The University Of Sheffield.

Automatic Control & Systems Engineering

ACS6503: Manipulator Robotics Laboratory Sheet 1

2019-2020

Updated: 24 September 2019

Contents

1. Introduction

This lab exercise is to extend the previous work in operating the Mover6 robot arm. In this lab we will continue to control the arm, however, we will use the Robot Operating System to allow us to program arm and interface to other computational environments.

Dr Jonathan Aitken

Module Leader
jonathan.aitken@sheffield.ac.uk

2. Laboratory Procedures

In this laboratory you'll be using a mixture of different control methodologies for simulating and controlling the Mover6 arm. Further information can be found on this arm in the documentation which is available through Blackboard. You should familiarise yourself with this information before you begin using the equipment.

2.1 Familiarisation with the Environment

The Robot Operating System (ROS) is a commonly used piece of middleware that provides access to a wide range of tools that allow robots to be simulate, programmed, and accessed through appropriate drivers. Typical applications can be written in C++ or Python, or through connected applications like Matlab/Simulink and the Robot Systems Toolbox.

There is a substantial amount of information available on ROS. It is worthwhile to review some of the online resources:

- The main Robot Operating Systems webpage: http://wiki.ros.org; this details the packages that are available and details how to setup the middleware, configure the environment and provides a collection of basic tutorials.
- The Matlab Robot Systems toolbox: https://uk.mathworks.com/products/robotics.html; provides an interface which allows for Simulink/Matlab functionality to be extended into ROS applications.
- In this lab we are using the Mover6. This arm has an existing Github page (https://github.com/CPR-Robots/cpr_robot)
-

2.2 Exploring Kinematics

Matlab 2019b, Robot Systems Toolbox enables us to model the various different types of arms that we may expect to use. These can range from simple 2-DOF planar robots to more complex 6-DOF structures like the mover6. The modelling is achieved by understanding the characteristics of each point, much in the same way that we have modelled manipulators geometrically.

In order to build a begin building a rigidBody, we need to define it, as shown in Figure 1:

```
>> body = rigidBody('body1')

body =

  rigidBody with properties:

          Name: 'body1'
         Joint: [1×1 rigidBodyJoint]
          Mass: 1
  CenterOfMass: [0 0 0]
       Inertia: [1 1 1 0 0 0]
      Children: {1×0 cell}
        Visuals: {}
```

Figure 1: Defining a rigidBody

The rigidBody object is the foundation of the robot manipulator, but in order to enable it to move, we need to attach a joint to it. Let's first define a joint, in this case it's a revolute joint (Figure 2):

```
>> jnt1 = rigidBodyJoint('jnt1','revolute')

jnt1 =

  rigidBodyJoint with properties:

                     Type: 'revolute'
                     Name: 'jnt1'
                JointAxis: [0 0 1]
           PositionLimits: [-3.1416 3.1416]
             HomePosition: 0
   JointToParentTransform: [4×4 double]
    ChildToJointTransform: [4×4 double]
```

Figure 2: Defining a Joint

If we look at the joint there are several features that we can define; firstly, there is the JointAxis, this is set to rotate about the Z-Axis. There are PositionLimits that limit the range, and a HomePosition, let's set this to 45 Degrees (Figure 3):

```
>> jnt1.HomePosition = pi/4

jnt1 =

  rigidBodyJoint with properties:

                     Type: 'revolute'
                     Name: 'jnt1'
                JointAxis: [0 0 1]
           PositionLimits: [-3.1416 3.1416]
             HomePosition: 0.7854
   JointToParentTransform: [4×4 double]
    ChildToJointTransform: [4×4 double]
```

Figure 3: Setting the Home Position

At the moment, this joint is just floating in space at the origin; in order to give structure to the robot we need to move it to the appropriate position in space, to do this we set the JointToParentTransform, in this case we will offset its position by 0.25 units in X and Y (naturally we could rotate it or do whatever we want) and assigning it to the joint (Figure 4):

```
>> tform = trvec2tform([0.25, 0.25, 0])

tform =

    1.0000         0         0    0.2500
         0    1.0000         0    0.2500
         0         0    1.0000         0
         0         0         0    1.0000

>> setFixedTransform(jnt1,tform)
```

