RobotBlockset for Matlab (RBS)

Synopsis

RobotBlockset (RBS) toolbox provides tools for designing, simulating, and testing robot manipulators in MATLAB and Simulink. The toolbox provides the means to describe the position and orientation of objects in 3D space and spatial velocities using homogenous matrices or quaternions. It defines it's own quaternion class with all necessary quaternion operations and provides functions for relevant transformations between different representations. For robot manipulators the toolbox provides algorithms for spatial trajectory generation, forward and inverse kinematics, considering also robot intrinsic and user defined functional redundancy, and control algorithms for motion control.

For robot models, grippers and sensors custom classes are defined, which let you execute your robot applications by connecting the robot models to different external simulation environments like MuJoCo Haptix, RoboWorks, Gazebo or CoppeliaSim. The toolbox provides models of some common robot manipulators from KUKA LWR, Franka Emika and Universal Robotics. For other robots tools for generation of kinematic models of serial-link manipulators using Denavit-Hartenberg parameters or URDF representations are provided. The main advantage of the toolbox is that it provides an unified higher level syntax to operate different robots. It allows the user to operate the robot in the simulation environment or to connect directly to a robotics platform and operate a real robot.

Installation

You need to have a recent version of MATLAB.

To install the RBS Toolbox on your computer create a new folder . Then copy the complete toolbox to that folder.

To obtain the RBS from github execute from the shell:

```
git clone https://repo.ijs.si/leon/robotblockset
```

Next, it is necessary to set the MATLAB paths. For that from within MATLAB in folder execute the following command:

```
>> startup_rbs
```

This sets up the MATLAB path appropriately but it's only for the current session. You can either:

1. Repeat this everytime you start MATLAB
2. Add the MATLAB commands above to your `startup.m` file
3. Once you have run `startup_rbs`, run `pathtool` and push the `Save` button, this will save the path settings for subsequent sessions.

Alternatively, if you do not need all RBS Toolbox functionality, you can include corresponding folders into MATLAB path.

Mandatory:

```
addpath('../robotblockset')
addpath('../robotblockset/robot_models')
addpath('../robotblockset/xml2struct')
```

For sensors and other peripheral devices:

```
addpath('../robotblockset/drivers')
```

For ROS-based robots:

```
addpath('../robotblockset/ROS')
```

Support for paths, parametrization using RBF and DMP:

```
addpath('../robotblockset/paths')
addpath('../robotblockset/RBF')
addpath('../robotblockset/DMP')
```

Graphics utilities:

```
addpath('../robotblockset/graphics')
```

Support for simulation using Simulink:

```
addpath('../robotblockset/simulink')
```

For simulation of robots in MuJoCo Haptix (Windows) you have to install MuJoCo Haptix. Download MuJoCo Haptix from https://www.roboti.us/download/mjhaptix150.zip and extract to desired directory . Add Haptix folder apimex and `Haptix` folder to MATLAB path.

```
addpath('../mjhaptix/apimex')
addpath('../robotblockset/Haptix')
```

For simulation of robots in RoboWorks(Windows) you have to install RoboWorks and to add `RoboWorks` folder to MATLAB path.

```
addpath('../robotblockset/RoboWorks')
```

For simulation of robots in Copellia Robotics simulation environment (Windows) you have to install CoppeliaSim. Download CoppeliaSim from https://www.coppeliarobotics.com/downloads and install it. Add folder `Coppelia` to MATLAB path.

```
addpath('../robotblockset/Coppelia')
```

Functions for spatial mathematics

The position and orientation of an object can be described using different representations. In RBS Toolbox we use following representations:

Representation	Single	Multiple	Comment
pose	4 x 4 matrix	4 x 4 x n array	homogenous matrix
pose	1 x 7 vector	n x 7 array	[position, quaternion]
position	3 x 1 vector	n x 3 array	position vector
orientation	3 x 3 matrix	3 x 3 x n array	rotation matrix
orientation	1 x 1 quaternion	n x 1 cell array	unit quaternion object
orientation	1 x 4 vector	n x 4 array	unit quaternion elements
orientation	1 x 3 vector	n x 3 array	Euler angles RPY (`Rz` * `Ry` * `Rx`)

To work with translation and rotation (position and orientation), RBS Toolbox provides:

- functions to create vectors and matrices representing the poses, positions and orientations in 3D space,
- functions for transformations between different representations,
- utility functions,
- path and trajectory generation functions, and
- interpolation functions, and
- approximation functions.

Translations and rotations

All rigid body movements are rotations, translations, or combinations of the two. To define a translation is straightforward

```
>> p=[2 -1 0.3]'
p =
    2.0000
   -1.0000
    0.3000
```

On the other hand, rotations can be defined using functions for principal rotations around x, y, and z axis and for rotation around a vector

```
>> rot_x(pi/6)
ans =
    1.0000         0         0
         0    0.8660   -0.5000
         0    0.5000    0.8660
>> rot_y(-pi/4)
```

```
ans =
    0.7071         0   -0.7071
         0    1.0000         0
    0.7071         0    0.7071
>> rot_z(pi/2)
ans =
    0.0000   -1.0000         0
    1.0000    0.0000         0
         0         0    1.0000
>> rot_v([1 1 1],0.4)
ans =
    0.9474   -0.1985    0.2511
    0.2511    0.9474   -0.1985
   -0.1985    0.2511    0.9474
```

or we can define the rotation by three Euler angles (we support commonly used RPY Euler angles)

```
>> rpy2r([pi/6 -pi/2 pi/2])
ans =
    0.0000   -0.8660    0.5000
    0.0000   -0.5000   -0.8660
    1.0000    0.0000    0.0000
>> rot_z(pi/6)*rot_y(-pi/2)*rot_x(pi/2)
ans =
    0.0000   -0.8660    0.5000
    0.0000   -0.5000   -0.8660
    1.0000    0.0000    0.0000
```

The rotations cen be represented in many ways. We support representations by means of rotation matrices, unit quaternions and Euler RPY angles.

Quaternions

The RBS Toolbox provides a class for representing and using quaternions as objects. A quaternion object can be created using different input parameters

```
>> Q=quaternion([1 0 1 0])
Q =
1.0000 <0.0000 1.0000 0.0000>
>> Q=quaternion(1,0,1,1)
Q =
1.0000 <0.0000 1.0000 1.0000>
>> Q=quaternion([0 1 0],pi/4) % vector and angle
Q =
0.9239 <0.0000 0.3827 0.0000>
>> Q=quaternion([0 -1 0;1 0 0;0 0 1]) % rotation matrix
Q =
0.7071 <0.0000 0.0000 0.7071>
>> Q=quaternion([1 2 3]) % pure quaternion using vector
Q =
0.0000 <1.0000 2.0000 3.0000>
```

Note that some of the above quaternions do not represent rotations as they are not unit quaternions. Therefore, it is necessary to normalize them

```
> Q=quaternion(1,0,1,1)
Q =
1.0000 <0.0000 1.0000 1.0000>
>> Q=normalize(Q)
Q =
0.5774 <0.0000 0.5774 0.5774>
```

The `quaternion` class supports usual quaternion operations, including

Operation	Description
`q1 + q2`	Addition
`q1 - q2`	Subtraction
`q1 * q2`	Multiplication
`q1 / q2`	Division
`q * s == s * q`	Scalar multiplication
`q / s` and `s / q`	Scalar division
`inv(q) == 1/q`	Reciprocal

In addition to the basic operations, we also provided a number of extra dependant properties and methods that are particularly useful for quaternions and their application in robotics, including

- Properties to reference quaternion components or represent them in other forms

Property	Description
`q.d`	4 quaternion elements
`q.s`	quaternion scalar element
`q.v`	quaternion vector component
`q.R`	quaternion as 3 x 3 rotation matrix
`q.T`	quaternion as 4 x 4 homogenous matrix

- Methods related to norms and distances

Method	Description
`abs(q)` , `norm(q)`	quaternion norm
`normalize(q)`	quaternion normalization (make it unit quaternion)
`err(q1,q2)`	distance between two quaternions

- Calculation methods

Method	Description
`exp(q)`	Exponential
`log(q)`	Logarithm
`integ(q,w,dt)`	Integration
`slerp(q1,q2,s)`	Spherical linear interpolation (SLERP)

- Representations

Method	Description
`disp(q)`	Quaternion display
`plot(q)`	Graphic representation of quaternion as a rotated frame

When we have a sequence of quaternions, they are gathered in a cell array

```
>> q1=quaternion(rot_x(pi/2))
q1 =
0.7071 <0.7071 0.0000 0.0000>
>> q2=quaternion(rot_y(pi/4))
q2 =
0.9239 <0.0000 0.3827 0.0000>
>> qq=slerp(q1,q2,0:0.5:1)
qq =
  3x1 cell array
    {1x1 quaternion}
    {1x1 quaternion}
    {1x1 quaternion}
>> qq{2}
ans =
0.8969 <0.3889 0.2105 0.0000>
```

