Using the Robotics Toolbox with a real robot

Peter Corke

March 2013

1 The robot

This document describes how to connect a real and relatively inexpensive hobby-class robot to the Robotics Toolbox[1] for MATLAB. The Toolbox provides a convenient environment in which to learn about arm-type robots and experiment with different types of robots (two link, six-link, underactuated and overactuated) and display the results graphically.

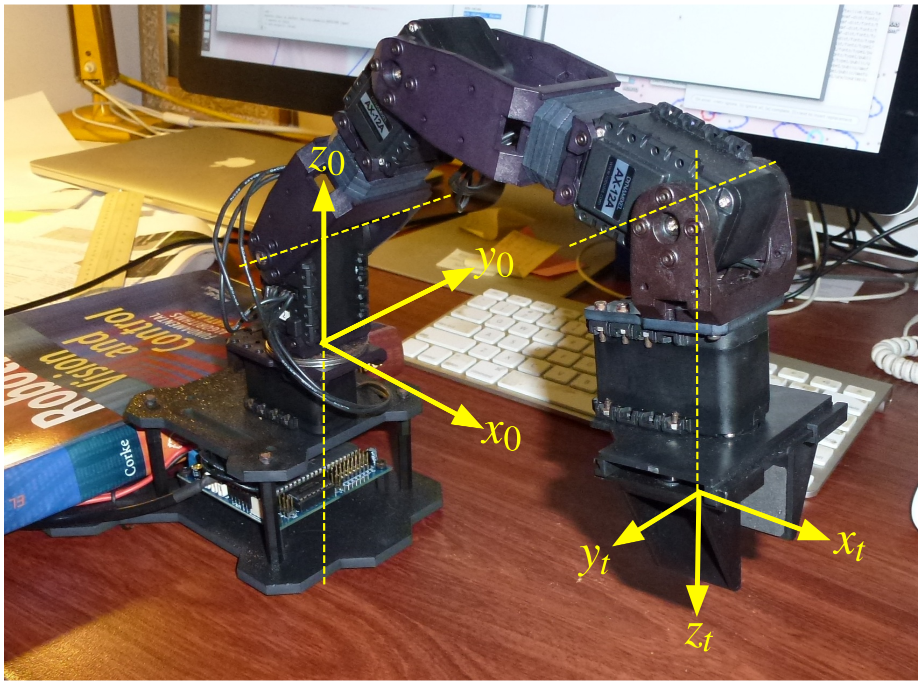

Figure 1: The PhantomX robot sitting on my desk. The world frame {0} and the tool frame {t} are shown. Note I'm using a book over the back feet to keep it from toppling over.

Property	Value
Weight	550 g
Vertical reach	350 mm
Horizontal reach	310 mm
Lifting capacity at 150 mm	100 g
Lifting capacity at 200 mm	70 g
Lifting capacity at 250 mm	40 g

Table 1: Mechanical parameters of the PhantomX Pincher robot.

However there is something to said in favour of the visceral excitement that comes from programming your own robot.

This document describes how to connect the Toolbox, running in the MATLAB environment, to a real four-axis robot that you can purchase for less than USD 400.

The robot we choose is the PhantomX Pincher AX-12 robot from Trossen Robotics. The robot is listed as having 5 degrees of freedom (DOF) but one of these is the gripper so from a kinematic perspective it has 4 DOF. Having less than 6 DOF means that the tool cannot achieve an arbitrary tool orientation at every position that it can reach.

The robot kit contains an ArbotiX controller which uses similiar hardware to the Arduino family of hobbyist embedded computers, a 16 MHz Atmega ATMEGA644P, and can be programmed using the Wiring language and programming environment which runs under Windows, Linux and MacOS. We communicate with the controller using a USB interface which appears as a serial port to the host computer. The ArbotiX has 64k of flash ROM and 4k of RAM.

The actuators on the robot are Dynamixel AX-12 servos which are like model aircraft servos on steroids. They are mechanically powerful and fast rotary actuators with a 300 deg range of motion. They contain an embedded microcontroller that performs the angle control function as well as supporting a packet-oriented communications interface over a 38,400 baud serial channel. The Dynamixel servos can be daisy chained using a 3-wire bus that greatly reduces the amount of wiring required to build the robot. Each servo on the bus is uniquely addressable.

The Dyanmixel servos have a very high level of functionality much of which is not available since the ArbotiX controller intermediates all communications between the host and the servos.

The whole robot and controller is powered from a 12 VDC plug pack. Its overall parameters are listed in Table 1.

2 Getting started

The first step is to connect the robot to a USB port on your computer and power it up. An LED will light on the ArbotiX board underneath the robot but apart from that nothing else will happen, the robot will not move. It's useful at this point to note which serial port device your operating system has created to communicate with the ArbotiX. On a Linux or MacOS system

```
$ ls /dev/tty*
```

and for my computer I see a device that I haven't seen before called `/dev/tty.usbserial-A800JDPN`.

2.1 Flashing the Arbotix

Next you need to ensure that the correct software is installed on the ArbotiX. You need to have the Arduino IDE installed on your computer and this can be downloaded from free from arduino.cc. This web site has a great deal of information to help somebody starting out with Arduino. To run Arduino you will also need to have Java installed. Run Arduino and exit in order to create the default files.

