Understanding URDF using MATLAB®

Peter Corke

November 2018

1 Introduction

The Universal Robot Data Format (URDF) is an increasingly common way to represent the kinematic structure and visual appearance of robots. It is widely used in the ROS software ecosystem. A URDF file is an XML format file and a number of specialized tags.

I've not found a particularly helpful or readable tutorial hence I'm writing my own.

2 XML 101

URDF files are expressed in XML so it's helpful to start with a refresher about XML.

2.1 What's in an XML file

The most obvious feature of an XML document are the angle-bracket-delimited *tags* such as <foo> or <bar>. The file contains structured text that can be read by a person as well as being easy to parse programatically.

An XML file is frequently referred to as a *document*. The file comprises *markup* and *content*. Generally, strings that constitute markup either begin with the character < and end with a >, or they begin with the character & and end with a ;. Strings of characters that are not markup are content.

2.1.1 Tags

A *tag* is a markup construct that begins with < and ends with >. Tags come in three flavours:

start-tag such as <tag>

end-tag such as </tag>

empty-element tag such as <tag />

where, in all cases, tag is the name of the tag and is given without quotes. For example <foo> is a tag named foo. Tag names start with a letter, underscore or colon, followed by alphanumeric, dot or underscore characters.

2.1.2 Elements

An *element* is a logical document component that either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag. The characters between the start-tag and end-tag, if any, are the element's *content*, and may contain markup, including other elements, which are called child elements.

An XML element has matching start and end tags like

```
<foo></foo>
```

or single empty-element tag such as

```
<bar />
```

which is equivalent to

```
<bar></bar>
```

The body of the *element* can be an arbitrary mixture of plain text or other elements, ie. elements can be nested within the body of other elements. This allows the creation of hierarchies or trees, with the first or outermost element as the root. For example:

```
<bar>some text</bar>
<foo><bar>one</bar><bar>two</bar></foo>
```

The elements must not overlap. i.e., an end tag of an element must have the same name as that of the most recent unmatched start tag.

2.1.3 Attributes

An *attribute* is a markup construct consisting of `name=value` pairs that exists within a start-tag or empty-element tag. The *attributes* are given as a name and a value. Attribute names are defined without quotation marks, whereas attribute values must always appear as a string delimited with either single or double quotes. For example

```
<img src="madonna.jpg" alt="Madonna" />
```

has two *attributes*. Their names are `"src"` and `"alt"`, and their *values* are `"madonna.jpg"` and `"Madonna"` respectively. An XML attribute can only have a single value and each attribute can appear at most once on each element. Attribute names start with a letter, underscore or colon, followed by alphanumeric, dot or underscore characters. Attribute names in XML (unlike HTML) are case sensitive.

2.1.4 Special characters &··;

Special characters are introduced using XML references. References always begin with the symbol `"&"` which is a reserved character and end with the symbol `";"`. XML has two types of references

Entity References contain a name between the start and the end delimiters. The name refers to a predefined strings such as

`&`	ampersand
`'`	single quote
`>`	greater than
`<`	less than
`"`	double quote

or a larger set defined for HTML which can be found in online lists, for example, `θ` is the Greek letter θ.

Character References These specify a character by its Unicode code in either decimal or hexadecimal format. For example the Roman character "A" can be specified as `A` in decimal or `A` in hexadecimal.

Because ampersand `"&"` indicates a reference it must always be escaped in text as `&`.

2.1.5 Comments and white space

Comments are a special kind of element

```
<!-- comment stuff -->
```

and the body of the comment can also span multiple lines

```
<!--   comment
       more comment
       still more comment -->
```

Comments cannot be nested.

White space is mostly insignificant so XML files are often formatted with indenting to highlight the structure to human readers. Some editors support auto-indenting.

White space is ignored outside of quotes, so the following two lines are equivalent

```
<foo param1 = "x1" param2 = "x2"> body </tag>
<foo param1="x1" param2="x2"> body </tag>
```

2.1.6 CDATA

A CDATA block refers to character data that is considered literally. The special characters are interpreted as themselves, and *not* as markup. For example

```
<![CDATA[
   characters with markup like 3<4;
]]>
```

This section may contain markup characters (`<`, `>`, and `&`). CDATA cannot contain the string `"]]>"`. This is somewhat analogous to a LaTeX verbatim block.