In some cases, it is more convenient to use array instead of cell array. Also, the use of quaternion objects is not particularly suitable when we work with spatial poses, where we want to join position and orientation information in one variable. Therefore, we have included into the RBS Toolbox functions that can deal with quaternions represented as an array with four elements in row. A cell-array of quaternions can be easily converted into numerical array

```
>> qq=slerp(q1,q2,0:0.5:1)
qq =
  3x1 cell array
    {1x1 quaternion}
    {1x1 quaternion}
    {1x1 quaternion}
>> qqa=q2q(qq) % convert from quaternion o quaternion array
qqa =
    0.7071    0.7071         0         0
    0.8969    0.3889    0.2105         0
    0.9239         0    0.3827         0
>>
```

We provide for quaternion arrays several functions for quaternion operations

Function	Description
`q2q(q)`	Quaternion object to array of quaternion elements
`qtranspose(q)`	Transpose
`qmtimes(q1,q2)`	Product
`qexp(q)`	Exponential
`qlog(q)`	Logarithm
`qinv(q)`	Inverse of quaternion array
`qnormalize(q)`	quaternion normalization (make it unit quaternion)
`qerr(q1,q2)`	distance between two quaternions
`qmean(q)`	quaternion mean rotation
`qnormalize(q)`	quaternion normalization (make it unit quaternion)
`qinterp(q1,q2,s)`	Spherical linear interpolation (SLERP)
`qspline(q,s,mode)`	Spline interpolation for array of quaternions (squad or hermite_cubic)
`qplot(q)`	Graphic representation of quaternion as a rotated frame

```
>> q1=[0.9950 0.0998 0 0]; % rot_x(0.2)
>> q2=[0.9211 0.3894 0 0]; % rot_x(0.8)
>> qerr(q2,q1)
ans =
    0.6000
         0
         0
```

Rotation matrices

We provide several functions for rotation matrices operations

Function	Description
`rder(R,w)`	Quaternion object to array of quaternion elements
`rexp(R)`	Exponential
`rlog(R)`	Logarithm
`rerr(R1,R2)`	distance between two rotations
`rinterp(q1,q2,s)`	Spherical linear interpolation (SLERP)

Spatial poses

When we are interested in translation and rotation (position and rotation) of one object, i.e. in the pose of the object, then it is conveniet to join translation and rotation into single representation. We use in most cases two representations:

- homogenous matrix

```
>> T=[rot_y(pi/4) [1 -0.2 0.4]';0 0 0 1]
T =
    0.7071         0    0.7071    1.0000
         0    1.0000         0   -0.2000
   -0.7071         0    0.7071    0.4000
         0         0         0    1.0000
>>
```

- array of position and quaternion [pos quaternion]

```
>> x=[1 -0.2 0.4 0.9239 0 0.3827 0]
x =
    1.0000   -0.2000    0.4000    0.9239         0    0.3827         0
```

Transformations between different representations

To transform from one representation to another use one of the following functions

	`T`	`x`	`y`	`Rp`	`R`	`q`	`rpy`	`v`	`p`
Homogenous matrix `T`		`t2x`	`t2prpy`	`t2rp`	`t2r`	`t2q`			`t2p`
Pose vec `x`	`x2t`		`x2prpy`			`x(4:7)`			`x(1:3)`
Pos+Euler angles `y`	`prpy2t`	`prpy2x`					`x(4:6)`		`x(4:7)`
Rotation matrix `R`, **position** `p`	`rp2t`								
Rotation matrix `R`	`rp2t`					`r2q`	`r2rpy`, `r2zyx`	`r2v`	
Quaternion `q`	`q2t`	`q2x`			`q2r`		`q2rpy`, `q2zyx`		
Euler angles `rpy`, `zyx`					`rpy2r`, `zyx2r`	`rpy2q`, `zyx2q`			
Axis/angle `v`					`v2r`				
Position `p`	`rp2t`, `p2t`								

```
>> R=rot_x(1.1)
R =
    1.0000         0         0
```

```
         0    0.4536   -0.8912
         0    0.8912    0.4536
>> p=[0.2 -0.3 0.1]'
p =
    0.2000
   -0.3000
    0.1000
>> rp2t(R)
ans =
    1.0000         0         0         0
         0    0.4536   -0.8912         0
         0    0.8912    0.4536         0
         0         0         0    1.0000
>> rp2t(p)
ans =
    1.0000         0         0    0.2000
         0    1.0000         0   -0.3000
         0         0    1.0000    0.1000
         0         0         0    1.0000
>> T=rp2t(R,p)
T =
    1.0000         0         0    0.2000
         0    0.4536   -0.8912   -0.3000
         0    0.8912    0.4536    0.1000
         0         0         0    1.0000
>> x=t2x(T)
x =
    0.2000   -0.3000    0.1000    0.8525    0.5227         0         0
>>
```

Utility functions

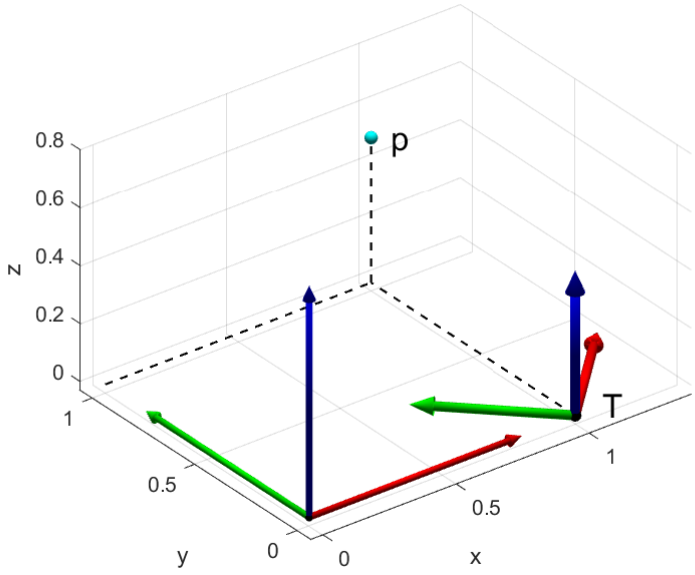
There are several utility functions related to spatial math, including

Function	Description
`phi=ang4v(v1,v2,vn)`	Angle between two vectors
`s=side4v(v1,v2,vn)`	Calculate on which side of plane is given point
`xw=frame2world(xf,T,typ)`	Mapping from a given frame to world frame*
`xf=world2frame(xw,T,typ)`	Mapping from world frame to a given frame*
`T=normalize(T)`	Normalize homogenous, rotation matrix or vector
`x=xnormalize(x)`	Normalize quaternion part of pose array

Function	Description
`xm=xmean(x)`	Mean pose (mean position and quaternion mean r)
`R=wexp(w)`	Exponent of rotational velocity vector
`e=terr(T2,T1)`	Error between two transformation matrices
`Ti=tinv(T)`	Inverse of homogeneous transformation matrix
`w=rlog(R)`	Logarithm of a rotation matrix
`R=rexp(w)`	Exponent of rotation matrix (rotation velocity)
`Rd=rder(R)`	Rotation matrix derivative
`S=skew(v)`, `S=v2s(v)`	Map vector to skew-symmetric matrix operator performing cross product
`v=invskew(S)`, `v=s2v(s)`	Generate vector from skew-symmetric matrix
`deadzone(x,w,c)`	Dead-zone with width w and with center at c
`sigmoid(x,c,a)`	Sigmoid functionat center c and gain a
`y=smoothstep(x,xmin,xmax)`	Sigmoid-like interpolation and clamping function
`T=t42point_sets(p1,p2)`	Find rigid transformation matrix between two poses of a rigid object defined by two sets of points
`[dist,points]=dist2lines(p1,n1,p2,n2)`	Shortest distance between two lines defined by points and direction

*Only fixed frames are supported and transformations consider only rotation between frames for twists. For wrenches using option `typ=1` maps wrenches to other frame otherwise only rotation is considered.

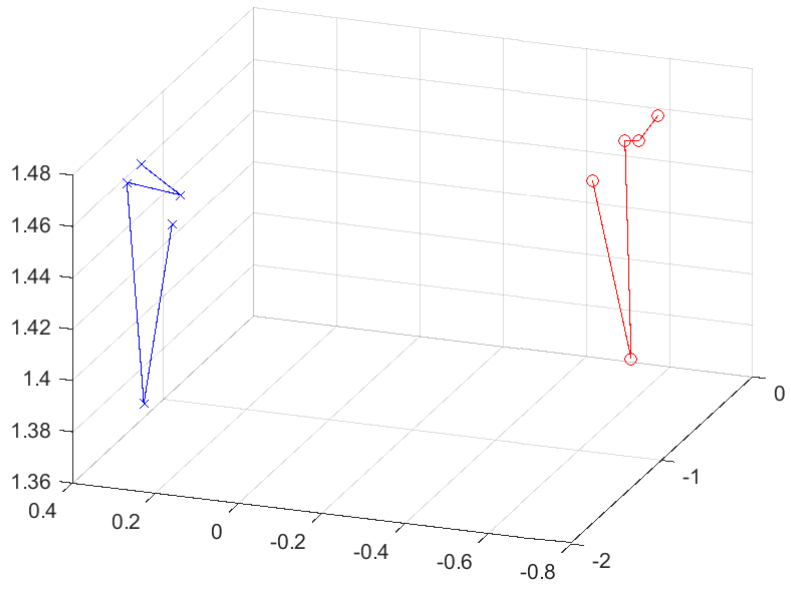
```
>> T=rp2t(rot_z(pi/4),[1 0 0]');
>> world2frame([1 1 0.5]',T)
ans =
    0.7071
    0.7071
    0.5000
```



