Chapter 3

Creating a robot model using C-STORM’s openchain class

This chapter outlines the general procedure used when solving problems with C-STORM. This procedure may require slight modifications for different problems, but the main steps are the same.

There are two Matlab classes which can be used to describe robot mechanisms in C-STORM. The openchain class and the link class. The openchain class can be used to describe unbranched serial open chains, while the link class can be used to describe both unbranched and branched chains. Since either class can be used to model unbranched chains, I should mention that they do provide some slightly different abilities. In general, the openchain class provides slightly faster code execution time while the link class provides more functionality. Don’t worry if you use the openchain class and later decide you want to use the link class, it is fairly easy to change between them.

In this section, I will “build” a simple robot using the openchain class. This is accomplished using a scripted Matlab .m files. For this example, we will construct a simple 3R robot. We will create a file called 3r_robot.m which will contain all the information and function calls needed to construct the robot from the Matlab command line.

3.1 Step 0: Understand Matlab classes

The following steps assume working knowledge of how Matlab classes work. For more information please read the Matlab 5.x user manual. For our purposes, the most
important concept to understand is the idea of a constructor. A constructor is a function which is used to construct an instance of a class. An instance is merely a fancy name for a variable which contains the information for a specific member of the class. For example, let us consider the “class” of integers. When we want to create an instance of this class in a programming language we might type:

```c
int a;
```

```c
a = 3;
```

The variable `a` is an instance of the integer class since it holds specific information (namely the number 3). The “constructor” is the declaration: `int a;`, in which `a` is defined to be an integer.

The syntax for class definitions in Matlab 5.x is different from C++. Unfortunately, Matlab does not contain any built-in classes which are used in the same way as user-defined class. If we imagine a hypothetical, user-defined integer class, it would be called in the following manner:

```c
a = int;
```

```c
a = 3;
```

First the default integer constructor is called, creating `a` as an integer, and then the value of `a` is set to 3. A more streamlined definition in Matlab would be `a = int(3);` in which the value of 3 is passed into the constructor.

C-STORM makes extensive use of Matlab classes. In the next few sections we will present sample Matlab code which calls constructors for a variety of classes. If this bothers you, I urge you to read the Matlab user manual and examine the C-STORM class definitions in Chapter 8.

### 3.2 Step 1: Describe the robot

The first step is to generate a description of the mechanism of interest. In order to accomplish this you will need a description of the robot kinematics, mass properties and link shapes. When describing these, it is very important to use a **consistent** set of units.

Before we attempt to describe the robot kinematics, it is useful to present several definitions:

- **Fixed frame**: This is the user defined fixed (or inertial) frame. This frame is fixed in space and can not move.
• **Home position:** This is a user defined robot configuration in which all the joint angles are zero. All the kinematic properties of the robot will be defined with the robot located in the home position.

• **Joint frame:** This is a local frame attached to each joint of the robot. This frame moves with the link attached to the joint. In the home position, this frame must be a translated, unrotated copy of the fixed frame located at any point on the joint axis. This idea is important during the following steps since the robot is always described in the home position.

• **Joint position:** This is a point on the joint axis defined in the fixed frame. Generally, the chosen point will coincide with the location of the joint – hence the name joint position. The joint position for each joint will change as the robot moves.

After you have chosen a fixed frame for your problem, the next step is to choose a “home” position for the robot. This choice is completely arbitrary – it merely defines the position of the robot when all joint values are considered to be zero. Without getting into topics beyond the scope of this manual, the kinematics of the chain can then be defined by the position and screw vector of each joint when located in the home position. The joint position and joint screw for the openchain class are described with respect to a fixed frame. Normally the fixed frame is located at the base of the robot, but it can be located anywhere. For the openchain class, the joint screw is a 3 dimensional vector. It describes the direction of the joint’s motion axis. The joint can then be defined as either revolute or prismatic in order to determine the type of motion. For example, a joint which revolves around the z-axis (when located in the home position) would have (0,0,1) as the 3 dimensional joint screw and would be defined as a revolute joint. A joint which translated along the z-axis would have (0,0,1) as the joint screw and be defined as a prismatic joint. Note that the joint screw is always defined in the joint frame when the robot is located in the home position.

Once we have this information, we are ready to begin building the robot in Matlab. We first construct individual joints using the joint class. The joint class is quite complicated, but we will begin by simply constructing a joint using the information we have. This is accomplished by calling the constructor function for the joint class.

