Chapter 4

Creating a robot model using C-STORM’s link 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 was described in the previous chapter. Unlike the openchain class, the link class can be used to model branched chains. The link class also provides more flexibility in defining the robot. However, this added flexibility does require some additional work.

In this section, I will “build” a simple robot using the link class. As with the openchain example, this is accomplished using a scripted Matlab .m files. For this example, we will construct a simple branched robot with 4 degrees of freedom. It will be very similar to the example robot created in the previous chapter except that it will have a branch in the chain. We will create a file called bar.robot.m which will contain all the information and function calls needed to construct the robot from inside Matlab.

4.1 Step 0: Understand the openchain class

It is advantageous to have a basic understanding of the openchain class outlined in the previous chapter. While it is certainly possible to use the link class without knowledge of the openchain class, it is important to understand that the link class was an evolution
of the openchain class – fixing many of the limitations imposed and making way for new
abilities. I will make numerous comparisons between the link class and the openchain
class in the following sections.

4.2 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 quantities, 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 positions are described using local joint frame transformations. This is a departure from the fixed frame coordinates used by the openchain class. For example, the home joint positions for the link class are defined using the $4 \times 4$ homogeneous transformation between a local joint frame and the local joint frame of its parent joint. A parent joint is defined as the joint located inward (towards the base) from a given joint. Note that each joint only has one parent joint. The parent joint frame for the joints located at the base of the robot is the fixed frame.

The use of local joint frame transformations in describing the robot kinematics has several advantages. The most compelling of these is the ability to make robot “modules”. Since the robot is represented by local transformations, it is easy to add a serial chain onto an existing robot to create a larger branched chain. This was not possible with the openchain class (even if the resulting robot was an unbranched chain) unless the two chains were both described with respect to the same fixed frame. Another advantage of the link class is the ability to arbitrarily choose the local joint frame orientation. In the openchain class the local joint frames all shared the same orientation as the fixed frame when the robot was located in the home position. This is not normally a problem, but the ability to define different orientations for the local joint frames does exist.

The joint screw for the link class is defined using a six dimensional vector. Three of the components represent translational motion (denoted $v$) and three represent angular motion (denoted $\omega$). In keeping with the modular nature of the link class, the joint screw is written in the local joint frame. This is different from the openchain class in which the joint screw was a 3 dimensional vector and a separate flag denoting whether the motion was translational (prismatic joint) or rotational (revolute joint). The ability of the link class to describe all the joint screw information in a single six dimensional vector allows us to create cylindrical joints which combine linear and angular motions. For example: a joint screw of $(\omega, v) = ((0, 0, 0), (0, 0, 1))$ represents a prismatic joint which moved along the local z axis, a joint screw of $(\omega, v) = ((0, 1, 0), (0, 0, 0))$ represents a revolute joint which rotates around the local y axis and the joint screw $(\omega, v) = ((1, 0, 0), (1, 0, 0))$ represents a cylindrical joint which both translates along and rotates around the local x axis – moving one unit along the axis for every radian it rotates.

Once we have the information needed to describe the robot kinematics, we are ready to begin building the robot in Matlab. Let us begin by creating individual members of the link class. The modular nature of the link class allows us to treat each joint in the
robot as a simple, single degree of freedom link class member. We can then connect these simple robots together in any desired fashion. This is unlike the openchain class which required us to first construct individual members of the joint class and then add them into a member of the openchain class in a structured way (ordered from the base to the tip). We construct the four joints (each represented by a member of the link class) of our robot as follows:

```c
b4r_robot.m

len1 = 1;
m1 = matrix4d(eye(3), vector3d(0,0,0));
m2 = matrix4d(eye(3), vector3d(0,len1,0));
m3 = matrix4d(eye(3), vector3d(len1,0,0));

roty = se3(vector3d(0,1,0), vector3d(0,0,0));
rotz = se3(vector3d(0,0,1), vector3d(0,0,0));

l1 = link(m1,roty);
l2 = link(m2,rotz);
l3 = link(m3,rotz);
l4 = link(m3,roty);
```

We have introduced three new constructors for the matrix4d, se3 and link classes. The matrix4d class is used to represent $4 \times 4$ homogeneous transformation matrices. The matrix4d constructor takes two arguments. The first argument is a $3 \times 3$ Matlab array representing the rotation matrix and the second argument is a vector3d representing the translation vector. These two parts are combined into a single $4 \times 4$ homogeneous transformation matrix. The se3 class is used to represent 6-dimensional screw vectors. The se3 constructor takes two vector3ds as inputs. The first represents the rotational motion and the second represents translational motion. The link class constructor takes two arguments – a matrix4d and a se3. The matrix4d represents the coordinate transformation between the local joint frame and the local frame of its parent link when the robot is located in the home position (recall that the parent link is the next link towards the base from a given link). Often it is simple to align all the local joint frames when the robot is located in
the home position, thereby making the rotation matrix of these transformation the identity matrix as was done in our example. The se3 argument to the link constructor represents the joint screw written in local joint frame coordinates.