Figure 4: Assigning Position of Joint

Finally we can assign the joint to our rigidBody (Figure 5):

```
>> body.Joint=jnt1

body =

  rigidBody with properties:

          Name: 'body1'
         Joint: [1×1 rigidBodyJoint]
          Mass: 1
  CenterOfMass: [0 0 0]
       Inertia: [1 1 1 0 0 0]
      Children: {1×0 cell}
        Visuals: {}
```

Figure 5: Assigning Joint to Body

We've now created a rigidBody which has a joint attached. But we haven't defined our robot, our robot will be a collection of rigidBody's connected together. This forms up a rigidBodyTree (Figure 6)

```
>> robot = rigidBodyTree

robot =

  rigidBodyTree with properties:

     NumBodies: 0
        Bodies: {1×0 cell}
          Base: [1×1 rigidBody]
     BodyNames: {1×0 cell}
      BaseName: 'base'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Figure 6: Defining the RigidBodyTree

```
>> addBody(robot,body,'base')
>> robot

robot =

  rigidBodyTree with properties:

     NumBodies: 1
        Bodies: {[1×1 rigidBody]}
          Base: [1×1 rigidBody]
     BodyNames: {'body1'}
      BaseName: 'base'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Figure 7: Adding the rigidBody to the rigidBodyTree

We can then view our rigidBody (Figure 8 and Figure 9):

```
>> robot.show

ans =

  Axes (Primary) with properties:

            XLim: [-0.5000 0.5000]
            YLim: [-0.5000 0.5000]
          XScale: 'linear'
          YScale: 'linear'
    GridLineStyle: '-'
        Position: [0.1300 0.1100 0.7750 0.8150]
           Units: 'normalized'

  Show all properties
```

Figure 8: Showing rigidBody

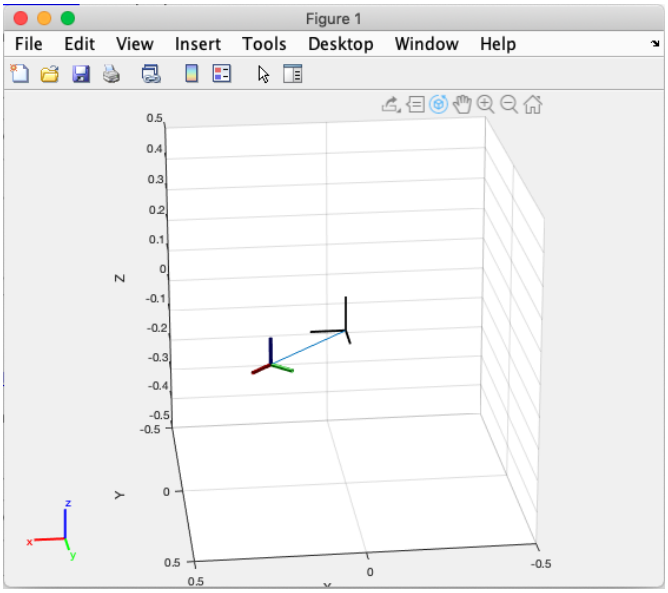


Figure 9: Body Layout (rotated view from created viewpoint)

Task 1: Create an iterative structure of joints that matches the following layout:

Joint 1: As indicated

Joint 2: Revolute joint, at default position of 30 degrees at a distance of 1 unit in the X from joint 1

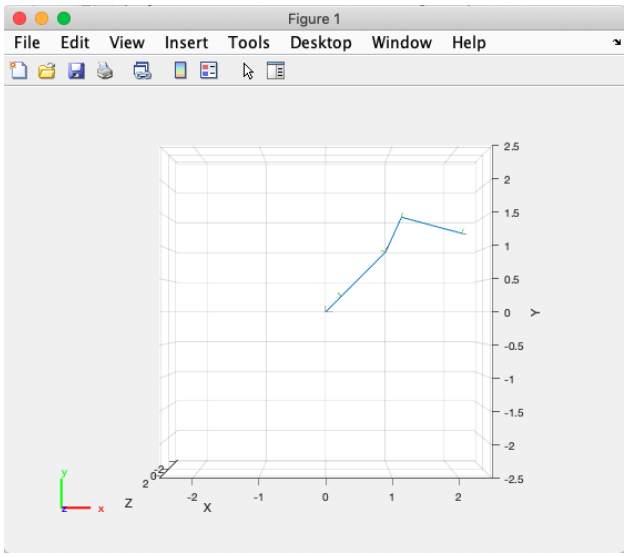
Joint 3: Revolute joint, at a default position of 0 degrees at a distance of 0.6 units in X and -0.1 units in Y from joint 2 with a frame rotation of -90 degrees about Z