Arduino keeps extensions in a part of the filesystem that varies according to operating system. Under MacOS it is in `~/Documents/Arduino`. Download the arbotix-XXXX.zip file from Trossen Robotics and copy its `arbotix` folder on top of the folder `Arduino`. The other top level folder, `pypose`, contains a Python script to communicate with the `pypose` application loaded into the ArbotiX. However we will use a MATLAB program to perform this role instead.

Now run Arduino again, and use the `Tools` menu options to set the serial port and the type of board, select Arbotix. Select `File/Sketchbook/pypose` to open the `pypose` application source code and then verify it and upload it. In Arduino all programs are referred to as sketches. The program will be loaded into flash program memory so this step need only be done once.

3 Driving the robot

3.1 Make MATLAB talk to your robot

Connect the robot to your computer as above and start MATLAB.

```
>> arb = Arbotix('port', '/dev/tty.usbserial-A800JDPN', 'nservos', 5)
arb =
Arbotix chain on serPort /dev/tty.usbserial-A800JDPN (open)
 5 servos in chain
```

If the class `Arbotix` cannot be found then add `robot/interfaces` to your path. The following command will do the trick

```
>> addpath(fullfile( fileparts(which('startup_rvc')), 'robot', 'interfaces'))
```

If you already have a connection to the Arbotix open you will see a warning message and the class will destroy the old connection before creating a new one.

Now you have a connection to the robot through the workspace class variable `arb`

```
>> about arb
arb [Arbotix] : 1x1 (112 bytes)
```

3.2 Exercising the robot

Let's start with something innocuous, taking the temperature of the servo motors

```
>> arb.gettemp
ans =
    41     40     40     40     43
```

which is the temperature of each servo motor in degrees Celsius. They are a bit warmer than ambient!

Now let's get the actual angle of joint 1, the waist joint

```
>> arb.getpos(1)
ans =
   -0.0869
```

which is the angle in radians. Of course, for your robot you will most likely get a different answer. You can get the joint angle for the other joints in a similar way. We can get all the joint angles simultaneously[1]

```
>> q = arb.getpos([])
q =
   -0.0818    0.4500    0.7159   -0.1125         0
```

where we consider the empty vector `[]` to be the wildcard meaning all joints.

Now let's try a cautious motion, we'll move the gripper to its current position

```
arb.setpos(5, q(5))
```

and nothing happens. Let's change the gripper angle

```
arb.setpos(5, q(5)+0.5)
```

and we hear a noise and see the gripper fingers move slightly.

Now let's move the whole robot to its current location

```
arb.setpos(q)
```

[1]Actually the method has to ask each servo in turn for its joint angle, but at the MATLAB level we can consider it instantaneous.

and the gripper moves back to its original location.

Enough with caution, let's move all joints by a small amount

```
arb.setpos(q + 0.1 * [1 1 1 1 0]);
```

all joints moved. Make sure that you have the robot sitting squarely on the desk, perhaps with a heavy book across the back feet so that it doesn't topple over. This simple move command simply tells the servos to move a new position, and they go as fast as they can. We can set the speed to something more sedate

```
arb.setpos(q, 3 * [1 1 1 1 1]);
```

which moves back to the original pose but with all joints set to a velocity of 3 encoder units per second[2]. The `setpos()` and `getpos()` methods make the conversion from encoder counts to angles.

The robot is quite rigid since the Dynamixel servos are doing their job very well. We can relax the robot by

```
>> arb.relax()
```

which puts all the servos into a zero torque mode and we move the joints by hand. The only force we feel is due to the friction in the Dynamixel gearbox. Change the configuration of the robot and then check the new joint angles

```
>> q = arb.getpos()
q =
    0.0102    0.5573    1.1505   -0.5829    0.0051
```

We can disable the relaxed mode by

```
>> arb.relax([], false)
```

which causes the Dynamixels to start servoing to whatever joint angle they currently have.

4 Robot kinematics

We can create a kinematic model of the PhantomX robot by

```
>> mdl_phantomx
```

which creates the workspace variables `px` (the robot) and `qz` which are the zero joint angles. Although the robot has five servo motors, since one is the gripper, this robot has only four joints

[2]The Dynamixel servos have an encoder value that varies from 0 to 1023 corresponding to -150 to 150 deg of rotation.

```
>> px
px =
PhantomX (4 axis, RRRR, stdDH)

+---+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |
+---+-----------+-----------+-----------+-----------+
|  1|         q1|         40|          0|     -1.571|
|  2|         q2|          0|       -105|      3.142|
|  3|         q3|          0|       -105|          0|
|  4|         q4|          0|       -105|          0|
+---+-----------+-----------+-----------+-----------+

grav =     0  base = 1  0  0  0   tool =    0  0 -1  0
           0         0  1  0  0            -1  0  0  0
        9.81         0  0  1  0             0  1  0  0
                     0  0  0  1             0  0  0  1
```

The physical robot has it's base coordinate frame as shown in Figure 1. Something about tool transform

Let's plot the robot in its zero angle pose

```
>> px.plot(qz)
```

and we see it is standing straight up. We can drive the graphical robot around using the virtual teach pendant

```
>> px.teach()
```

and manipulating the sliders causes the graphical robot to move. Note the coordinate frames of the world and of the tool.