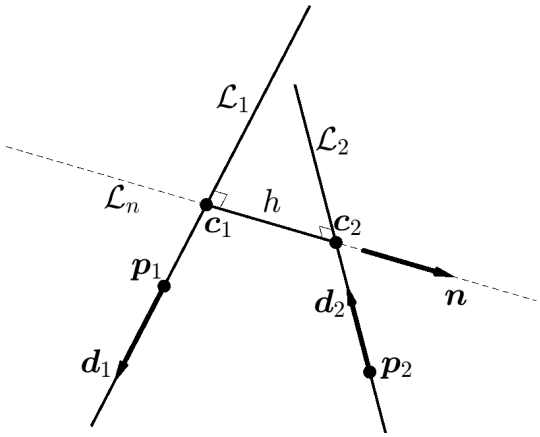

```
>> ang4v([0 0 1]',[1 0 0]')*180/pi
ans =
    90
>> T=rp2t(rot_z(pi/4),[1 0 0]');
>> world2frame([1 1 0.5]',T)
ans =
    0.7071
    0.7071
    0.5000
>> v2s([1 2 3])
ans =
     0    -3     2
     3     0    -1
    -2     1     0
>> v2s([1 2 3])*[2 -1 1]'
ans =
     5
     5
    -5
>> cross([1 2 3]',[2 -1 1]')
ans =
     5
     5
    -5
>> wexp([2 0 0]') % gives rotation produced by given velocity in 1 sec
ans =
    1.0000         0         0
         0   -0.4161   -0.9093
         0    0.9093   -0.4161
>> rot_x(2)
ans =
    1.0000         0         0
         0   -0.4161   -0.9093
         0    0.9093   -0.4161
```

Transformation between two poses of a body calculated using captured body points

```
>> % Body pose 1
>> o1p=[
>>    -1.5562    0.2572    1.4492
>>    -1.6330    0.3083    1.3808
>>    -1.5965    0.3571    1.4649
>>    -1.6991    0.2063    1.4663
>>    -1.7070    0.2998    1.4768];
>> %Body pose 2
>> o2p=[
>>    -0.2731   -0.4744    1.4389
>>    -0.2718   -0.5672    1.3712
>>    -0.3304   -0.5649    1.4582
>>    -0.1481   -0.5588    1.4520
>>    -0.2185   -0.6201    1.4654];
>>
>> figure(11)
>> plot3(o1p(:,1),o1p(:,2),o1p(:,3),'bx-')
>> hold on
>> plot3(o2p(:,1),o2p(:,2),o2p(:,3),'ro-')
>> hold off
>> grid
>> view([-70 30])
>> % Transformation
>> T=t42point_sets(o1p,o2p)
T =
   -0.5886   -0.8075    0.0386   -1.0380
    0.8083   -0.5887    0.0117    0.9184
    0.0133    0.0381    0.9992    0.0016
         0         0         0    1.0000
```



Calculate a shortest distance between two lines defined by points p1 and p2 and directions d1 and d2




```
>> p1=[0.25 0 -0.3]';
>> d1=[-0.5000 0 -0.8660]';
>> p2=[1.5 0.3 -0.8]';
>> d2=[-0.4890 0.5000 0.7148]';
>> [distance,closest_points]=dist2lines(p1,d1,p2,d2);
distance =
    0.9712
closest_points =
    0.5399         0    0.2022
    0.9935    0.8179   -0.0597
```

Interpolation functions

For interpolation in joint space we can use standard interpolation functions available in MATLAB, like `interp1`. Also for interpolation of 3D positions in Cartesian task space, we can use `interp1`. On the other hand, the interpolation of orientations is not so straightforward. In RBS Toolbox we use in most cases the linear spherical interpolation SLERP. Basic functions for SLERP interpolation are

Function	Description
`q=slerp(q1,q2,s)`, `q=qinterp(q1,q2,s)`	SLERP interpolation for quaternion inputs
`R=rinterp(R1,R2,s)`	SLERP interpolation for rotation matrix inputs

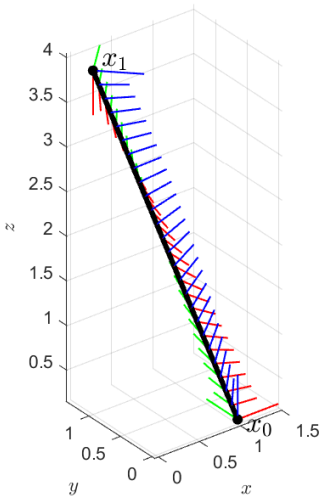
```
>> q1=r2q(rot_x(1));
>> q2=r2q(rot_z(2));
>> s=0:0.2:1;
>> qq=slerp(q1,q2,s)  % short-way rotation
qq =
    1.0000         0         0         0
    0.9969    0.0785         0         0
    0.9877    0.1564         0         0
    0.9724    0.2334         0         0
    0.9511    0.3090         0         0
    0.9239    0.3827         0         0
>> qq=slerp(q1,q2,-s)  % long-way rotation
    1.0000         0         0         0
```

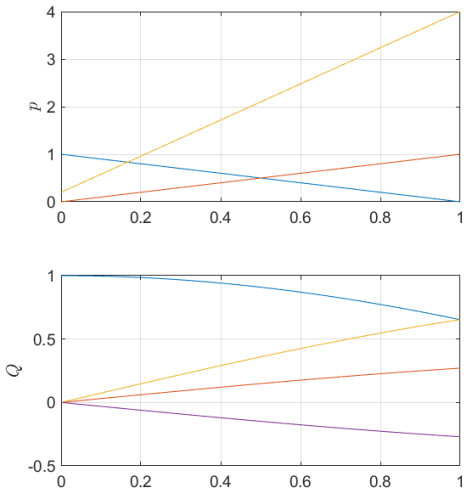
```
    0.8526   -0.5225         0         0
    0.4540   -0.8910         0         0
   -0.0785   -0.9969         0         0
   -0.5878   -0.8090         0         0
   -0.9239   -0.3827         0         0
```

Interpolating between two spatial poses we combine linear interpolation for positions and SLERP interpolation for orientations. Functions for interpolation are

Function	Description
`x=xinterp(xa,xb,s)`	Linear interpolation of spatial pose (position+quaternion)
`T=tinterp(Ta,Tb,s)`	Linear interpolation for homogenous matrices
`x=xarcinterp(xa,xb,s)`	Linear interpolation on arc for position and SLERP for orientation (position+quaternion)
`T=tarcinterp(Ta,Tb,pC,s)`	Linear interpolation on arc for position and SLERP for orientation (homogenous matrices)

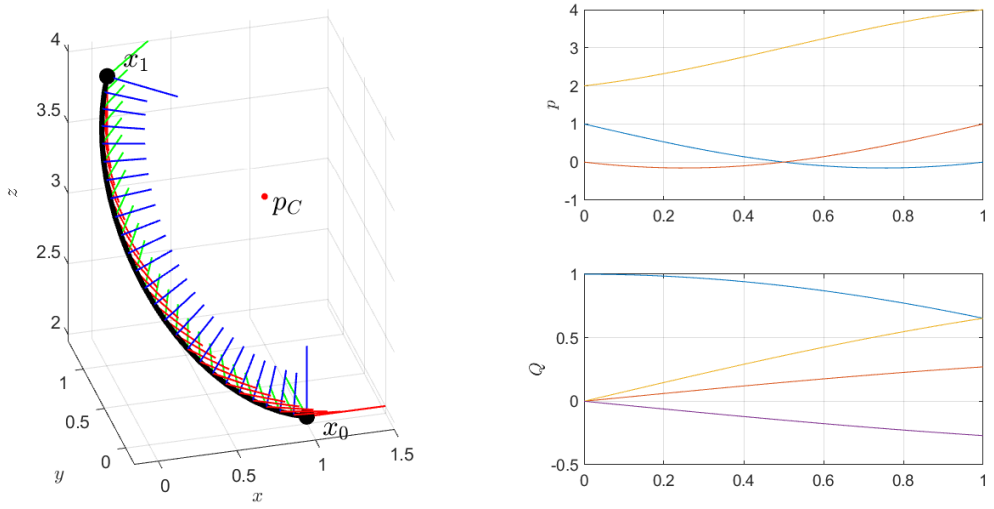
```
>> x0=t2x(rp2t(eye(3),[1 0 0.2]'));
>> x1=t2x(rp2t(rot_y(pi/2)*rot_x(pi/4),[0 1 4]'));
>> s=0:0.2:1;
>> xx=xinterp(x0,x1,s)
xx =
    1.0000         0    0.2000    1.0000         0         0         0
    0.8000    0.2000    0.9600    0.9853    0.0611    0.1475   -0.0611
    0.6000    0.4000    1.7200    0.9416    0.1204    0.2906   -0.1204
    0.4000    0.6000    2.4800    0.8701    0.1761    0.4252   -0.1761
    0.2000    0.8000    3.2400    0.7731    0.2267    0.5473   -0.2267
         0    1.0000    4.0000    0.6533    0.2706    0.6533   -0.2706
```





We can interpolate positions also on an arc

```
>> x0=[1 0 2 1 0 0 0];
>> x1=[0 1 4 r2q(rot_y(pi/2)*rot_x(pi/4))];
>> pC=[1 1 3]';
>> s=0:0.2:1;
>> xx=xarcinterp(x0,x1,pC,s)
xx =
    1.0000         0    2.0000    1.0000         0         0         0
    0.5303   -0.1484    2.3213    0.9853    0.0611    0.1475   -0.0611
    0.1419   -0.0982    2.7599    0.9416    0.1204    0.2906   -0.1204
   -0.0982    0.1419    3.2401    0.8701    0.1761    0.4252   -0.1761
   -0.1484    0.5303    3.6787    0.7731    0.2267    0.5473   -0.2267
   -0.0000    1.0000    4.0000    0.6533    0.2706    0.6533   -0.2706
```



Trajectory generation

One the important functionality in robotics toolboxes is generation of paths and trajectories. RBS Toolbox provides tools to generate paths and trajectories in joint and task space. Trajectory generation is based on interpolation. The difference between the interpolation and trajectory generation is that we define the time for the movement from the initial position to the final position and the velocity profile.