4.3 Step 2: Defining the link shapes

The next step is to describe the physical appearance of the links. As mentioned in the previous chapter, 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.

The way in which shapes are used is very similar to the procedure outlined in the previous chapter. Shapes are attached to a member of the link class in the same way they are attached to a member of the joint class. The following Matlab code would attach the needed shapes:

```matlab
b4r_robot.m

...  
color = 1;  
shape1 = rod(vector3d(0,0,0),vector3d(0,len1,0),0.1,color);  
l1 = addtogeom(l1,shape1);  

color = 0.5;  
shape2 = rod(vector3d(0,0,0),vector3d(len1,0,0),0.1,color);  
l2 = addtogeom(l2,shape2);  

color = 0.3;  
shape3 = rod(vector3d(0,0,0),vector3d(len1,0,0),0.1,color);  
shape4 = csphere(vector3d(len1,0,0),0.3,color);  
l3 = addtogeom(l3,shape3);  
l3 = addtogeom(l3,shape4);  
```
color = 0.1;
shape5 = rod(vector3d(0,0,0),vector3d(0,0,len1),0.1,color);
l4 = addtogeom(l4,shape5);

We see that the same function (addtogeom.m) that was used for the joint class is used to add shapes to member of the link class.

4.4 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.

The procedure for this is exactly the same as the procedure outlined in the previous chapter for the openchain class. The setmassinfo function is defined in the same way for the link class as it was for the joint class in the previous chapter:

b4rrobot.m

... 
mass = 1;
I = [mass*len1*len1/12 0 0 0 0 0 mass*len1*len1/12];
cm = vector3d(0,len1/2,0);
l1 = setmassinfo(l1,I,mass,cm);

I = [0 0 0; 0 mass*len1*len1/12 0 0 0 mass*len1*len1/12];
cm = vector3d(len1/2,0,0);
l2 = setmassinfo(l2,I,mass,cm);

mass = 1.5;
I = [0.018 0 0 .1847 0; 0 0 .1847];
cm = vector3d(0.6667,0,0);
l3 = setmassinfo(l3,I,mass,cm);

mass = 1;
\[ I = \begin{bmatrix} \text{mass}\cdot\text{len1}\cdot\text{len1}/12 & 0 & 0 \\ 0 & \text{mass}\cdot\text{len1}\cdot\text{len1}/12 & 0 \\ 0 & 0 & 0 \end{bmatrix}; \]
\[ \text{cm} = \text{vector3d}(0, 0, \text{len1}/2); \]
\[ \text{l4} = \text{setmassinfo}(\text{l4}, I, \text{mass}, \text{cm}); \]

Note that the inertia matrix is defined about the center of mass for each link. It is written in a translated, unrotated copy of the local joint frame with its origin located at the center of mass. The vector to the center of mass is written in the local joint frame. As earlier, we chose to model the links and slender rods.

### 4.5 Step 4: Putting it all together

We have now defined the kinematic and mass properties of each link in our robot. We must now connect these links together into a single mechanism. In the previous chapter, the individual members of the \textit{joint} class were combined into a single member of the \textit{openchain} class. The \textit{link} class is handled in a different manner. Due to the modular nature of the \textit{link} class, there is no fundamental difference between a single link and a combination of several links. This means that we can simply combine the single degree of freedom links into a Matlab array which we will call \textit{l}. This must be done in a strange manner because we have not implemented an overloaded bracket ([]) function for the \textit{link} class. The procedure is as follows:

\begin{verbatim}
 b4r_robot.m

   ...
   l = l1; 
   l(2) = l2; 
   l(3) = l3; 
   l(4) = l4; 

\end{verbatim}

The four links of our robot are now stored in a single Matlab array named \textit{l}. We must now tell Matlab how these links are connected together. This is a new procedure and was not necessary for the openchain class since the individual joints were always ordered from the base to the tip. This is not possible for a branched open chain since there can be more than one “tip” of the robot.

In order to describe the interconnections of the branched chain we must define some new terminology:
• **Parent Link**: The inward (towards the base) link of a given link

• **Child Link**: *One* of the outward (away from base) links of a given link. Note that the choice of which outward link is the child is completely arbitrary. This means that even if a link has several outward links (each leading into a separate branch of the chain), only one of these links is chosen as its child.