---

3r_robot.m
\begin{verbatim}
3r_robot.m

j1 = joint(vector3d(0,0,0),vector3d(0,1,0),-4*pi,4*pi,'revolute');

j2 = joint(vector3d(0,1,0),vector3d(0,0,1),-2*pi,2*pi,'revolute');

We have now created a variable called j2 which contains a joint located at (0,1,0) which revolves around the (0,0,1) axis and can move between $-2\pi$ and $2\pi$.

We continue to describe all the kinematic parameters for the joints in the chain. The third joint is located at (1,1,0) and revolves around the (0,0,1) axis:

3r_robot.m

j1 = joint(vector3d(0,0,0),vector3d(0,1,0),-4*pi,4*pi,'revolute');

j2 = joint(vector3d(0,1,0),vector3d(0,0,1),-2*pi,2*pi,'revolute');

j3 = joint(vector3d(1,1,0),vector3d(0,0,1),-2*pi,2*pi,'revolute');

We have now described the three joints in our robot. At this time they are completely separate entities. Later, we will connect them into a single openchain variable.
\end{verbatim}
3.3 Step 2: Defining the link shapes

The next step is to describe the appearance of the link attached to each joint. Currently the available shapes are fairly rudimentary. They include cylinders, spheres and 2 dimensional polygons. More complex shapes, such as polyhedra, can be created using polygons to describe each facet. This is currently non-optimal and a dedicated 3 dimensional polytope class is needed. For now, it is probably best to stick with spheres and cylinders if a 3 dimensional shape is needed.

Each shape has a corresponding Matlab class. The csphere class is used to hold sphere shapes – it could not be called “sphere” since Matlab already has a sphere function. The rod class holds cylindrical shapes – again, the “cylinder” function was taken by Matlab. The poly2d class holds polygon shapes. Any shape or combination of shapes can be used to describe the appearance of the link attached to each joint. It is not necessary to attach a shape to each joint.

Let us return to our example robot and add a cylindrical link to the first joint. It will reach from the origin (0,0,0) to the location of the second joint (0,1,0). We will use a radius of 0.1.

\begin{verbatim}
3r_robot.m

j1 = joint(vector3d(0,0,0),vector3d(0,1,0),-4*pi,4*pi,'revolute');
j2 = joint(vector3d(0,1,0),vector3d(0,0,1),-2*pi,2*pi,'revolute');
j3 = joint(vector3d(1,1,0),vector3d(0,0,1),-2*pi,2*pi,'revolute');

color = 1;
shape1 = rod(vector3d(0,0,0),vector3d(0,1,0),0.1,color);
j1 = addtogeom(j1,shape1);
\end{verbatim}

The variable shape1 now holds the information about the shape – namely the two endpoints of the cylinder, the radius and the color. The color of the shape is defined between 0 and 1 since that is the default range for caxis in Matlab – for more information about Matlab pseudocolor see the help file for caxis in Matlab. It is IMPORTANT to note that the two endpoints – (0,0,0) and (0,1,0) – are represented in the joint frame. This is slightly ambiguous for the first joint, since the origin of the joint frame is located at the origin of the fixed frame. This will become more clear as we add shapes to the other joints.
Once `shape1` is created, it is then attached to `j1` using the `addtogeom` function. It may seem strange to experienced C++ programmers that the `addtogeom` has `j1` as an output and also as an input argument. This is a necessary evil of the Matlab interface. There is no way to change an input argument to a Matlab function, so in order to change the information in `j1`, it must also be a function output.

We will continue to add shapes until we have described all the links in robot. The second link `shape` will be another cylinder:

```
3r_robot.m

j1 = joint(vector3d(0, 0, 0), vector3d(0, 1, 0), -4*pi, 4*pi, 'revolute');
j2 = joint(vector3d(0, 1, 0), vector3d(0, 0, 1), -2*pi, 2*pi, 'revolute');
j3 = joint(vector3d(1, 1, 0), vector3d(0, 0, 1), -2*pi, 2*pi, 'revolute');

color = 1;
shape1 = rod(vector3d(0, 0, 0), vector3d(0, 1, 0), 0.1, color);
j1 = addtogeom(j1, shape1);

color = 0.5;
shape2 = rod(vector3d(0, 1, 0), vector3d(1, 0, 0), 0.1, color);
j2 = addtogeom(j2, shape2);
```

Note that `shape2` is another cylinder. Again, the two end points are written in the joint frame. It goes from the origin (0,0,0) of joint `j2` to the point (1,0,0) – which corresponds to the origin of joint `j3`. To see this, you must remember that the shape coordinates are always given in the joint frame with the robot located in the home position. We have chosen a different color (0.5) for this link.

