Guidelines on the Peter Corke Robotics Toolbox on MATLAB

Prepared by: Eng. Sima Rishmawi

Downloading the toolbox:

- Go to www.petercorke.com
- Click on **Robotic Toolbox**
- From **Contents**, choose **2 Downloading the Toolbox**
- Click on **here** to download the Toolbox in .zip format
- Choose **robot-9.10.zip**
- When the download is complete, extract the .zip file and save it in the c:\ directory. It will have the name **rvctools**

Importing the toolbox to MATLAB:

- Open Matlab.
- In the Command Window type:

```
>> addpath c:\rvctools
```

- Now Matlab knows where to look for functions and commands.
- To start using the toolbox you have to run the startup command, just type:

```
>> startup_rvc
```

Commands:

- <u>3X3 homogeneous Rotation matrix</u>

`>> rotx(theta)` : rotates a frame about the x-axis with an angle theta.

`>> roty(theta)` : rotates a frame about the y-axis with an angle theta.

`>> rotz(theta)` : rotates a frame about the z-axis with an angle theta.

- <u>4X4 homogeneous Transformation matrix</u>

`>> transl(x,y,z)` : translates a frame with x, y, z along respective axes.

`>> trotx(theta)` : rotates a frame about the x-axis with an angle theta.

- To get the complete transformation matrix:

```
>>transl(x,y,z)*trotx(theta)
```

Commands:

>> trplot(R) : plots the coordinate system in the orientation specified by the Rotation matrix R

>> tranimate(R1,R2) : animates the rotation of the coordinate system specified by R2 with respect to the coordinate system specified by R1. If R1 is not specified, its default value is:

$$R_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

which represents the orientation of the universal coordinate system.

>> tripleangle : opens a GUI where you can simulate the Euler convention

rotx(theta)

```
function R = rotx(t, deg)

    if nargin > 1 && strcmp(deg, 'deg')
        t = t *pi/180;
    end

    ct = cos(t);
    st = sin(t);
    R = [
        1    0     0
        0    ct   -st
        0    st    ct
        ];
```

Object-Oriented Programming:

- OOP is based on the concept of **objects** which may contain **data** in the form of fields often known as **attributes.** The code comes in the form of procedures known as **methods.**

- Each object has a number of specific methods that do not apply to other objects. Methods can access the attributes of the object to read them or write on them.

- The computer program is usually made out of objects that interact with each other.

- Most popular OOP languages are **class-based** meaning that objects are **instances** of **classes.**

Classes in the Robotic Toolbox:

- The Peter Corke robotic toolbox is an Object-Oriented Toolbox.
- Examples of Classes:

1. Link

To create an instance of this class (object):

```
>> L = Link(attributes)
```

Attributes of a link are: ‘d’, ‘a’, ‘alpha’, ‘theta’ …

- If a joint is revolute → theta is variable → remove from attributes
- If a joint is prismatic → d is variable → remove from attributes

```
>> L = Link(‘a’,1,’d’,0.5,’alpha’,pi/2)
```

Classes in the Robotic Toolbox:

- The Peter Corke robotic toolbox is an Object-Oriented Toolbox.
- Examples of Classes:

2. SerialLink

To create an instance of this class (object):

```
>> myRobot = SerialLink(attributes)
```

Attributes of a Serial link are: links …

```
>> myRobot = SerialLink(L,'name','myRobot')
```

Some Useful Functions:

- `SerialLinkName.isspherical` gives 1 if the robot has a spherical wrist and 0 if it does not have a spherical wrist.
- `SerialLinkName.fkine(q)` gives the Transformation matrix that describes the position and orientation of the end-effector with respect to the base frame for a set of joint variables specified in the vector q. Dimensions of q should be equal to the number of DOF.
- `SerialLinkName.ikine(T,q0,m)` gives the set of joint variables that solves the inverse kinematic problem. T describes the desired orientation and position of the end-effector with respect to the base frame. q0 specifies the initial estimate of the joint variables. m is called the mask (see next slide).
- `SerialLinkName.plot(q)` draws a schematic picture of the manipulator at a certain pose that depends on the values of the joint variables described in the vector q

Using a Mask with ikine

- For the case were the manipulator has fewer than 6 DOF the solution space has more dimensions than can be spanned by the manipulator joint variables.