• **Next Link**: A link who shares the same parent as a given link and is not already either a child or a next of any other link.

By using these definitions we can completely describe any branched chain. Using the above definitions we can create a “map” for our robot which describes the interconnections of its links. Let us first define the parent for each link in the chain. This is done using the `parentmap` command:

```matlab
b4r_robot.m

...
1 = parentmap(1, [0 1 2 2]);
```

This command has two inputs: the array of links `1` and an array of numbers. Each number in the array is an index into the array of links `1`, with the number zero corresponding to “none”. Let us consider our four link branched robot. The first link (`1(1)`) is at the base of the chain and does not have a parent, so the first number in the array would be zero. The second link has the first link as its parent – making one the second number in the array. The third and fourth links both have link 2 as their parent. This command changes each element of `1` so that it knows which element of `1` is its parent. We can then go on to define the child and next mappings:

```matlab
b4r_robot.m

...
1 = childmap(1, [2 3 0 0]);
1 = nextmap(1, [0 0 4 0]);
```

Again, these commands change the information attached to each element in the array `1`. Each element in `1` now knows which element (if any) is its child and which element (if any) is its sibling.
From this we see that link 2 is the first link's child and link 3 is the second link's child. Link 3 and 4 do not have a child (they have a zero in the childmap array) which means that they are the last link in their branch of the robot. We see that link 3 has link 4 as a next link and that no other links have a next. This makes sense since links 3 and 4 share the same parent (link 2) and link 3 was defined as the child of link 2. NOTE: A quick way to check your mappings is to note that every link but the base link appears once and only once in the child and next mappings.

In this example I put the base link at the beginning of the array l. This is NECESSARY. All of the other links can be ordered inside the array l however you choose, but the base link should be the first link in the array. This is a result of some assumptions I make in the dynamics code.

Once we have defined the parent, child and next mappings we have completed the description of our robot.

4.6 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 baseaccel function:

```matlab
b4r_robot.m

\[
\ldots
1 - \text{baseaccel}(1,\text{vector3d}(0,9.81,0));
\]

Note that baseaccel 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. Also, it is important to note that the baseaccel function for the link class assumes that the base link is the first element in the array l.

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

4.7 Example program listing

b4r_robot.m
len1 = 1;
m1 = matrix4d(eye(3),vector3d(0,0,0));
m2 = matrix4d(eye(3),vector3d(0,len1,0));
m3 = matrix4d(eye(3),vector3d(len1,0,0));

roty = se3(vector3d(0,1,0),vector3d(0,0,0));
rotz = se3(vector3d(0,0,1),vector3d(0,0,0));

l1 = link(m1,roty);
l2 = link(m2,rotz);
l3 = link(m3,rotz);
l4 = link(m3,roty);

color = 1;
shape1 = rod(vector3d(0,0,0),vector3d(0,len1,0),0.1,color);
l1 = addtogeom(l1,shape1);

color = 0.5;
shape2 = rod(vector3d(0,0,0),vector3d(len1,0,0),0.1,color);
l2 = addtogeom(l2,shape2);

color = 0.3;
shape3 = rod(vector3d(0,0,0),vector3d(len1,0,0),0.1,color);
shape4 = csphere(vector3d(len1,0,0),0.3,color);
l3 = addtogeom(l3,shape3);
l3 = addtogeom(l3,shape4);

color = 0.1;
shape5 = rod(vector3d(0,0,0),vector3d(0,0,len1),0.1,color);
l4 = addtogeom(l4,shape5);

mass = 1;
I = [mass*len1*len1/12 0 0;0 0 0;0 0 mass*len1*len1/12];
cm = vector3d(0,len1/2,0);
l1 = setmassinfo(l1,I,mass,cm);

I = [0 0 0; 0 mass*len1*len1/12 0; 0 0 mass*len1*len1/12];
cm = vector3d(len1/2,0,0);
l2 = setmassinfo(l2,I,mass,cm);

mass = 1.5;
I = [0.018 0 0; 0.1847 0; 0 0 0.1847];
cm = vector3d(0.6667,0,0);
l3 = setmassinfo(l3,I,mass,cm);

mass = 1;
I = [mass*len1*len1/12 0 0; 0 mass*len1*len1/12 0; 0 0 0];
cm = vector3d(0,0,len1/2);
l4 = setmassinfo(l4,I,mass,cm);

l = l1;
l(2) = l2;
l(3) = l3;
l(4) = l4;

l = parentmap(l,[0 1 2 2]);
l = childmap(l,[2 3 0 0]);
l = nextmap(l,[0 0 4 0]);

l = baseaccel(l,vector3d(0,9.81,0));