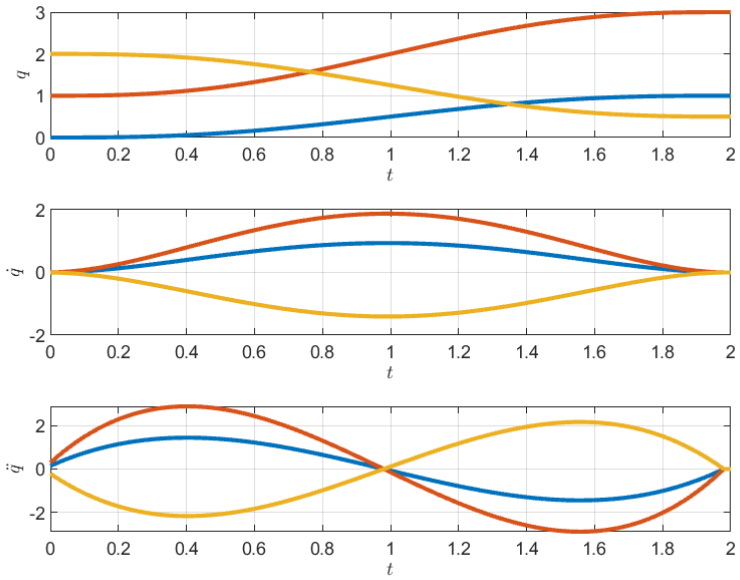
Joint space trajectories

There are three functions for trajectory generation in joint space.

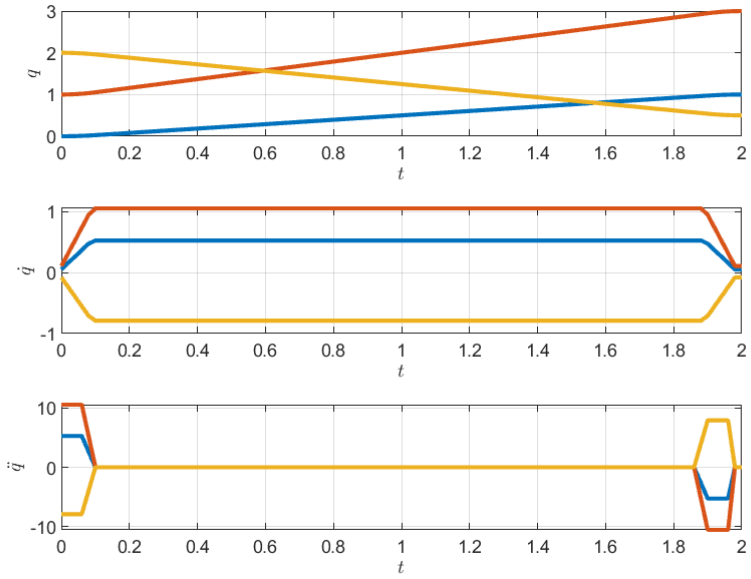
Function	Description
`[qt,qdt,qddt]=jpoly(q0,q1,t)`, `[qt,qdt,qddt]=jtraj(q0,q1,t)`	Trajectory based on 5th order polynomial (allows also to select initial and final velocities)
`[qt,qdt,qddt]=jtrap(q0,q1,t,ta)`	Trajectory with trapezoidal velocity profile
`[qt,qdt,qddt]=jline(q0,q1,t)`	Trajectory with constant velocity

They generate a sequence of interpolated joint positions, velocities and accelerations for motion from initial configuration `q0` to final configuration `q1` for instances defined by sequence `t`.

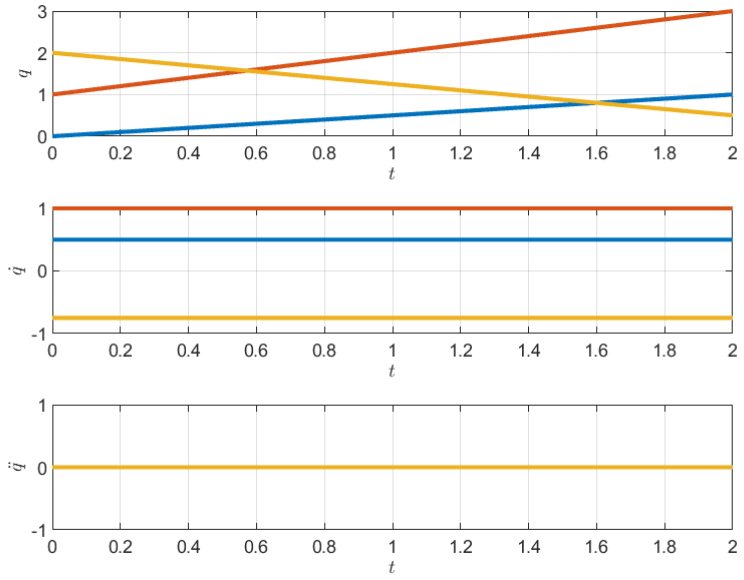
```
t=0:0.02:2;
[qt,qdt,qddt]=jtraj([0 1 2]',[1 3 0.5]',t);
```




```
t=0:0.02:2;
[qt,qdt,qddt]=jtrap([0 1 2]',[1 3 0.5]',t);
```




```
t=0:0.02:2;
[qt,qdt,qddt]=jline([0 1 2]',[1 3 0.5]',t);
```



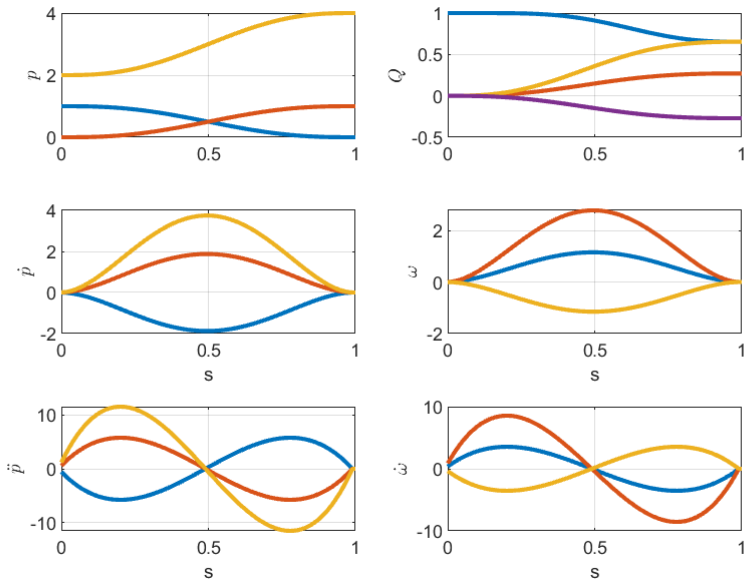
Task space trajectories

There are three basic functions for trajectory generation in task space.

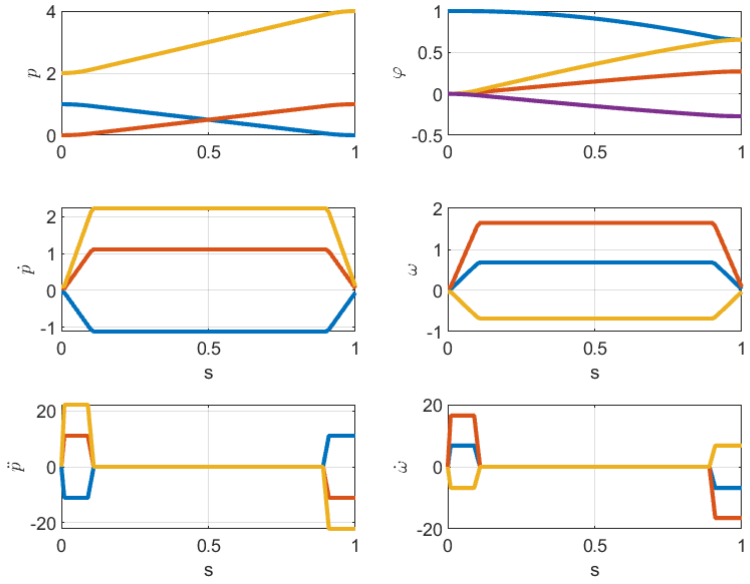
Function	Description
`[xt,vt,at]=cpoly(T0,T1,t)`	Cartesian trajectory based on 5th order polynomial
`[xt,vt,at]=ctrap(T0,T1,t,ta)`	Cartesian trajectory with trapezotidal velocity profile
`[xt,vt,at]=cline(T0,T1,t)`	Cartesian trajectory with constant velocity
`[xt,vt,at]=ctraj(T0,T1,t,fun)`	Cartesian trajectory with custom velocity profile
`[xt,vt,at]=carc(x0,x1,pC,t,fun)`	Cartesian trajectory trajectory for motion on an arc with custom velocity profile

They generate a sequence of interpolated positions and orientations, velocities and accelerations for motion from initial pose `x0` to final pose `x1` for instances defined by sequence `t`. All functions use linear interpolation for positions and SLERP for orientations.