- A mask vector should be used to specify the Cartesian DOF that will be ignored when trying to reach a solution.

- The mask vector has six elements, 3 correspond to translation in X, Y, Z, and 3 correspond to rotation about X, Y, Z respectively.

- For example, when using a 3 DOF manipulator, orientation may be ignored: `m = [1 1 1 0 0 0]`

Standard vs. Modified DH-notation:

i	$\alpha_i - 1$	$a_i - 1$	d_i	θi
1	0	0	0	θ_1
2	$-90°$	0	0	θ_2
3	0	a_2	d_3	θ_3
4	$-90°$	a_3	d_4	θ_4
5	$90°$	0	0	θ_5
6	$-90°$	0	0	θ_6

- Standard DH notation:

$a_i\ \alpha_i\ d_i\ \theta_i \rightarrow a_6 = \alpha_6 = 0$

- Modified DH notation:

$a_{i-1}\ \alpha_{i-1}\ d_i\ \theta_i$

- The robotics toolbox uses Standard DH notation.

Building a PUMA560 using the Robotics Toolbox

```
% Build a six degree of freedom PUMA560
robot with six links.

addpath c:\rvctools

startup_rvc


L(1)=Link('d', 0, 'a', 0, 'alpha', -pi/2);

L(2)=Link('d', 0, 'a', 0.432, 'alpha', 0);
% a2=432 mm

L(3)=Link('d', 0.149, 'a', 0.0203, 'alpha',
-pi/2); %a3=20.3 mm, d3=149 mm

L(4)=Link('d', 0.433, 'a', 0, 'alpha',
pi/2); %d4=433 mm

L(5)=Link('d', 0, 'a', 0, 'alpha', -pi/2);

L(6)=Link('d', 0, 'a', 0, 'alpha', 0);
```

```
% The next step is to combine all the
six links into one seriallink (i.e.,

% serial chain manipulator arm.


MyPuma=SerialLink(L, 'name', 'MyPuma')

% Initial pose

q=[0 0 0 0 0 0];


% Find the pose that corresponds to
these set of angles.

T=MyPuma.fkine(q);


% Plot the manipulator

MyPuma.plot(q)
```