Joint 4: Revolute joint, at default position of 45 degrees at a distance of 1 unit in the X direction from joint 3

End effector: Connect this to joint 4.

Hint: Remember that the transform in Joint 3 is a compound transform, and can be made of the multiplication of two components; a translation of $[0.6\ -0.1\ 0]^T$ and a rotation of -90 degrees about the Z axis; check that your resultant transform is correct and states this.

If completed, and the robot is viewed in the X-Y plane (remember there's no movement in Z!), it should look as below:



Task 2a: Determine the transform from the base to the end effector

Task 2b: Determine the transform from joint 1 to the end effector

Task 2c: Are these values what you expect, if so why.

Task 2d: Investigate the transform throughout the kinematic chain looking at each set of pairs. Verify that the overall transform reflects the sequential nature of each component.

Hint: Remember that going form the end effector to the base is equivalent to a series of motions through the robot body.

2.3 Exploring the Kinematics of the Mover6

You have been provided with the file CPMOVER6.urdf. This file is in the form of a *Unified Robot Description File* (URDF); this file describes the physical structure of the arm. The structure of this file describes the geometric structure of the arm, this happens in the same way as when we model a manipulator using a joint-to-joint transform. A sample of the file is shown in Figure 10, this removes the meshes that provide the look and feel of the robot. The components shown detail the structure of the robot, this is formed up as a link-to-link transform, in the same way that we perform a geometric transform.

The initial component that is defined is the base frame (base_link), the subsequent links are defined in relation to their immediate predecessor. This is the same method by which we develop the geometric-based method for defining the kinematics of an open kinematic chain. Moving from link to link you'll see the axis direction given, and the translation of the point from its child (or previous joint).

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

```
<!-- |   This document was autogenerated by xacro from catkin_ws/src/cpr_robot/robots/CPRMover6.urdf.xacro | -->
<!-- |   EDITING THIS FILE BY HAND IS NOT RECOMMENDED                                | -->
<!-- ================================================================================== -->
<robot name="CPRMover6" xmlns:xacro="http://ros.org/wiki/xacro">
  <!-- URDF file for the Commonplace Robotics Mover6 robot arm -->
  <!-- Version 1.1 from Oct. 04th, 2016. -->
                <!—link mesh (animation skin) removed →
<joint name="joint1" type="continuous">
    <axis xyz="0 0 -1"/>
    <parent link="base_link"/>
    <child link="link1"/>
    <origin rpy="0 0 0" xyz="0 0 0.130"/>
    <limit effort="100" velocity="30"/>
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
  <joint name="joint2" type="continuous">
    <axis xyz="0 1 0"/>
    <parent link="link1"/>
    <child link="link2"/>
    <origin rpy="0 0 0" xyz="0 0 0.0625"/>
    <limit effort="100" velocity="30"/>
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
  <joint name="joint3" type="continuous">
    <axis xyz="0 1 0"/>
    <parent link="link2"/>
    <child link="link3"/>
    <origin rpy="0 1.57079632679 0" xyz="0 0 0.190"/>
    <limit effort="100" velocity="30"/>
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
  <joint name="joint4" type="continuous">
    <axis xyz="0 0 1"/>
    <parent link="link3"/>
    <child link="link4"/>
    <origin rpy="0 0 0" xyz="-0.06 0 0"/>
    <limit effort="100" velocity="30"/>
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
  <joint name="joint5" type="continuous">
    <axis xyz="0 1 0"/>
    <parent link="link4"/>
    <child link="link5"/>
    <origin rpy="0 0 0" xyz="0 0 0.290"/>
    <limit effort="100" velocity="30"/>
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
 <joint name="joint6" type="continuous">
    <axis xyz="0 0 1"/>
    <parent link="link5"/>
    <child link="link6"/>
    <origin rpy="0 0 0" xyz="0 0 0.055"/>
    <limit effort="100" velocity="30"/>
    <joint_properties damping="0.0" friction="0.0"/>
  </joint>
</robot>
```