For the last link, we choose to add another cylinder and also a sphere. The `csphere` class constructor takes the center point, radius and color as inputs. Again, note that the sphere center point is given with respect to the local joint origin. This completes the description of the link shapes.

```
3r_robot.m

j1 = joint(vector3d(0, 0, 0), vector3d(0, 1, 0), -4*pi, 4*pi, 'revolute');
```
j2 = joint(vector3d(0,1,0),vector3d(0,0,1),-2*pi,2*pi,'revolute');
j3 = joint(vector3d(1,1,0),vector3d(0,0,1),-2*pi,2*pi,'revolute');

color = 1;
shape1 = rod(vector3d(0,0,0),vector3d(0,1,0),0.1,color);
j1 = addtogeom(j1,shape1);

color = 0.5;
shape2 = rod(vector3d(0,0,0),vector3d(1,0,0),0.1,color);
j2 = addtogeom(j2,shape2);

color = 0.3;
shape3 = rod(vector3d(0,0,0),vector3d(1,0,0),0.1,color);
shape4 = csphere(vector3d(1,0,0),0.3,color);
j3 = addtogeom(j3,shape3);
j3 = addtogeom(j3,shape4);

3.4 Step 3: Defining the mass properties

The next step in describing the robot is to define the mass properties for each link in the chain. The mass properties include the total mass, inertia about the center of mass and position of the center of mass. Again, it is important to note that the position of the center of mass is given with respect to a local frame located at the joint. Let us continue with the example in hopes that this will become clear. Recall that the first link had a length of 1 (reaching from the origin to the position of the second joint at (0,1,0)). We will model it as a uniform slender rod of mass 1:

3r_robot.m

...
len1 = 1;
I1 = [mass1*len1*len1/12 0 0;0 0 0;0 0 mass1*len1*len1/12];
cml = vector3d(0,len1/2,0);
j1 = setmassinfo(j1,I1,mass1,cml);

Here we see a new function called setmassinfo. The inputs to this function are a member of the joint class (j1), a $3 \times 3$ inertia matrix (I1), a mass (mass1) and a homogeneous 3-dimensional vector from the joint position to the link's center of mass (cml). Note that in order to change the information contained in j1 (adding the mass information), it must also be passed as an output argument of the setmassinfo function. The vector to the center of mass cml is written in the local joint frame. The inertia matrix is always written about the center of mass of the link. The frame it is written in is a translated, unrotated copy of the local joint frame. For this link, that means that $I_{yy} = 0$ since the link is modeled as a slender rod and is aligned with the y axis of the joint frame.

We continue by adding the mass properties for the remaining links:

```matlab
3r_robot.m

mass1 = 1;
len1 = 1;
I1 = [mass1*len1*len1/12 0 0;0 0 0;0 0 mass1*len1*len1/12];
cml = vector3d(0,len1/2,0);
j1 = setmassinfo(j1,I1,mass1,cml);

mass2 = 1;
len2 = 1;
I2 = [0 0 0;0 mass2*len2*len2/12 0;0 0 mass2*len2*len2/12];
cml2 = vector3d(len2/2,0,0);
j2 = setmassinfo(j2,I2,mass2,cml2);

mass3 = 1.5;
len3 = 1;
```
I3 = [0.018 0 0; 0.1847 0; 0 0 .1847];
cm3 = vector3d(0.6667,0,0);
j3 = setmassinfo(j3,I3,mass3,cm3);

The second link is again modeled as a slender rod. To define the inertia matrix we translate a copy of the local joint frame to the link’s center of mass. When we do this, we see that $I_{xx} = 0$ since the link is aligned with $x$ axis of the joint frame and is modeled as a slender rod.

The third link is slightly more complicated since it consists of a cylinder ($m = 1$, $cm = (0.5, 0, 0)$) and a small sphere ($m = 0.5$, $cm = (1, 0, 0)$, radius = 0.3) located at the end effector. We will need to combine the masses and moments of inertia for these two objects. First, we locate the center of mass and then combine the inertia matrices for the two shapes.

\[
I_{sphere} = \begin{bmatrix}
\frac{2}{5}(0.5)(0.3)^2 & 0 & 0 \\
0 & \frac{2}{5}(0.5)(0.3)^2 & 0 \\
0 & 0 & \frac{2}{5}(0.5)(0.3)^2 \\
\end{bmatrix}
\]  

(3.1)

\[
I_{cylinder} = \begin{bmatrix}
0 & 0 & 0 \\
0 & \frac{1}{12}(1.0)(1.0)^2 & 0 \\
0 & 0 & \frac{1}{12}(1.0)(1.0)^2 \\
\end{bmatrix}
\]  

(3.2)

\[
m_{total} = 1.5
\]  

(3.3)

\[
cm_{total} = \frac{0.5(1, 0, 0) + (0.5, 0, 0)}{1.5} = (0.67, 0, 0)
\]  