Building a PUMA560 using the Robotics Toolbox

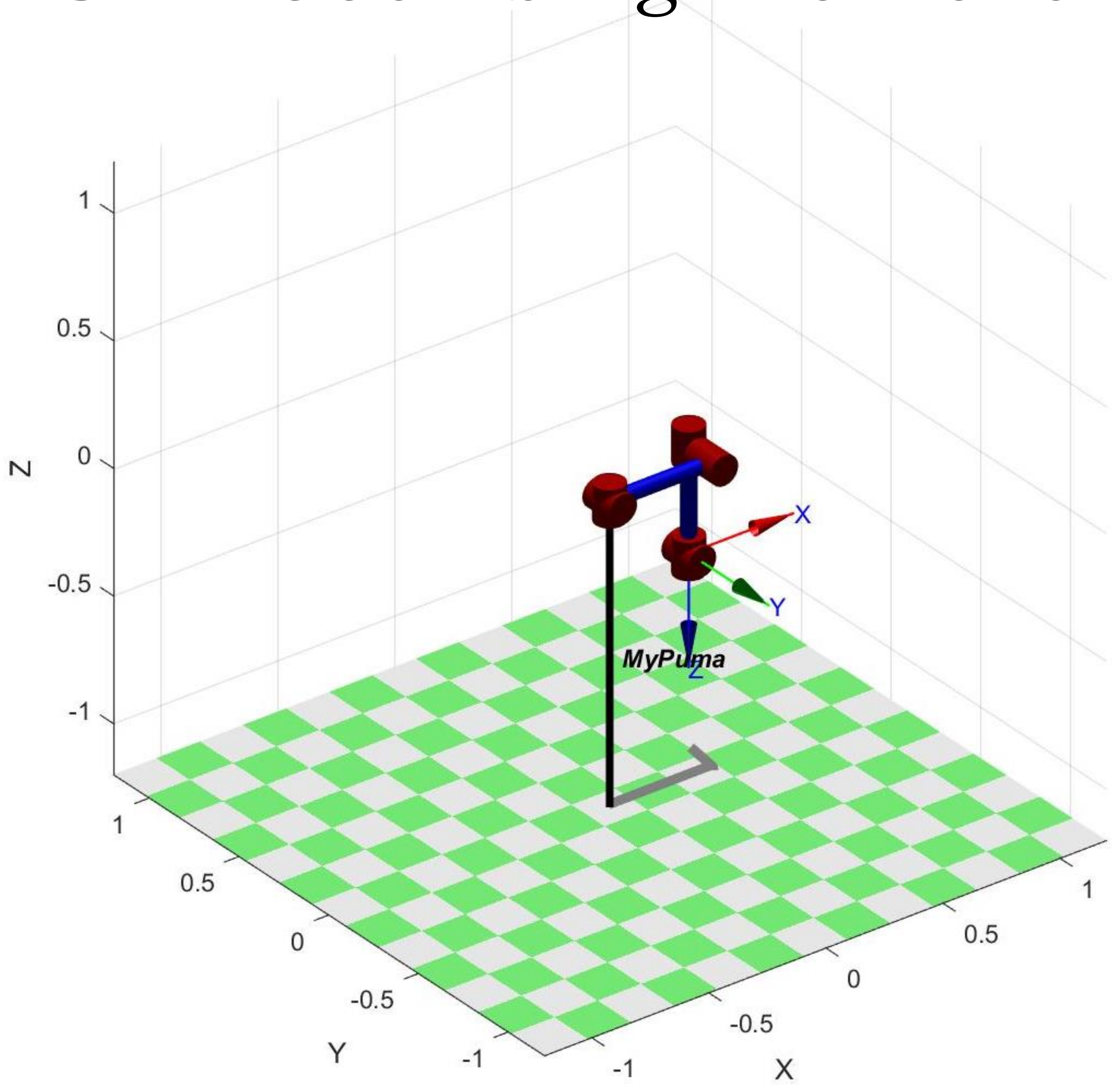
```
MyPuma =

MyPuma (6 axis, RRRRRR, stdDH, fastRNE)

+---+-----------+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |    offset |
+---+-----------+-----------+-----------+-----------+-----------+
|  1|         q1|          0|          0|     -1.571|          0|
|  2|         q2|          0|      0.432|          0|          0|
|  3|         q3|      0.149|     0.0203|     -1.571|          0|
|  4|         q4|      0.433|          0|      1.571|          0|
|  5|         q5|          0|          0|     -1.571|          0|
|  6|         q6|          0|          0|          0|          0|
+---+-----------+-----------+-----------+-----------+-----------+

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

Building a PUMA560 using the Robotics Toolbox



Forward Kinematic Animation (PUMA560)

- Adding these lines to the previous code creates an animation of the robot arm as the joint variables change from qi to qf.
- Keep in mind that this is a Forward Kinematic Problem. Robot is dumb, you are telling it where to go.

```
qi = [0 0 0 0 0 0];
qf = [pi/2 -pi/2 0 0 0 0];

t = [0:0.01:2];
qq = jtraj(qi,qf,t);
MyPuma.plot(qq)
```

Inverse Kinematics of a 4 DOF robot:

```
% Build a four degree of freedom robot with
four links.

% this is basically the first four links of
the puma 560.

L(1)=Link('d', 0, 'a', 0, 'alpha', -pi/2);

L(2)=Link('d', 0, 'a', 0.432, 'alpha', 0);
% a2=432 mm

L(3)=Link('d', 0.149, 'a', 0.0203, 'alpha',
-pi/2); %a3=20.3 mm, d3=149 mm

L(4)=Link('d', 0.433, 'a', 0, 'alpha',
pi/2); %d4=433 mm


% the next step is to combine all the four
links into one seriallink (i.e.,

% serial chain manipulator arm.


FOURDOF=SerialLink(L, 'name', 'FourDof')

%initial pose

q=[0 0 0 0]
```

```
% Find the pose that corresponds to
these set of angles.

T=FOURDOF.fkine(q)


% Change the pose slightly by chaning
the x of the origin.

T(1,4)=0.5


% Prepare the mask

m=[1 1 1 1 0 0]


% Carry out the inverse kinematics.

qi=FOURDOF.ikine(T, q, m)


FOURDOF.plot(qi)
```

Trajectory Planning

- There are two functions to perform trajectory planning:

```
Tc = ctraj(T1,T2,n)
```

This is a Cartesian Trajectory from pose T1 to pose T2 with n points that follow a trapezoidal velocity profile along the path.