2.1.7 Document

XML documents may begin with an XML declaration that describes some information about themselves. An example is

```
<?xml version="1.0" encoding="UTF-8"?>
```

which is case-sensitive and must be the first statement in the document. An XML document can have only one root element.

2.2 The DOM model

The Document Object Model (DOM) is a hierarchical or tree representation of the file called a node tree. Everything in an XML document is a node:

- The entire document is a document node
- Every XML element is an element node
- The non-markup text in the XML elements are text nodes
- Every attribute is an attribute node
- Comments are comment nodes

The document node is the root of the tree. If must have at most one (XML rule) child node. That node may have multiple children of the types mentioned above. For example the XML element `<date type="year">2005</date>` is represented by an element node `<date>` with two child nodes: an attribute node with the name `"type"` and value `"year"`, and a child text node with the value `"2005"`.

2.3 Reading XML with MATLAB

Consider a short XML file that represents employees at a company

```
1 <?xml version="1.0"?>
2 <people>
3
4   <person employee_number="123" gender="female" >
5     <name>Alice</name>
6     <address>11 Alice St</address>
7     <commenced date="2016-01-01" />
8   </person>
9
10  < person employee_number =  "237" gender = "male" status="on vacation" >
11    <name>Bob</name>
12    <commenced date="2015-07-07" />
13    <address>13  Bob St</address>
14  </person>
15
16 </people>
```

MATLAB provides a function to read and parse an XML file and return a DOM node-tree

Node type	nodeName()	nodeType()	nodeValue()
Element	tag name	ELEMENT_NODE = 1	null
Attribute	attribute name	ATTRIBUTE_NODE =2	attribute value
Text	#text	TEXT_NODE = 3	the string
CDATA	#cdata-section	CDATA_SECTION_NODE=4	the CDATA text
Entity	entity name	ENTITY_NODE=6	null
Comment	#comment	COMMENT_NODE=8	the comment
Document	#document	DOCUMENT_NODE=9	null

Table 1: Names and values for different node types. The symbolic value in column 3 is a property of any node, ie. `node.ATTRIBUTE_NODE` is the value 2.

```
>> root = xmlread('simple.xml');
```

The type of a node is indicated by an enum

```
root.getNodeType()
ans =
     9
```

which Table 1 indicates is a document node. The Table also shows the *name* and *value* associated with different types of node. These are the display value of the variable, shown when it is evaluated without a semicolon on the end of the line

```
>> root
root =
[#document: null]
```

This root node has at least one child node

```
>> root.hasChildNodes
ans =  logical
   1
```

and we can get a list of the child nodes

```
>> rootChildNodes = root.getChildNodes;
>> rootChildNodes.getLength
ans =
      1
```

The first (and only) child node from this list is

```
>> node = rootChildNodes.item(0)
ans =
[people: null]
>> node.getNodeType
ans =
      1
```

which is an element node and the display value shows the tag name: people.

The syntax looks quite unMATLAB-like and that's because the implementation is based on the standard Java DOM processor and the objects are native Java objects. Elements in the list are indexed starting at zero, and we use properties and methods of the Java objects.

This node has a name, the tag name, but no value

```
>> node.getNodeName
ans =
people

>> node.getNodeValue
ans =
     []
```

The people node has five children

```
>> persons = node.getChildNodes();
>> persons.getLength()
ans =
     5
```

which is somewhat surprising since the people node has only two children, the two person nodes: Alice and Bob. We can list the child nodes

```
>> for i=0:node.getLength-1
   persons.item(i)
 end

ans =
[#text:

    ]

ans =
[person: null]

ans =
[#text:

    ]

ans =
[person: null]

ans =
[#text:

]
```

and what we've overlooked is the non-markup text in the XML file, the three blank lines. Since we are only interested in the `person` nodes we can extract them specifically[1]

```
>> persons = node.getElementsByTagName('person');
>> persons.getLength
ans =
     2
```

