the quaternion by

```
>> qi = inv(q);
```

which returns a new quaternion `qi` equal to the inverse of `q`.

MATLAB® checks the type of the first argument and because it is a `Quaternion` it invokes the `inv` method of the `Quaternion` class. Most object-oriented languages use the *dot* notation which would be

```
>> qi = q.inv();
```

which makes it very clear that we are invoking the `inv` method of the object `q`. Either syntax is permissible in MATLAB® but in this book we use the dot notation for clarity. MATLAB® does not require the empty parentheses either, we could write

```
>> qi = q.inv
```

but for consistency with object-oriented practice in other languages, and to avoid confusion with accessing properties, we will always include them.

Any MATLAB® expression without a trailing semicolon will display the value of the expression. For instance

```
>> qi
qi =
0.995 < -0.099833, 0, 0 >
```

causes the `display` method of the quaternion to be invoked. It is exactly the same as typing

```
>> qi.display()
qi =
0.995 < -0.099833, 0, 0 >
```

This in turn invokes the `char` method to convert the quaternion value to a string

```
>> s = qi.char();
>> about(s)
s [char] : 1x25 (50 bytes)
```

We will create another quaternion

```
>> q2 = Quaternion( roty(0.3) );
```

and then compute the product of the two quaternions which we can write concisely as

```
>> q * q2
```

This is an example of operator overloading which is a feature of many object-oriented languages. MATLAB® interprets this as

```
>> q.mtimes(q2)
```

For more complex expressions operator overloading is critical to expressivity, for example we can write

```
>> q*q2*q
ans =
0.96906 < 0.19644, 0.14944, 0 >
```

and MATLAB® does the hardwork of computing the first product `q*q2` into a temporary quaternion, multiplying that by `q` and then deleting the temporary quaternion. To implement this without operator overloading would be the nightmare expression

```
>>  q.mtimes( q2.mtimes(q) )
ans =
0.96906 < 0.19644, 0.14944, 0 >
```

which is both difficult to read and to maintain.

Creating a Class

The quaternion class is defined by the Toolbox file `Quaternion.m` which is over 500 lines long but the basic structure is

```
1   classdef Quaternion
2
3       properties (SetAccess = private)
4           s       % scalar part
5           v       % vector part
6       end
7
8       methods
9
10          function q = Quaternion(a1, a2)
11              % constructor
12          end
13           .
14           % other methods
15           .
16           .
17      end
18  end
```

The `properties` block, lines 3–6, defines the data associated with each quaternion instance, in this case the internal representation is the scalar part in the variable `s` and the vector part in the variable `v`. The methods block, lines 8–17, defines all the methods that the class supports. The name after `classdef` at line 1 must match the name of the file and is the name of the class.

The properties have a `SetAccess` mode `private` which means that the properties can be read directly by programs but not set. If `q` is a quaternion object then `q.s` would be the value of the scalar part of the quaternion. This is a matter of programming style, and some people prefer that all access to object properties is via explicit *getter* functions such as `q.get_s()`.

Every class must have a *constructor* method which is a function with the same name as the class. The constructor is responsible for initialising the data of the object, in this case its properties `s` and `v`. Some object-oriented languages also support a *destructor* function that is invoked when an object is no longer needed, in MATLAB® this is the optional method `delete`.

The quaternion class implements 20 different methods. Each method is written as a MATLAB® function with an `end` statement. The first argument to each method is the quaternion object itself. For example the method that returns the inverse of a quaternion is

```
function qi = inv(q)
    qi = Quaternion( [q.s -q.v] );
end
```

which uses the constructor method `Quaternion` to create the quaternion that it returns.