```
[q qd qdd] = jtraj(q1,q2,n)
```

This is a Joint Trajectory from q1 to q2. A 5^{th} order polynomial is used with zero boundary conditions for velocity and acceleration.

Singularities

- This code will be used to plot joint angle changes around a singularity:

```
% This example shows the problem of a path passing near a singularity, we first do this for a cartesian path moving in a straight line from the first pose to the second pose.

% This is then repeated for the case of joint space trajectory.

addpath c:\rvctools

startup_rvc

% Generate a puma 560 model

mdl_puma560

% Generate a time vector over 2 seconds.

t=[0:0.1:2]

% Set the first pose

T1=transl(0.5, 0.3, 0.44)*troty(pi/2)

% Set the second pose

T2=transl(0.5, -0.3, 0.44)*troty(pi/2)

% Generate a number of intermediate poses between the two poses

Ts=ctraj(T1, T2, length(t))

% Find the inverse kinematics trajectory for the joints to achieve these intermediate poses.

qc =p560.ikine6s(Ts)

% Plot the joint angles against time.

qplot(t,qc)

hold on

pause

% Start a new figure

figure

% Do a joint space trajectory betweent the two poses

q1=p560.ikine6s(T1)

q2=p560.ikine6s(T2)

[qj qjd qjdd]=jtraj(q1,q2,length(t))

% Plot the set of joint angles against time.

qplot(t,qj)

pause

close

qplot(t,qjd)

pause

close

qplot(t,qjdd)
```

Manipulability

- This code will be used to calculate the manipulability of the PUMA560 in 4 different poses

```
% This code finds the manipulability of the Puma560 robot for 4
% pre-defined poses in the Puma560 model

addpath c:\rvctools
startup_rvc

% Load Puma560 Model
mdl_puma560

% Display 4 pre-defined poses of the model
qz      % All joint angles are zero
qs      % Manipulator is fully stretched out horizontally
qr      % Manipulator is fully stretched out vertically
qn      % Nominal location that is far from singularities
```

```
p560.plot(qz)
pause
p560.plot(qs)
pause
p560.plot(qr)
pause
p560.plot(qn)

% Calculating Manipulability
mz = p560.maniplty(qz)
ms = p560.maniplty(qs)
mr = p560.maniplty(qr)
mn = p560.maniplty(qn)
```

Skew Symmetric Matrix

- To find the Skew Symmetric Matrix of 3 elements, use:

```
S = skew(a,b,c)
```

- To find the 3 elements that create the Skew Symmetric Matrix, use:

```
[a b c] = vex(S)
```

Finding the Jacobian using RTB:

- To find the Jacobian Matrix for a certain pose use:

```
J = p560.jacob0(q)
```

- To know if a singularity exists, use:

```
jsingu(J)
```

Torque Calculations

```
% This code calculates the joint torques for the
Puma560 in different configurations.

% Access the Robotic Toolbox and Start using it

addpath c:\rvctools

startup_rvc

% Define the Puma560 model

mdl_puma560

% Set a time vector:

t = [0:0.5:10];

% Calculate the angles, angular velocities and
accelerations for a certain joint trajectory (robot
is supposed to move from the zero position to the
nominal position

[q, qd, qdd] = jtraj(qz,qn,length(t));

% Plot the angles, angular velocities and
accelerations

figure(1)

qplot(q)

figure(2)

qplot(qd)

figure(3)

qplot(qdd)

% Scaling velocity and acceleration to 10 s

qd = (1/10)*qd;

qdd = (1/10)*qdd;


% Calculate the Motor Torques required to
account for gravitation loads(Static Loads)

Qstat = p560.gravload(q)


% Calculate the Motor torques required to move
the joints with the calculated speeds and
accelerartions:

Qdyn = p560.rne(q,qd,qdd)


% Calculate the Motor Torques required to move
the joints with the calculated speeds and
accelerations with a load at the end-effector:

% Add a load at the tip

p560.payload(2.5,[0 0 0.1]) %load is 2.5 N, its
center of gravity is at 10 cm from the end-
effector

Qload = p560.rne(q,qd,qdd)
```