Figure 10: Important components of the URDF for the Mover 6

2.3 Exploring the Kinematics of the Mover6

Let's use Matlab to visualise the arm. Matlab 2019b, Robot Systems Toolbox can load URDF files as an object, as shown in Figure 11.

```
>> mover6=importrobot('CPMOVER6.urdf')

mover6 =

  rigidBodyTree with properties:

     NumBodies: 6
        Bodies: {1×6 cell}
          Base: [1×1 rigidBody]
     BodyNames: {'link1'  'link2'  'link3'  'link4'  'link5'  'link6'}
      BaseName: 'base_link'
       Gravity: [0 0 0]
    DataFormat: 'struct'
```

Figure 11: Matlab Code to Load in URDF Model

We can then view the manipulator rigidBodyTree (Figure 12):

```
>> mover6.show

ans =

  Axes (Primary) with properties:

              XLim: [-1 1]
              YLim: [-1 1]
            XScale: 'linear'
            YScale: 'linear'
     GridLineStyle: '-'
          Position: [0.1300 0.1100 0.7750 0.8150]
             Units: 'normalized'

  Show all properties
```

Figure 12: Matlab Code to View URDF Model

This produces a visualisation of the model (Figure 13) which clearly shows the joint locations, the frame locations and orientations, this will show the robot in the Home Configuration:

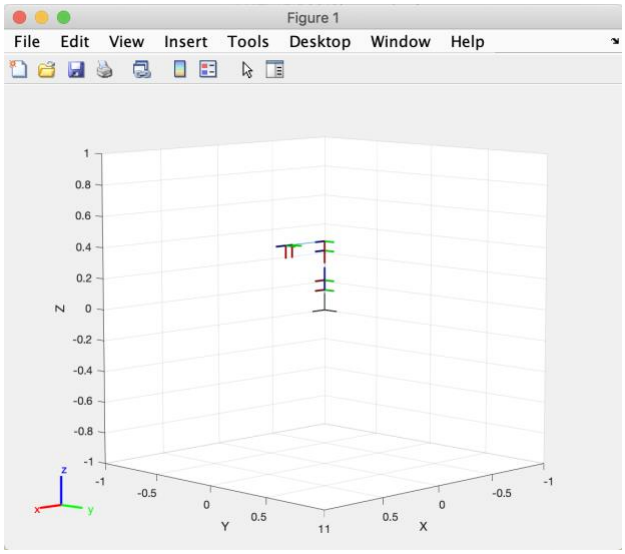


Figure 13: Visualisation of the Matlab Model

Task 3a: Using the geometry contained within the file develop the forward kinematics for the robot, given that it is defined in the URDF at its home position. Given the forward kinematics, calculate the transformation matrix for the home position, do this using a point to point method

Task 3b: Confirm that the model gives the transform below:

The forward kinematics for any position can easily be extracted from the rigidBody model, providing a joint configuration is provided as shown in

```
>> tform = getTransform(mover6,mover6.homeConfiguration,'link6','base_link')

tform =

    0.0000         0    1.0000    0.3450
         0    1.0000         0         0
   -1.0000         0    0.0000    0.4425
         0         0         0    1.0000
```

Figure 14: Transform from base_link (base) to link6 (end effector) for the Mover6 in the Home Configuration

Naturally what we've done is a computationally-led calculation. There's no reason that we shouldn't be able to do this in Matlab!

2.4 Inverse Kinematics of the Mover6

Whilst solving the inverse kinematics of the Mover6 is challenging, we can use Matlab to develop a numerical solution.