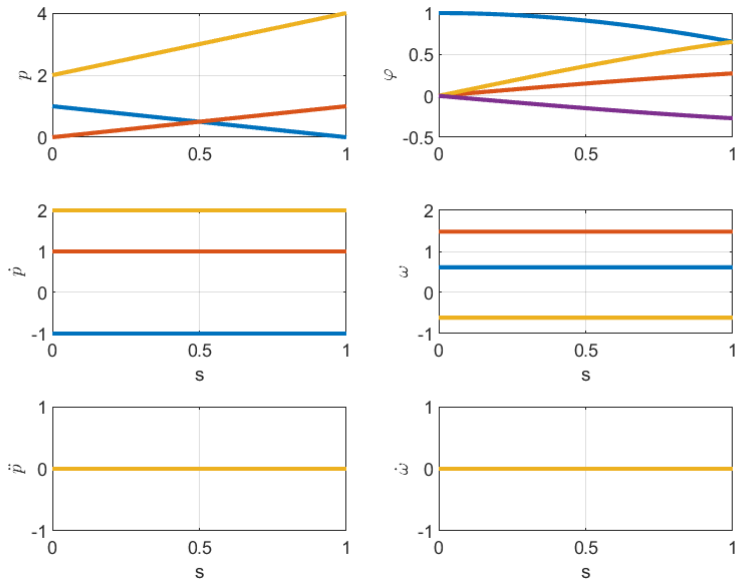
```
T0=rp2t(eye(3),[1 0 2]');
T1=rp2t(rot_y(pi/2)*rot_x(pi/4),[0 1 4]');
t=0:0.01:1;
[xt,vt,at]=cpoly(t2x(T0),t2x(T1),t);
```




```
[xt,vt,at]=ctrap(T0,T1,t);
```




```
[xt,vt,at]=cline(T0,T1,t);
```



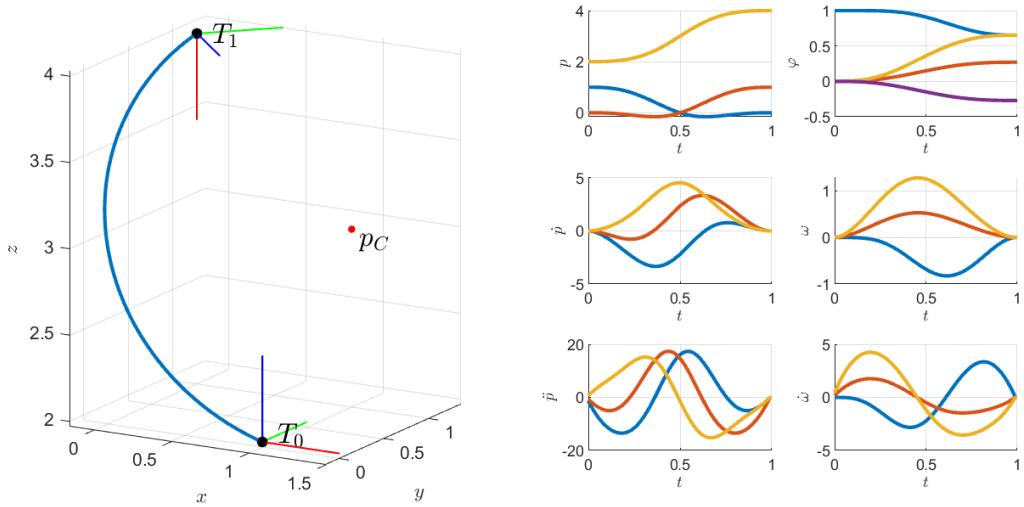
Additionally, there is a function `ctraj(T0,T1,t,fun)`, where the velocity profile can be defined as an input parameter. This allows to define custom function for motion profile. For example

```
[xt,vt,at]=ctraj(t2x(T0),t2x(T1),t,@jpoly);
[xt,vt,at]=cpoly(t2x(T0),t2x(T1),t);
```

generate the same trajectory.

Using function `carc` we can define a trajectory for motion on an arc, where we can select the velocity profile in the same way as for `ctraj`.

```
T0=rp2t(eye(3),[1 0 2]');
T1=rp2t(rot_y(pi/2)*rot_x(pi/4),[0 1 4]');
t=0:0.01:1;
pC=[1 1 3]';
[xt,vt]=carc(T0,T1,pC,t,@jpoly);
```



Path generation and interpolation

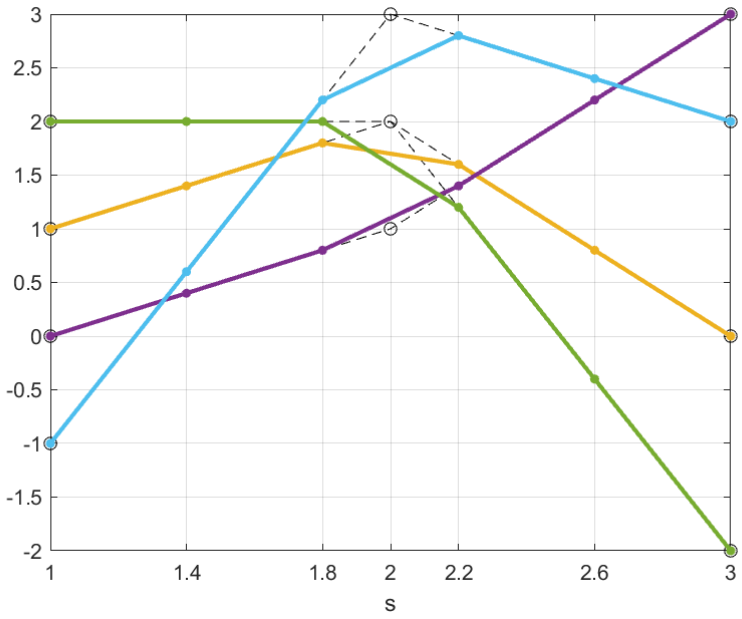
With the trajectory generation functions we can generate motion between two positions, where the motion starts in the initial position and stops in the final position, i.e. initial and final velocities are 0. To generate a more complex motion, like continuous motion over or near many points, RBS Toolbox provides several functions to work with paths

Function	Description
`xnew=interpPath(s,x,snew)`	Interpolate path for query path values
`xnew=interpCartesianPath(s,x,snew)`	Interpolate Cartesian path for query path values
`xnew=interpQuaternionPath(s,Q,snew)`	Interpolate quaternion path for query path values
`x=uniqueCartesianPath(x)`	Correct the quaternions in cartesian path so that they are unique (dot product of consecutive quaternions is positive)
`Q=uniqueQuaternionPath(Q)`	Correct the quaternion path so that they are unique (dot product of consecutive quaternions is positive)
`[xi,si]=pathoverpoints(points,...)`	Generates path over points using spline interpolation
`auxpnt=pathauxpointspoints(points,...)`	Generates auxiliary points for path points
`[send,si]=pathlen(path,scale)`	Calculates path length using positions or orientations or both
`[px,d]=distance2line(p0,p,dir)`	Find the closest point on line and calculate distance
`[px,d,sx]=distance2path(x,path,s,scale)`	Find the closest point on path and calculate distance
`qout=qspline(q,s,mode)`	Spline interpolation of quaternions in spherical space
`xd=gradientPath(x,t)`	Calculate derivative along path
`xd=gradientCartesianPath(x,t)`	Calculate derivative along Cartesian path
`Qd=gradientQuaternionPath(Q,t)`	Calculate gradient along quaternion path