The first person is

```
>> person = persons.item(0)
person =
[person: null]
>> nodes = person.getChildNodes;
>> nodes.getLength
ans =
     7
```

which has seven child nodes. Once again, we see that this is due to white space in the file

```
>> for i=0:nodes.getLength-1
    nodes.item(i)
end

ans =
[#text:
      ]

ans =
[name: null]

ans =
[#text:
      ]

ans =
[address: null]

ans =
[#text:
      ]

ans =
[commenced: null]

ans =
[#text:
    ]
```

The address node could be obtained by

```
>> address = nodes.item(3)
address =
[address: null]
```

but the index could vary depending on the formatting (white space) in the XML file. It can be obtained more robustly by

```
>> address = nodes.getElementsByTagName('address').item(0)
address =
[address: null]
```

noting that the `.item(0)` is required because the `getElementsByTagName` method always returns a list. The address node has just one child which is a text node whose value is Alice's address

```
>> txt = address.getChildNodes.item(0)
txt =
[#text: 11 Alice St]
>> txt.getNodeValue
ans =
11 Alice St
```

The person node also has attributes. We can obtain a list of those by

```
>> person.hasAttributes
ans =  logical
   1
>> attributes = person.getAttributes;
>> attributes.getLength
ans =
     2
```

The first attribute

```
>> att0 = attributes.item(0)
att0 =
employee_number="123"
```

has a display value that shows both its name and its value. We can get those strings by

```
>> att0.getNodeName
ans =
employee_number

>> att0.getNodeValue
ans =
123
```

Attributes can also be obtained directly

```
>> person.getAttribute('employee_number')
ans =
123
```

This is all somewhat tedious to do manually but it lends itself to a programmatic solution.

3 URDF

A URDF file is an XML-format file that describes both the kinematic structure and the visual appearance of a robot. We will deal with these separately, starting with kinematic structure.

3.1 Kinematic structure

A URDF file is an XML format file with an extension of `.urdf`. A simple example is given below. We use the example of a simple 2 joint arm that moves in the XY plane

```
<?xml version="1.0"?>
<robot name="planar2">

  <link name="base_link" />
  <link name="link1" />
  <link name="link2" />
  <link name="end" />

  <joint name="q1" type="continuous">
    <parent link="base_link" />
    <child link="link1" />
    <axis xyz="0 0 1" />
  </joint>

  <joint name="q2" type="continuous">
    <parent link="link1" />
    <child link="link2" />
    <origin xyz="1 0 0" />
    <axis xyz="0 0 1" />
  </joint>

  <joint name="notajoint" type="fixed">
    <parent link="link2" />
    <child link="end" />
    <origin xyz="1 0 0" />
  </joint>

</robot>
```

Given our newfound knowledge about XML this file is quite easy to interpret. Line 2 is the top-level element in the file with the tag `robot`. Nested below this are elements with the tags `link` and `joint`. The robot, the links and the joints all have names which should be unique.

The links have no content and attributes are used to name them. The first link *must* be named `base_link`.

A joint connects a pair of links. The parent is the link closer to the base (or root) of the mechanism, the child is closer to the tool tip. The first joint, lines 9–13, has attributes that define its named, `q1`, and its type. In this case it is a `continuous` type which is a revolute joint with no motion limits. Nested elements provide additional information about the joint. Lines 10-11 indicate that it connects the base link to link1. Line 12 indicates that the rotation is about the z-axis, in the *parent* link's coordinate frame.

Since no `origin` tag is given the joint connects to the parent link at the origin in its coordinate frame. That point also defines the origin of the child coordinate frame

assuming a joint rotation angle of zero. The first part of the transformation chain for this robot is therefore

$$\boldsymbol{R}_z(q_1)$$

The second joint, lines 15-20, is named `q2` and similar except for the addition of an `origin` element. This specifies that the second joint connects to the parent link, `link1` at the point $[1,0,0]$ and that this point in the world is the origin of the child link. This is shown graphically in Figure 1. The next part of the transformation chain for this robot is therefore

$$\boldsymbol{T}_x(1)\boldsymbol{R}_z(q_2)$$

We are interested in the motion of the end of the robot which is at the end of link 2, at a fixed offset with respect to the link2 frame. To describe this we need to create another link that is *fixed* to link 2 and this is described by lines 22–26. The next part of the transformation chain for this robot is therefore

$$\boldsymbol{T}_x(1)$$

The total transformation chain is

$$\boldsymbol{R}_z(q_1)\boldsymbol{T}_x(1)\boldsymbol{R}_z(q_2)\boldsymbol{T}_x(1)$$

which can be readily interpreted in terms of standard Denavit-Hartenberg parameter as:

θ_i	d_i	a_i	α_i
q_1	0	1	0
q_2	0	1	0

For more complex URDF descriptions this transformation can be performed using the RTB function `DHFactor` and the transform chain might result in Denavit-Hartenberg parameters with joint angle offsets, and non-null base or tool transforms.