Let's start by taking our Mover6 and putting it into a random configuration and getting the transform between the base_link and the end effector (link6), Figure 15 (NB: Your matrix will be different as it's a random choice!):

```
>> randConfig = mover6.randomConfiguration

randConfig =

  1×6 struct array with fields:

    JointName
    JointPosition

>> tform = getTransform(mover6,randConfig,'link6','base_link')

tform =

   -0.0109    0.9347    0.3554    0.1037
   -0.9009    0.1450   -0.4091   -0.0764
   -0.4339   -0.3246    0.8404    0.7064
         0         0         0    1.0000
```

Figure 15: Setting a Random Configuration

We can then view the arm in its random configuration (Figure 16 and):

```
>> show(mover6, randConfig)
```

Figure 16: Showing the Mover6 in a Random Configuration

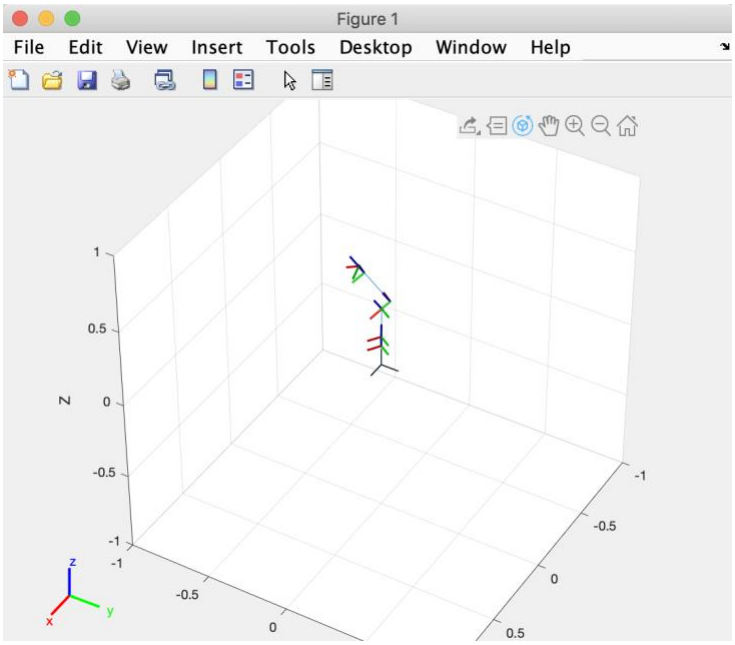


Figure 17: Random Configuration of Mover 6

Let's begin by defining an inverse kinematics object for the URDF model (Figure 18):

```
>> ik=inverseKinematics('RigidBodyTree', mover6)

ik =

  inverseKinematics with properties:

      RigidBodyTree: [1×1 rigidBodyTree]
    SolverAlgorithm: 'BFGSGradientProjection'
   SolverParameters: [1×1 struct]
```

Figure 18: Generating the Inverse Kinematics for the Mover 6

This provides us with an inverse kinematic solution for the arm; as can be seen from the output the BFGSGradientProjection (the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm applied to an unconstrained non-linear problem) method has been used to solve the inverse kinematics, run with parameters as shown in Figure 19.

```
>> ik.SolverParameters

ans =

  struct with fields:

       MaxIterations: 1500
             MaxTime: 10
    GradientTolerance: 1.0000e-07
    SolutionTolerance: 1.0000e-06
   EnforceJointLimits: 1
   AllowRandomRestart: 1
        StepTolerance: 1.0000e-14
```

Figure 19: Optimisation Algorithm Parameters

Let's specify a weights vector that specifies the relative importance of the pose and position in the form [pose position], Figure 20:

```
>> weights = [0.25 0.25 0.25 1 1 1];
```

Figure 20: Weights Vector

Let's make an initial guess for a solution based on the Home Position (we could use anything, but it provides a starting point for the solver so we want a valid position). We can then use the inverse kinematics solver to generate a candidate solution (Figure 21):

```
>> initialguess = mover6.homeConfiguration;
>> [configSoln,solnInfo] = ik('link6',tform,weights,initialguess)

configSoln =

  1×6 struct array with fields:

    JointName
    JointPosition


solnInfo =

  struct with fields:

          Iterations: 49
    NumRandomRestarts: 0
        PoseErrorNorm: 6.1884e-10
             ExitFlag: 1
               Status: 'success'
```

Figure 21: Generating the Inverse Kinematic Solution for the Random Configuration

As can be seen a valid solution has been found, which took 49 iterations. Now we want to investigate the Configuration Solution, and compare the Joint Positions, to those we developed in the Random Configuration. Remember the in the Random Configuration we picked "random" joint angles to achieve a position in space, the Configuration Solution takes that position and then solves backwards for the joint angles. This means we can evaluate the accuracy of the inverse kinematic solution.