Path interpolation

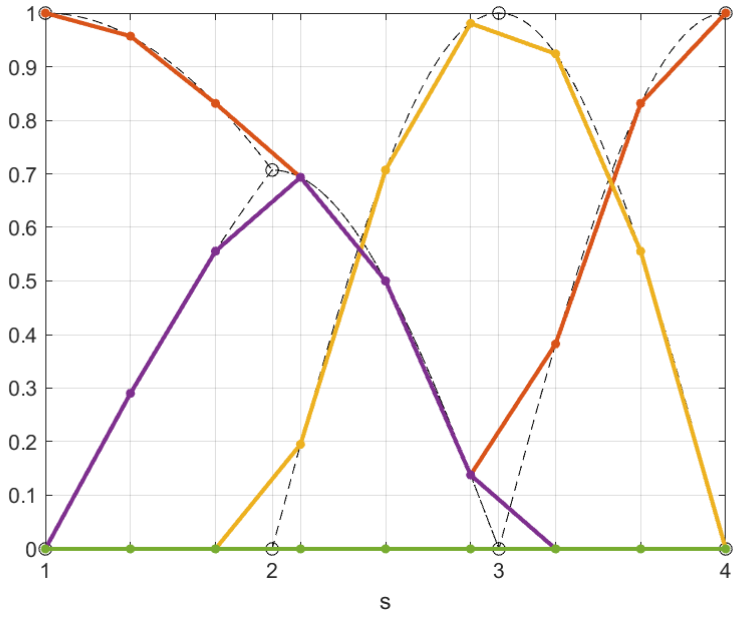
Path interpolation can be done for joint and task space paths. In case of joint space paths, we use function `interpPath`, which is a wrapping function for standard linear interpolation.

```
>> qpath=[1 0 2 -1;2 1 2 3;0 3 -2 2]
>> s=(1:3)';
>> si=linspace(1,3,6)';
>> qi=interpPath(s,qpath,si)
qi =
    1.0000         0    2.0000   -1.0000
    1.4000    0.4000    2.0000    0.6000
    1.8000    0.8000    2.0000    2.2000
    1.6000    1.4000    1.2000    2.8000
    0.8000    2.2000   -0.4000    2.4000
         0    3.0000   -2.0000    2.0000
```



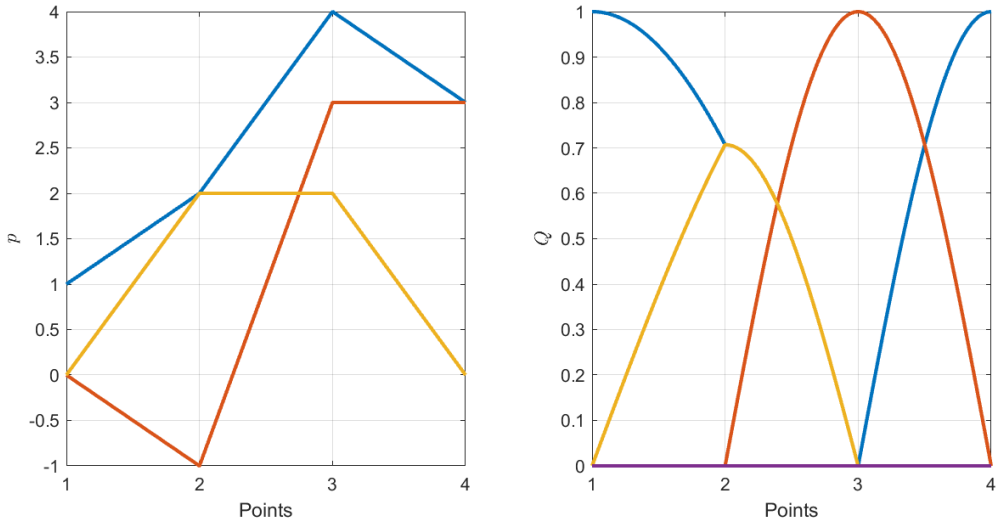
Path interpolation for quaternion paths is wrapper for SLERP interpolation.

```
>> q=[ 1.0000         0         0         0
    0.7071         0    0.7071         0
    0.0000    1.0000   -0.0000         0
    1.0000         0         0         0];
>> s=(1:4)';
>> si=linspace(1,4,9)';
>> qi=interpQuaternionPath(s,q,si)
qi =
    1.0000         0         0         0
    0.9569         0    0.2903         0
    0.8315         0    0.5556         0
    0.6935    0.1951    0.6935         0
    0.5000    0.7071    0.5000         0
    0.1379    0.9808    0.1379         0
    0.3827    0.9239         0         0
    0.8315    0.5556         0         0
    1.0000         0         0         0
```



Cartesian path interpolation is a combination of linear interpolation for positions and SPLINE interpolation.

```
x=[ 1  0 0 1.0000         0         0         0
    2 -1 2 0.7071         0    0.7071         0
    4  3 2 0.0000    1.0000   -0.0000         0
    3  3 0 1.0000         0         0         0];
s=(1:4)';
si=linspace(1,4,100)';
xi=interpCartesianPath(s,x,si)
```

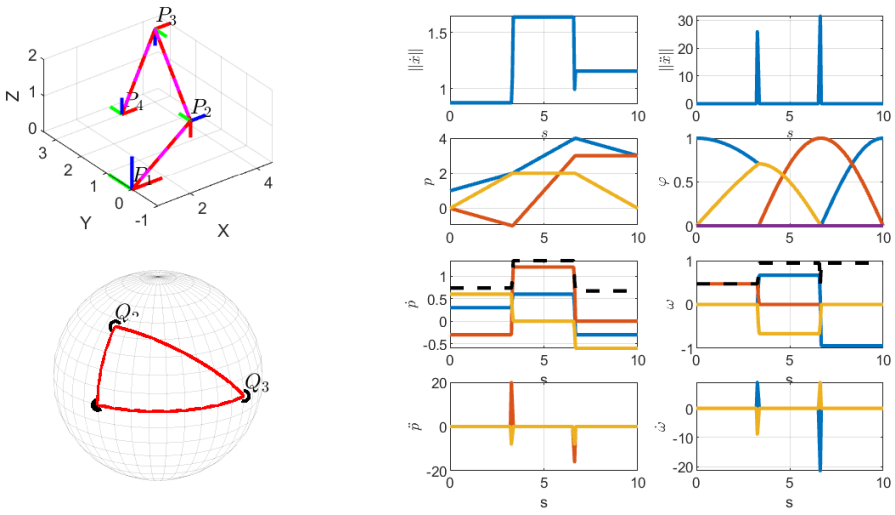


Path gradient and derivatives

To calculate gradient along the path use functions `gradientPath` for joint space paths and for pure position paths in task space. If orientations are included use `gradientQuaternionPath` or `gradientCartesianPath`.

Velocities can be calculated if the independent variable is time, i.e. we generate a trajectory. For that, the independent variables (`s` and `t` in the example) must have the same initial and final value.

```
>> x=[ 1  0 0 1.0000         0         0         0
>>     2 -1 2 0.7071         0    0.7071         0
>>     4  3 2 0.0000    1.0000   -0.0000         0
>>     3  3 0 1.0000         0         0         0];
>> tmax=10;
>> s=linspace(0,tmax,4)';
>> t=linspace(0,tmax,200)';
>> xt=interpCartesianPath(s,x,t);
>> vt=gradientCartesianPath(xt,t);
>> at=gradientPath(vt,t);
```

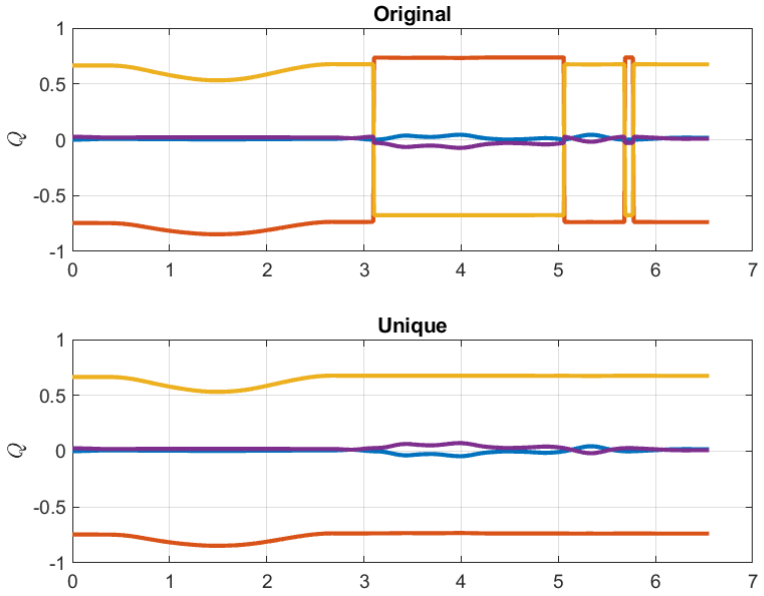


> Note that for the calculation of path acceleration, we always use `gradientPath` as rotational velocities form a vector space.

Unique quaternion paths

Because $Q = -Q$, there are always two quaternions representing the same orientation. Therefore, when a task-space paths are represented using quaternions it may happen that the quaternions are switching along path form one solution to another. This may cause problems. To change the path representations so that quaternions are using unique solution for representing orientations, RBS Toolbox provides two functions `uniqueQuaternionPath(Q)` and `uniqueCartesianPath(x)` that make quaternions unique, i.e. they change sequence of quaternions in such a way, that the dot product between two consecutive quaternions keeps the sign.

```
>> load('sample_traj.mat')
>> subplot(2,1,1)
>> h=plotcpos_ori(time,cart_traj,'Graph','time','Type','Ori');
>> title('Original')
>> cart_traj_u=uniqueCartesianPath(cart_traj); % correction
>> subplot(2,1,2)
>> h=plotcpos_ori(time,cart_traj_u,'Graph','time','Type','Ori');
>> title('Unique')
```



Path via multiple points

Using function `pathoverpoints` we can generate smooth trajectories from the initial point via or near intermediate points to the final point. This function has several options defining the path generation.