The `origin` tag can also contain an orientation. The translation is applied first, then the rotation which is defined in terms of roll (about X), pitch (about Y) and yaw (about Z) where all angles are expressed in radians. Explicitly this is the transform sequence

$$\boldsymbol{R}_z(Y)\boldsymbol{R}_y(P)\boldsymbol{R}_x(R)$$

or ZYX in RTB notation.

URDF explicitly separates links and joints which the Denavit-Hartenberg notation merges together. It is also more general and supports branching mechanisms, that is, a link could have multiple child links which can be used to represent a humanoid robot. Note that some URDF models may not be expressible in Denavit-Hartenberg notation.

A number of different joint types are defined:

revolute a joint that rotates about the specified axis and has a limited range specified by the upper and lower limits.

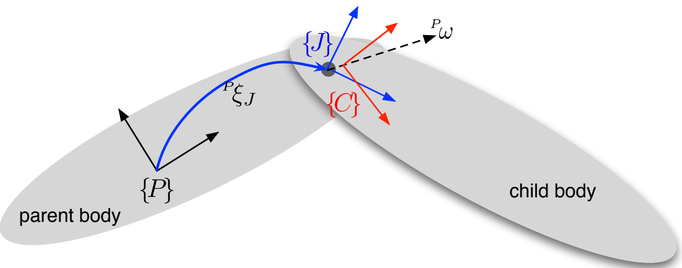

Figure 1: Relationship between parent and child rigid bodies connected by a joint.

continuous a revolute joint that has no upper or lower limits.

prismatic a joint that slides along the specified axis, and has a limited range specified by the upper or lower limits.

fixed is not really a joint because it cannot move. All degrees of freedom are locked. This type of joint does not require the axis.

calibration

floating allows motion for all 6 degrees of freedom.

planar allows motion in a plane perpendicular to the axis.

For prismatic and revolute types the motion limits are specified by nested elements like

```
<limit lower="-0.5" upper="0.5" />
```

where the limits are given in units of radians or metres for revolute for prismatic joints respectively. If either of the `upper` or `lower` limit is not specified, a limit, it defaults to zero.

3.1.1 Link transformations

We will use the notation of [1] where ${}^X\xi_Y$ represents rigid-body motion from frame $\{X\}$ to $\{Y\}$ and $\oplus$ represents rigid-body motion composition. For this problem we consider three coordinates frames: {P} of the parent link, {C} of the child link and {J} of the joint. The transformation from the link frame of the parent frame to the link frame of the child is shown in Fig. 1 and given by

$$ {}^P\xi_C = {}^P\xi_J \oplus {}^J\xi_C(q, {}^C\omega, \sigma) $$

The first transform is a constant set by the`origin` tag of the joint element. The second transform represents the action of the joint itself which is defined by a rotation about ($\sigma = 0$), or translation along ($\sigma = 1$), the vector ${}^P\omega$ by the joint variable q.

3.2 Visual appearance

URDF can also be used to describe graphical objects. A very simple example[2] of this is

```
1 <?xml version="1.0"?>
2 <robot name="myfirst">
3
4   <link name="base_link">
5     <visual>
6       <geometry>
7         <cylinder length="0.6" radius="0.2"/>
8       </geometry>
9     </visual>
10  </link>
11 </robot>
```

which defines a cylinder, with its axis parallel to the z-axis, of length 0.6 m and radius of 0.2 m centred at the world origin. Substituting line 7 for

```
<box size="1 2 3"/>
```

would draw a box centred at the world origin of lengths 1, 2 and 3 m in the x-, y- and z-directions respectively.