Figure 22 shows our set of Joint Positions for the Random Configuration, Figure 23 shows the solution for the inverse kinematic solver. Just from looking at these values it is hard to determine whether we've got our solution correct. However, if we compare the Random Configuration (left) with the Configuration Solution (right), shown in Figure 24, we something very interesting. Whilst the joints are set to different values, the final pose and position of the end effector are identical, the internal coordinates are different, and this is a function of the inverse kinematic solution.

Task 4a: Repeatedly obtain solutions for your random configuration, and verify that they are correct within the workspace, but observe the differences in the internal joint positions. What implications might some of these configurations have on the use of the surrounding area?

Task 4b: Repeat 4a for different seed positions of the robot (i.e. not Home)

Hint: You could generate other Random Configurations.

```
>> randConfig.JointPosition

ans =

    0.5699

ans =

   -0.0149

ans =

   -0.9975

ans =

   -1.5995

ans =

    0.1534

ans =

    0.7156
```

Figure 22: Random Configuration Joint Positions

```
>> configSoln.JointPosition

ans =

    0.5699

ans =

    0.4360

ans =

   -1.7361

ans =

   -0.5049

ans =

    0.3212

ans =

   -0.4013
```

Figure 23: Configuration Solution Joint Positions

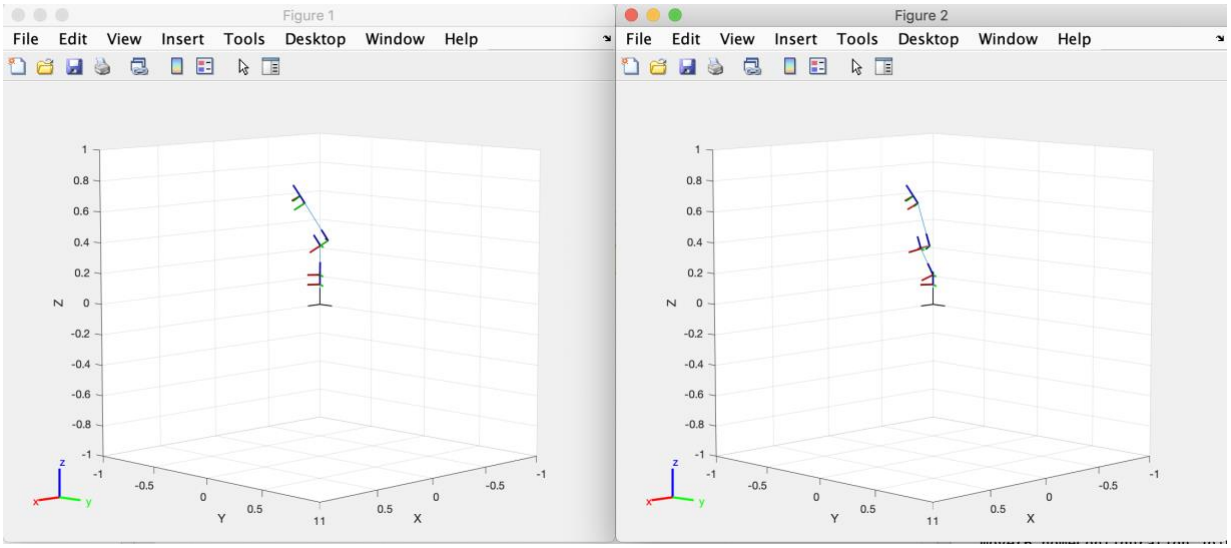


Figure 24: Random Configuration on Left, Configuration Solution on Right