The method to convert a quaternion to a string is

```
function s = char(q)
    s = [ num2str(q.s), ' < ' num2str(q.v(1)) ...
    ', ' num2str(q.v(2)) ', '   num2str(q.v(3)) ' >' ];
end
```

The method `mtimes` is invoked for operator overloading whenever the operand on either side of an asterisk is a quaternion object.

```
function qp = mtimes(q1, q2)
    if ~isa(q1, 'Quaternion')
        error('left-hand side of * must be a Quaternion');
    end

    if isa(q2, 'Quaternion')
       %Multiply unit-quaternion by unit-quaternion
        s1 = q1.s;  v1 = q1.v;
        s2 = q2.s;  v2 = q2.v;
         qp = Quaternion([s1*s2-v1*v2' s1*v2+s2*v1+cross(v1,v2)]);
    elseif  isa(q2, 'double'),
        if length(q2) == 3
            % Multiply vector by unit-quaternion
            qp = q1 * Quaternion([0 q2(:)']) * inv(q1);
            qp = qp.v(:);
        elseif length(q2) == 1
            % Multiply quaternion by scalar
            qp = Quaternion( double(q1)*q2);
        else
            error('quaternion-vector product: must be a 3-vector
or scalar');
        end
    end
end
```

The method tests the type of the second operand and computes either a quaternion-quaternion, quaternion-vector or quaternion-scalar product.

MATLAB® classes support inheritance. This is a feature whereby a new class can inherit the properties and methods of an existing class and extend that with additional properties or methods. In Part II the various planners such as `Dstar` and `RRT` inherit from the class `Navigation` and in Part IV the different types of camera such as `CentralCamera` and `FishEyeCamera` inherit from the class `Camera`. Inheritance is indicated at the `classdef` line, for example

```
classdef Dstar < Navigation
```

Inhertitance, particularly multiple inheritance, is a complex topic and the MATLAB® documentation should be referred to for the details.

The MATLAB® functions `methods` and `properties` return the methods and properties of an object. The function `metaclass` returns a data structure that includes all methods, properties and parent classes.

Pass by Reference

One particularly useful application of inheritance is to get around the problem of *pass by value*. Whenever a variable is passed to a function MATLAB® passes its value, that is a copy of it, rather than a reference to it. This is normally quite convenient, but consider now the case of some object which has a method that changes a property. If we write

```
>> myobj.set_x(2);
```

then MATLAB® creates a copy of the object `myobj` and invokes the `set_x()` method on the copy. However since we didn't assign the copied object to anything the change is lost. The correct approach is to write this as

```
>> myobj = myobj.set_x(2);
```

which is cumbersome. If however the object `myobj` belongs to a *reference class* then we can write

```
>> myobj.set_x(2);
```

and the value of `myobj` would change. To create a reference class the class must inherit from the `handle` class

```
classdef MyClass < handle
```

A number of classes within the Toolbox, but not the `Quaternion` class, are reference classes. A possible trap with reference classes is that an assignment of a reference class object

```
>> myobj2 = myobj;
```

means that `myobj2` points to the same object as `myobj`. If `myobj` changes then so does `myobj2`. It is good practice for an object constructor to accept an argument of the class type and to return a copy

```
>> myobj2 = MyClass(myobj);
```

so now changes to `myobj` will not effect `myobj2`

Arrays of Objects

MATLAB® handles arrays or vectors of objects in a very familiar way. Consider the example of an array of SIFT feature objects (from page 384)

```
>> s1 = isurf(im1);
```

which returns a vector of `SurfPointFeature` objects. We can determine the number of objects in the vector

```
>> n = length(s1)
n =
   1288
```

or perform indexing operations such as

```
>> x = s1(1:100);
>> y = s1(1:20:end);
```

Note that the `SurfPointFeature` objects are reference objects so the elements of `x` and `y` are the same objects as referred to by `s1`. We can also delete objects from the vector

```
>> s1(50:end) = [];
```

Invoking a method on an object array, for example the hypothetical method

```
>> z = s1.fewer();
```

results in the entire vector being passed to the method

```
function r = fewer(s)
    r = s(1:20:end);
end
```

so methods can perform operations on single objects or arrays of objects.

A class that supports vectors must have a constructor that handles the case of no passed arguments.

Multi-File Implementation

For a complex class a single file might be too long to be workable and it would be preferable to have multiple files, one per method or group of methods. This would certainly be the case if some of the methods were defined as MEX-files rather than M-files.

In MATLAB® this is handled by creating a directory in the MATLAB® search path with an '@' symbol prefix. The `SerialLink` class which represents a robot arm is defined this way, and all its files are within a directory called `@SerialLink`.