By default the cylinder or box is drawn centred at the origin of the link coordinate frame. It can be repositioned by introducing an `origin` element. For example

```
<origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>
```

after line 8 (within the `visual` element) will *first* translate the cylinder by -0.3 m in the z-direction so its top is now at $z = 0$ and *then* rotate it by $pi/2$ about the y-axis.

The object is drawn with a default color[3] but we can change that by altering the material of the link. This can be done adding

```
<material rgba="0 0 0.8 1" />
```

below line 9. The four numbers are red, green, blue in the range 0 to 1 and alpha (0 transparent, 1 opaque). Alternatively

```
1 <?xml version="1.0"?>
2 <robot name="myfirst">
3
4   <material name="blue">
5     <color rgba="0 0 0.8 1"/>
6   </material>
7
8   <link name="link">
9     <visual>
10      <geometry>
11        <cylinder length="0.6" radius="0.2"/>
12      </geometry>
13      <material name="blue" />
14    </visual>
15  </link>
16 </robot>
```

we can create a named material, lines 4–6, and then specify it for the link in line 14.

3.3 Combining kinematics and appearance

Now we will extend our kinematic model of the planar 2-link robot with simple visual models of the links. The links will be drawn as long thin boxes, red and green respectively (somewhat like Fig 7.2b in [1]). The URDF file is now

```
<?xml version="1.0"?>
<robot name="planar2">

  <material name="red">
    <color rgba="1 0 0 0.8" />
  </material>
  <material name="green">
    <color rgba="0 1 0 0.8" />
  </material>

  <link name="base_link" />
  <link name="link1">
    <visual>
      <geometry>
        <box size="1 0.1 0.1" />
      </geometry>
      <origin xyz="0.5 0 0" />
      <material name="red" />
    </visual>
  </link>
  <link name="link2">
    <visual>
      <geometry>
        <box size="1 0.1 0.1" />
      </geometry>
      <origin xyz="0.5 0 0" />
      <material name="green" />
    </visual>
  </link>
  <link name="end">
    <visual>
      <geometry>
        <cylinder radius = "0.05" length="0.2" />
      </geometry>
    </visual>
  </link>

  <joint name="q1" type="continuous">
    <parent link="base_link" />
    <child link="link1" />
    <axis xyz="0 0 1" />
  </joint>
  <joint name="q2" type="continuous">
    <parent link="link1" />
    <child link="link2" />
    <origin xyz="1 0 0" />
    <axis xyz="0 0 1" />
  </joint>
  <joint name="notajoint" type="fixed">
    <parent link="link2" />
    <child link="end" />
    <origin xyz="1 0 0" />
  </joint>

</robot>
```

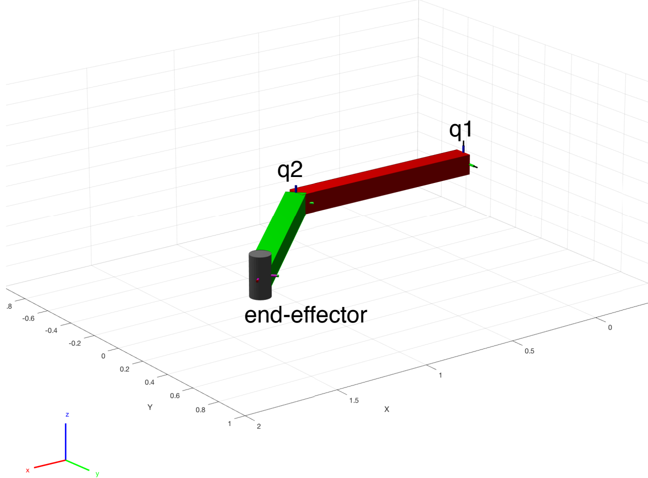

Figure 2: Planar 2 robot with joint angles $[0, \pi/4]$ parsed and plotted using the Robotic Systems Toolbox®.

where lines 38–55 are as before. Lines 4–9 define two translucent materials called `red` and `green` respectively. Link 1 now has a `visual` element defined in lines 13–19. The geometry is a box of length 1 m in the x-direction and by default the reference frame of the visual element is with respect to the reference frame of the link. One end of the box should be at joint 1, at the origin, so we shift it by 0.5 m in the x-direction using an `origin` element (line 17). Finally, we set the color to red using a `material` element (line 18). Link 2 is similar.