(3.4)

We can now use the parallel axis theorem to rewrite the two inertias about the center of mass for the composite body and add them together:

\[
I_{sphere} = \begin{bmatrix}
0.018 & 0 & 0 \\
0 & 0.0736 & 0 \\
0 & 0 & 0.0736 \\
\end{bmatrix}
\]  

(3.6)

\[
I_{cylinder} = \begin{bmatrix}
0 & 0 & 0 \\
0 & 0.1111 & 0 \\
0 & 0 & 0.1111 \\
\end{bmatrix}
\]  

(3.7)
\[ I_{total} = \begin{bmatrix} 0.018 & 0 & 0 \\ 0 & 0.1847 & 0 \\ 0 & 0 & 0.1847 \end{bmatrix} \]  

(3.8)

### 3.5 Step 4: Putting it all together

We have now defined the joint positions, joint screws, link shapes and mass properties. The last step is to combine all the `joint` class variables (\( j_1 \), \( j_2 \) and \( j_3 \)) into a single `openchain` class. To accomplish this, we first create a “default” member of the `openchain` class. This is done by calling the `openchain` class constructor without any arguments:

```matlab
3r_robot.m
.
.
.
oc = openchain;
```

This command creates an instance of the `openchain` class called `oc`. We can now combine \( j_1 \), \( j_2 \) and \( j_3 \) as follows:

```matlab
3r_robot.m
.
.
.
oc = openchain;

oc = addjoint(oc, j1);
oc = addjoint(oc, j2);
oc = addjoint(oc, j3);
clear j1, j2, j3;
```
We use the addjoint function to append the joints onto oc. It is imperative that we add the joints in the correct order – namely from the base out to the tip. The openchain class stores a copy of all the joint information we entered earlier (positions, link shapes and mass properties), so we can go ahead and clear j1, j2 and j3 since we will no longer be needing them.

The final appearance of our robot is shown in Figure ??.

### 3.6 Step 5: The environment

Now that we have our robot, we need to place it into an environment. The most important aspect is to define gravity using the setbaseaccel function:

```matlab
3r_robot.m

oc = setbaseaccel(oc, vector3d(0, 9.81, 0));
```

Note that setbaseaccel employs a vector in the opposite direction of gravity. For the above example, by setting the base acceleration to (0, 9.81, 0), we define the gravity vector in the (0, -1, 0) direction with a magnitude of 9.81. Note that the units for gravity should match the units used for the link lengths and mass properties.

We might also want to add an obstacle field to the robot environment. To find more information about this, see Chapter 6.

### 3.7 Example program listing

```matlab
3r_robot.m

j1 = joint(vector3d(0, 0, 0), vector3d(0, 1, 0), -4*pi, 4*pi, 'revolute');
j2 = joint(vector3d(0, 1, 0), vector3d(0, 0, 1), -2*pi, 2*pi, 'revolute');
j3 = joint(vector3d(1, 1, 0), vector3d(0, 0, 1), -2*pi, 2*pi, 'revolute');

color = 1;
shape1 = rod(vector3d(0, 0, 0), vector3d(0, 1, 0), 0.1, color);
j1 = addtogeom(j1, shape1);
```
color = 0.5;
shape2 = rod(vector3d(0,0,0),vector3d(1,0,0),0.1,color);
j2 = addtogeom(j2,shape2);

color = 0.3;
shape3 = rod(vector3d(0,0,0),vector3d(1,0,0),0.1,color);
shape4 = csphere(vector3d(1,0,0),0.3,color);
j3 = addtogeom(j3,shape3);
j3 = addtogeom(j3,shape4);

mass1 = 1;
len1 = 1;
I1 = [mass1*len1*len1/12 0 0 0; 0 0 0 mass1*len1*len1/12];
cm1 = vector3d(0,len1/2,0);
j1 = setmassinfo(j1,I1,mass1,cm1);

mass2 = 1;
len2 = 1;
I2 = [0 0 0 mass2*len2*len2/12 0 0 0 mass2*len2*len2/12];
cm2 = vector3d(len2/2,0,0);
j2 = setmassinfo(j2,I2,mass2,cm2);

mass3 = 1.5;
len3 = 1;
I3 = [0.018 0 0; 0 .1847 0; 0 0 .1847];
cm3 = vector3d(0.6667,0,0);
j3 = setmassinfo(j3,I3,mass3,cm3);

oc = openchain;

oc = addjoint(oc,j1);
oc = addjoint(oc,j2);
oc = adjoin(oc, j3);

clear j1, j2, j3;

oc = setbaseaccel(oc, vector3d(0, 9.81, 0));