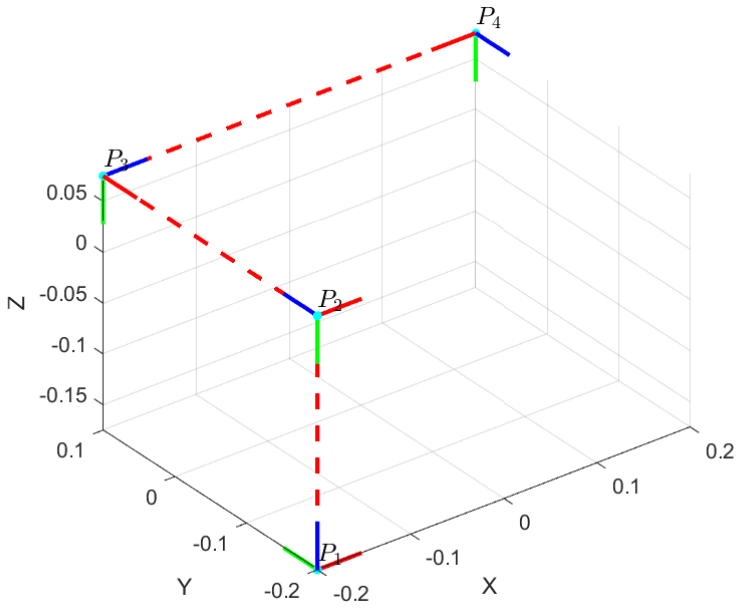
Option	Values	Description
`Interp`	`'inner'`, `'spline'`, `'none'`	Interpolation type: `'inner'` spline by uniform subdivision `'spline'` cubic spline curve `'none'` no interpolation
`Step`	0.01	Maximal difference in path parameter
`Npoints`	0	Minimal number of path points - if 0 then Step is used
`Auxpoints`	`'absolute'`, `'relative'`, `'none'`	Auxiliary points: `'absolute'` Auxdist parameter is absolute distance `'relative'` Auxdist parameter is relative segment distance `'none'` No Auxpoints
`Auxdist`	[0.1 0.2]	distance of auxiliary points `[position orientation]`
`Order`	4	Order of inner spline (>=3)
`Natural`	`'on'`, `'off'`	Make path parameter natural (=path distance)
`NormScale`	0	Scaling factor for rotation norm
`Plot`	`'on'`, `'off'`	Generate plots

Option	Values	Description
`Figure`	1	Figure handle

*Defualts are in bold.

For example, let generate a smooth path over the following points

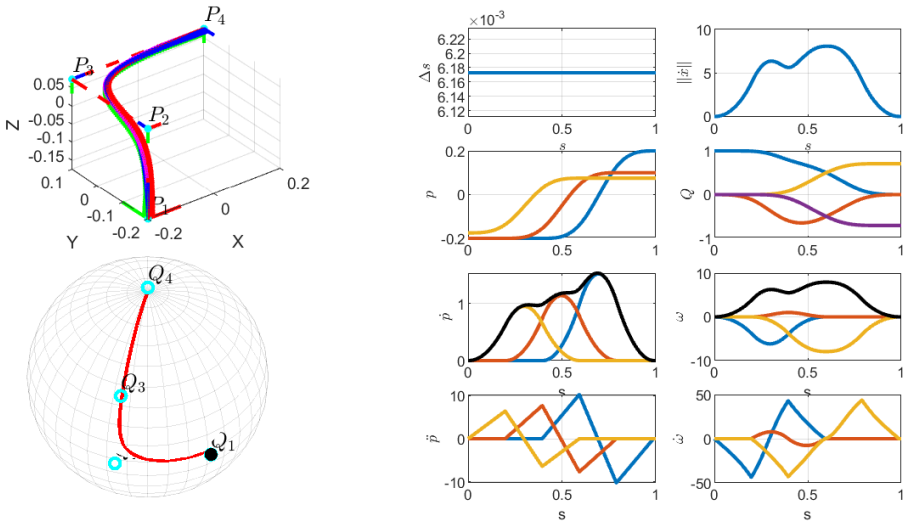
```
pte=[-0.2 -0.2 -0.175   0     0     0;...
    -0.2 -0.2  0.075    0     0   -pi/2;...
    -0.2  0.1  0.075 -pi/2    0   -pi/2;...
     0.2  0.1  0.075 -pi      0   -pi/2];
pt=prpy2x(pte);
plotpathpoints(pt)
```



By changing the options we get different paths. First, we generate a path using `'inner'` spline with `'Order'=4` without auxiliary points

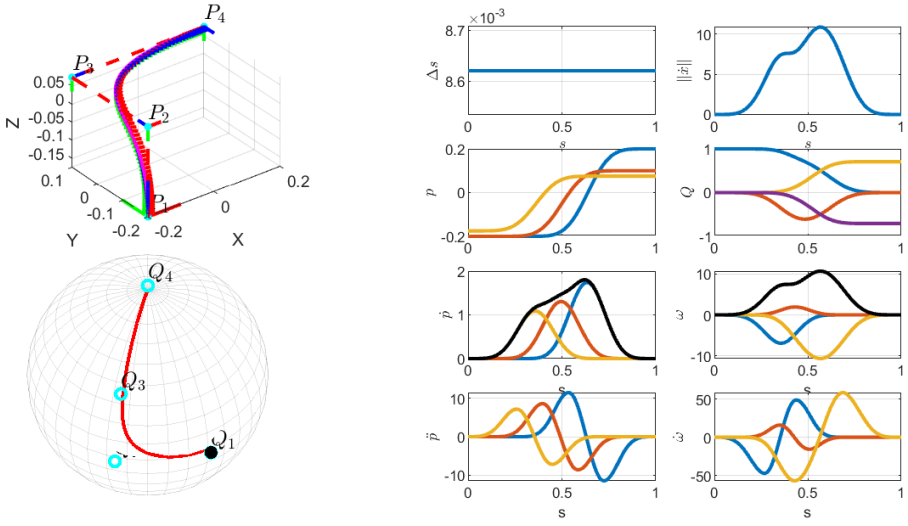
```
xi=pathoverpoints(pt,'Step',0.01,'Order',4,'Plot','on');
```

The generated path does not go over intermediate points, but approaches them (how close depends on parameter `'Order'`).



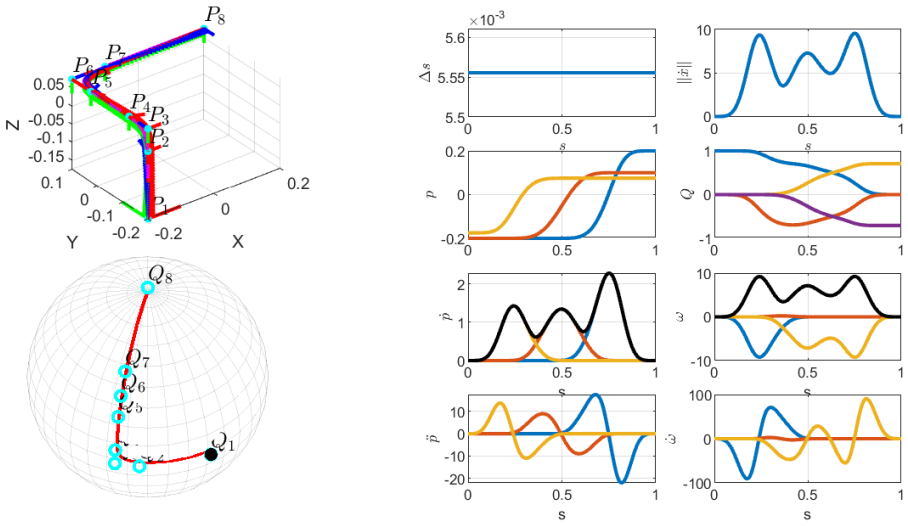
Note that the generated accelerations are not smooth. By increasing interpolation order, the generated path becomes smoother, but also the distance to intermediate points is greater.

```
xi=pathoverpoints(pt,'Step',0.01,'Order',6,'Plot','on');
```



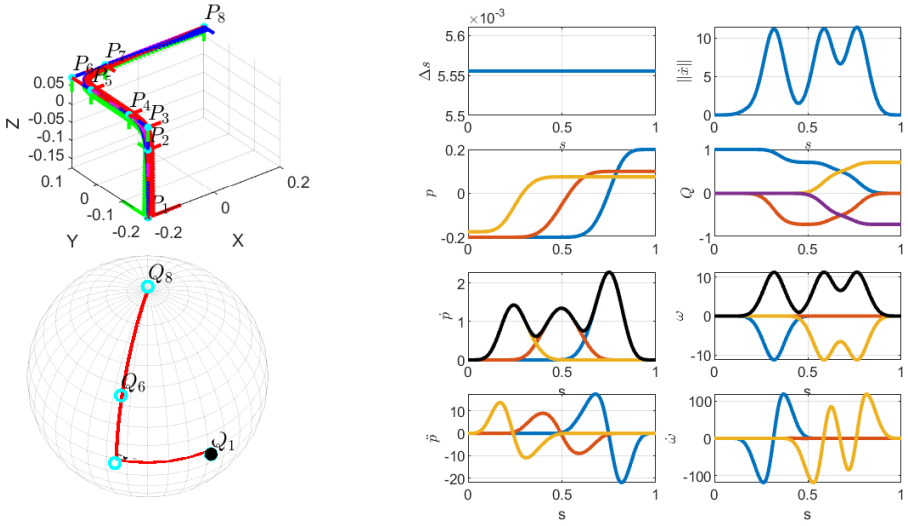
If we want that the path goes closer to the intermediate points, we can add auxiliary points. In this case we have added relative auxiliary points for position and rotation.

```
xi=pathoverpoints(pt,'Step',0.01,'Order',6,'Auxpoints','relative','Auxdist',0.25,
'Plot','on');
```