The end of the robot is indicated by a small cylinder, in default color, which is centred at the origin of the end link's coordinate frame.

Using the Robotic Systems Toolbox® the file can be parsed and displayed very simply by

```
>> r = importrobot('/Users/corkep/pic3.urdf')
>> r.DataFormat='row'
>> show(r, [0 pi/4])
```

and the result is shown in Figure 2.

3.4 Displaying meshes

The visual geometry can contain a `mesh` element instead of `cylinder` or `box` as shown in this snippet[4]

```
1     <visual>
2       <origin rpy="0 0 0" xyz="0 0 0"/>
```

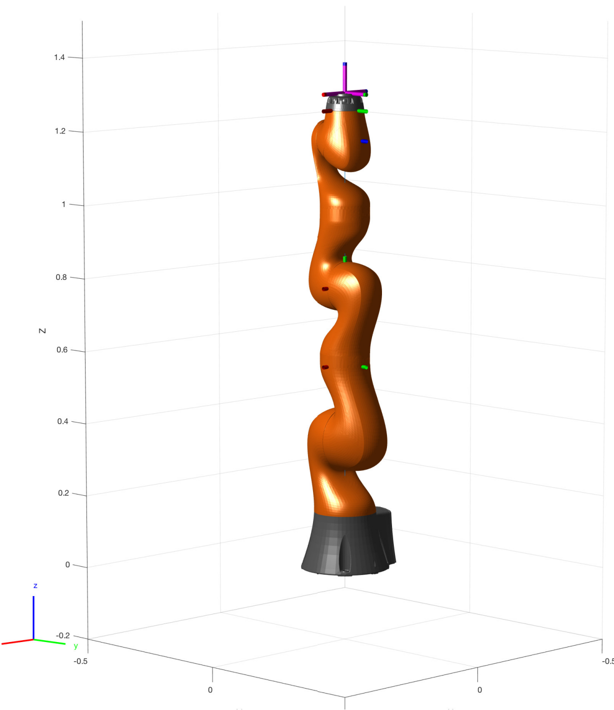

Figure 3: Kuka iiwa14 robot.

```
3        <geometry>
4          <mesh filename="package://iiwa_description/meshes/iiwa14/visual/link_6.stl"/>
5        </geometry>
6        <material name="Orange"/>
7      </visual>
```

A mesh element, line 4, specifies a a filename, and an optional scale factor that scales the mesh's axis-aligned-bounding-box. The recommended format for best texture and color support is Collada .dae files, though .stl files are also supported. The mesh file is not part of the URDF file and must be provided along with the URDF file and have the appropriate path. The <mesh> tag also supports a scale option with a list of x-, y- and z-axis scale factors.

The Robotics System Toolbox includes a model of the Kuka iiwa14 arm and this can be imported, along with its meshes defined in STL format, and displayed

```
>> robot = importrobot('iiwa14.urdf')
>> show(robot)
```

and the result is shown in Figure 3.

3.5 xacro macros

Very large URDF files can be difficult to create and maintain, and this job can be simplified by introducing macros and even conditional code. The ROS package xacro

is a macro-preprocessor written in Python that can be used to transform files written with `xacro` macros into regular XML-based URDF which can be parsed as discussed above. For example

```
<xacro:macro name="pr2_arm" params="suffix parent reflect">
  <pr2_upperarm suffix="$suffix" reflect="$reflect" parent="$parent" />
  <pr2_forearm suffix="$suffix" reflect="$reflect" parent="elbow_flex_$suffix" />
</xacro:macro>
<xacro:pr2_arm suffix="left" reflect="1" parent="torso" />
<xacro:pr2_arm suffix="right" reflect="-1" parent="torso" />
```

expands to:

```
<pr2_upperarm suffix="left" reflect="1" parent="torso" />
<pr2_forearm suffix="left" reflect="1" parent="elbow_flex_left" />
<pr2_upperarm suffix="right" reflect="-1" parent="torso" />
<pr2_forearm suffix="right" reflect="-1" parent="elbow_flex_right" />
```