We can also plot the pose of the actual robot, earlier we read the joint angles, so

```
>> px.plot(q)
```

should reflect the pose of your physical robot.

4.1 Forward kinematics

The pose of the end-effector can be determined using forward kinematics, so for the actual pose of the physical robot that is

```
>> px.fkine(q(1:4))
ans =
   -0.9999    0.0102   -0.0102   -4.2268
   -0.0102   -0.9999   -0.0001   -0.0432
   -0.0102   -0.0000    0.9999  321.1689
         0         0         0    1.0000
```

Note that the translational component has units of millimetres, and that we had to explicitly specify `q(1:4)` since the robot has four joints but `q` has five servo motor positions. The end-effector pose can be expressed more concisely as

```
>> trprint(ans)
t = (-4.22678,-0.0432268,321.169), RPY = (0.00599222,-0.585907,-179.414) deg
```

4.2 Inverse kinematics

When the robot is reaching down to lift an object the z-axis of its end-effector is pointing downward, that is, in the negative world z-direction. It's x-axis will be pointing in the same direction as the world x-axis. We can create a pose by

```
>> T=transl(150, 80, 0)*trotx(pi)
T =
    1.0000         0         0  150.0000
         0   -1.0000   -0.0000   80.0000
         0    0.0000   -1.0000         0
         0         0         0    1.0000
```

which has its orientation consistent with that of the end-effector trying to reach an object.

Now we will compute the joint angles required to achieve this end-effector configuration. The robot is underactuated so we need to specify a mask matrix to indicate the DOF that we care about. We will choose: x, y, z and the orientation of z.

```
>> q=px.ikine(T, qz, [1 1 1  0 0 1])
Warning: Initial joint angles results in near-singular configuration, this may slow convergence
> In SerialLink.ikine at 140
Warning: solution diverging, try reducing alpha
> In SerialLink.ikine at 234
Warning: solution diverging, try reducing alpha
> In SerialLink.ikine at 234
Warning: ikine: iteration limit 1000 exceeded (row 1), final err 2.389275
> In SerialLink.ikine at 161

q =
   -8.9349   -8.7578   -8.5723  -10.5647
```

As a quick sanity check we will perform the forward kinematics for these joint angles

```
>> px.fkine(q)
ans =
   -0.5101   -0.4705    0.7200  150.0066
   -0.2720    0.8824    0.3839   79.9876
   -0.8160    0.0000   -0.5781   -0.0000
         0         0         0    1.0000
```

and we see that the translational components are very close, accurate to three significant figures. The orientation is clearly quite different but as already discussed a robot with only four joints cannot achieve an arbitrary orientation at every point in its workspace.

We receive quite a few warning messages and the joint angles are well outside the range $-\pi$ to π. However we did start with a very naive estimate of the joint angles to achieve this end-effector pose. A more sensible initial pose might be

```
>> qn = [0 0.6 -1 -1.2];
```

which can be found by using the graphical teach pendant to position the robot into the approximate pose. Now the inverse kinematics has fewer warnings

```
>> q=px.ikine(T, qn, [1 1 1  0 0 1])
Warning: Initial joint angles results in near-singular configuration, this may slow convergence
> In SerialLink.ikine at 140
Warning: ikine: iteration limit 1000 exceeded (row 1), final err 0.489862
> In SerialLink.ikine at 161

q =
    0.4900    0.6236   -1.1427   -1.3211
```

and gives a result with the joint angles within the range $-\pi$ to π.

5 A RobotArm object

So far we've been using two objects to represent our robot: `px` is the standard Toolbox kinematic model that performs kinematics, plotting, teaching and so on, and `arb` is an interface to the real robot. We can combine them into a single more useful object

```
>> arm = RobotArm(px, arb)
```

The new object

```
>> about arm
arm [RobotArm] : 1x1 (112 bytes)
```

is a subclass of the `SerialLink` class so it inherits all its methods. However it now has a link to the real robot and some additional methods.

Let's start with

```
>> arm.mirror()
```

which puts the arm into the relaxed mode mentioned earlier, but continually reads the joint angles and reflects them to a graphical version of the robot. As you move the physical robot you can see it moving on the screen. The joint angles can be obtained with

```
>> q = arm.getq()
```

The gripper servo is treated separately from motor servos. We can close the gripper

```
>> arm.gripper(0);
```

or fully open it

```
>> arm.gripper(1);
```

The argument is a floating point number between 0 (fully closed) and 1 (fully open).

We can move smoothly to a pose by

```
>> arm.jmove(qz, 5)
```

which will move the robot to the desired joint configuration in 5 seconds. The actually trajectory used is computed using the Toolbox function `jtraj()`.

ALSO CMOVE

References

[1] P. I. Corke, *Robotics, Vision & Control: Fundamental Algorithms in MATLAB*. Springer, 2011. ISBN 978-3-642-20143-1.