It can also be useful for parameterizing elements

```
<xacro:property name="the_radius" value="2.1" />
<xacro:property name="the_length" value="4.5" />

<geometry type="cylinder" radius="$the_radius" length="$the_length" />
```

A A simple URDF parser in MATLAB

Following is a fairly complete URDF parser that returns three structure arrays: `joints`, `links` and `materials`. For the `links` array, the `material` field is an integer index into the `materials` array. For the `joints` array, the `parent` and `child` fields are integer indices into the `links` array. The source can be found in LiveScript file `urdfparse.mlx` and is for illustrative purposes only. The Robotics Toolbox for MATLAB includes a complete parser as `urdfparse`.

```
root = xmlread(['planar2b.urdf']);

clear joints links materials

robot = root.getChildNodes.item(0);

materialNodes = robot.getElementsByTagName('material');
i = 1;
for j=1:materialNodes.getLength
    node = materialNodes.item(j-1);
    if isEqualNode(node.getParentNode, robot)
        % only process material nodes at the top level
        materials(i).name = string(node.getAttribute('name'));
        materials(i).rgba = str2num(node.getElementsByTagName('color').item(0).getAttribute('rgb
        materials(i).node = node;
        i = i + 1;
    end
end
try
    materialNames = [materials.name];
end

materialNames
```

```
materialNames =
    "red"    "green"
```

```
materials(2)
```

```
ans =
    name: "green"
    rgba: [0 1 0 0.8000]
    node: [1x1 org.apache.xerces.dom.DeferredElementImpl]
```

```
linkNodes = robot.getElementsByTagName('link');
for i=1:linkNodes.getLength
    node = linkNodes.item(i-1);
    links(i).name = string(node.getAttribute('name'));
    links(i).geometry = node.getElementsByTagName('geometry').item(0);
    try
        links(i).material = find(strcmp((node.getElementsByTagName('material').item(0).getAttrib
    end

    try
        t = str2num( node.getElementsByTagName('origin').item(0).getAttribute('xyz'));
        rpy = str2num( node.getElementsByTagName('origin').item(0).getAttribute('rpy'));
    end
        if isempty(t)
            t = [0 0 0];
    end
    if isempty(rpy)
        rpy = [0 0 0];
    end
    links(i).T = SE3(t) * SE3.rpy(rpy);
    links(i).node = node;
end

linkNames = [links.name]
```

```
linkNames =
    "base_link"    "link1"    "link2"    "end"
```

```
links(2)
```

```
ans =
        name: "link1"
    geometry: [1x1 org.apache.xerces.dom.DeferredElementImpl]
           T: [1x1 SE3]
        node: [1x1 org.apache.xerces.dom.DeferredElementImpl]
    material: 1
```

```
jointNodes = robot.getElementsByTagName('joint');
for i=1:jointNodes.getLength
    node = jointNodes.item(i-1);
    joints(i).name = string(node.getAttribute('name'));
```

```
    joints(i).type = string(node.getAttribute('type'));
    try
        joints(i).axis = node.getElementsByTagName('axis').item(0).getAttribute('xyz');
    end
    joints(i).parent = find(strcmp(node.getElementsByTagName('parent').item(0).getAttribute('lin
    joints(i).child = find(strcmp(node.getElementsByTagName('child').item(0).getAttribute('link'

    try
        t = str2num( node.getElementsByTagName('origin').item(0).getAttribute('xyz'));
        rpy = str2num( node.getElementsByTagName('origin').item(0).getAttribute('rpy'));
    end
    if isempty(t)
            t = [0 0 0];
    end
    if isempty(rpy)
        rpy = [0 0 0];
    end
    joints(i).T = SE3(t) * SE3.rpy(rpy);
    joints(i).node = node;
end

joints(2)
```

```
ans =
      name: "q2"
      type: "continuous"
      axis: [1x1 java.lang.String]
    parent: 2
     child: 3
         T: [1x1 SE3]
      node: [1x1 org.apache.xerces.dom.DeferredElementImpl]
```

References

[1] P. I. Corke, *Robotics, Vision & Control: Fundamental Algorithms in MATLAB*, 2nd ed. Springer, 2017, iSBN 978-3-319-54412-0.