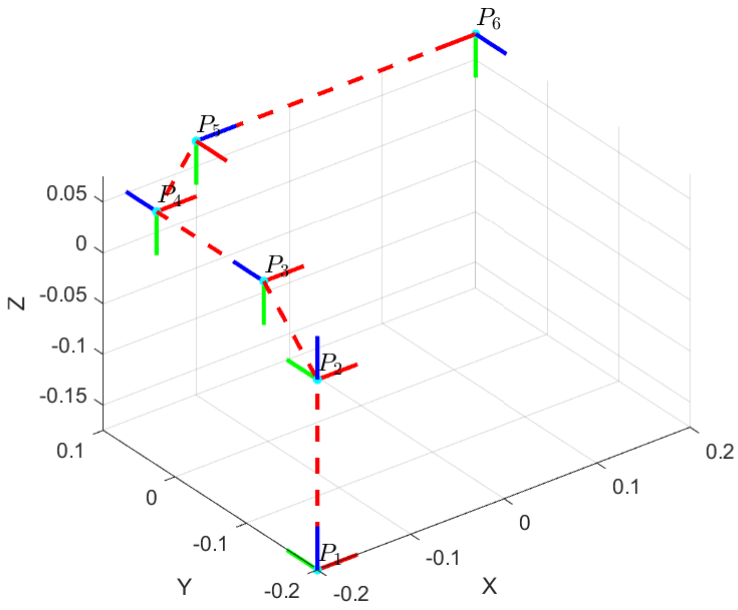
In this case we have added only auxiliary points for position, rotation in these auxiliary points is as on the other side of a segment where the auxiliary position point is added.

```
xi=pathoverpoints(pt,'Step',0.01,'Order',6,'Auxpoints','relative','Auxdist',[0.25
0],'Plot','on');
```



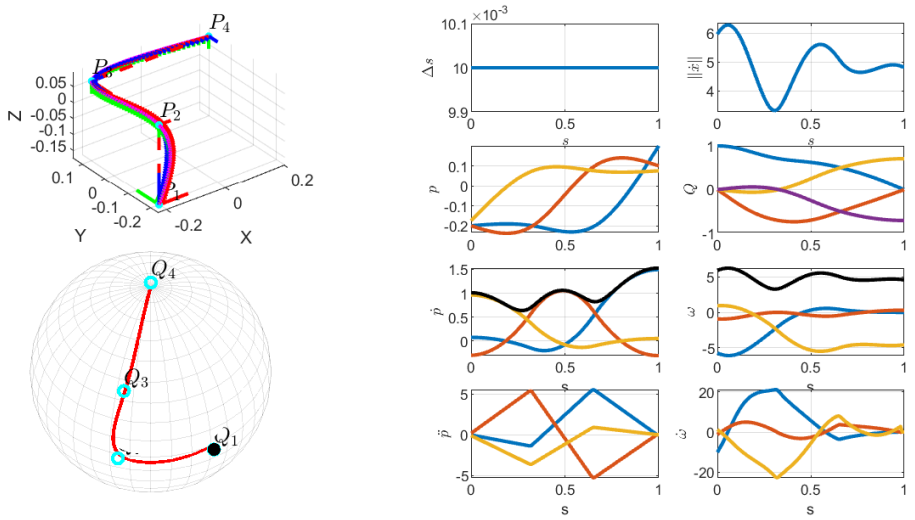
Using function `pathauxpoints` the generation is almost the same, except that this function has an additional option `'Viapoints'`, which allows to use for path generation only auxiliary points, while the intermediate points (via) points are neglected.

```
apt=pathauxpoints(pt,'Auxpoints','relative','Auxdist',[0.25
0],'Viapoints','off');
plotpathpoints(apt)
```



On the other hand, using `'spline'` interpolation the generated path passes the intermediate points

```
xi=pathoverpoints(pt,'Step',0.01,'Interp','spline','Plot','on');
```



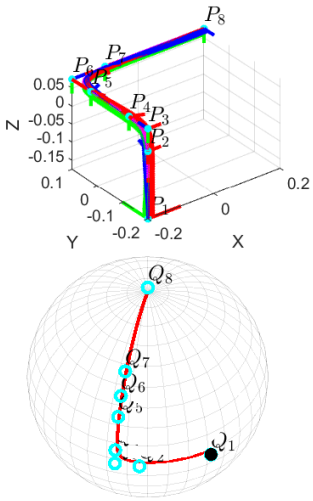
Until now the path samples have been equidistantly distributed over the whole path with `s=0` at start and `s=1` in the final point. Consequently, we can observe that changes between samples (velocities $\|\dot{x}\|$) were not constant. Selection option `'Natural'='on'` the path parameter is defined by the path length

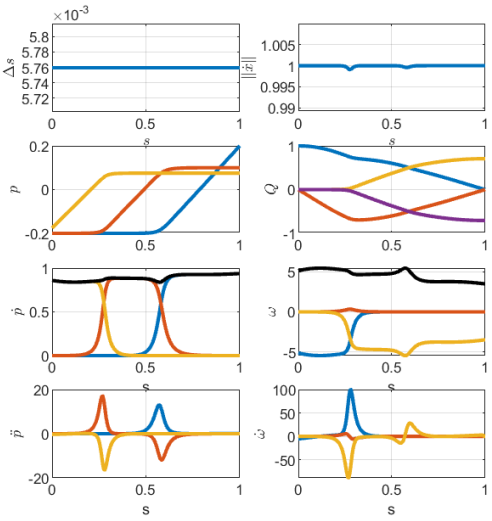
$$s(k) = \sum_{i=2}^{k} \sqrt{\|p_k - p_{k-1}\|^2 + a^2 \|2\log(Q_k * Q_{k-1}^{-1})\|^2}$$

where p_k are the positions and Q_k the orientations of generated path points. Parameter a is the scaling factor `'NormScal'`. Using natural path parameter assures that weighted norm $\|\dot{x}\|_a = 1$.

For our example, a good choice is `For a=0.1`.

```
[xi,si]=pathoverpoints(pt,'Step',0.01,'Natural','on','NormScale',0.1,'Order',6,'A
uxpoints','relative','Auxdist',0.25,'Plot','on');
```





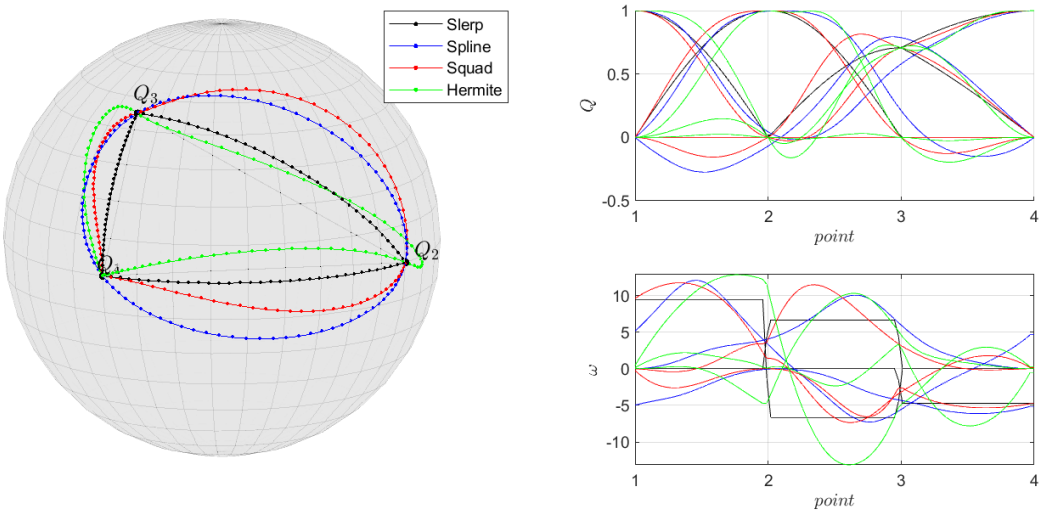
Orientation interpolation

Note that in function `pathoverpoints` the interpolation of orientation is not based on SLERP. If for some reason a specific interpolation of orientation is required, then it is better to use function `qspline`, which allows two types of interpolation

Parameter	Values	Comment
`mode`	`'hermite_cubic'`, `'squad'`	`'hermite_cubic'` cubic curve using Bezier-Bernstein basis `'squad'` spherical and quaternion angle interpolation. Analogous to bilinear interpolation in euclidean space.

The differences between different interpolation methods for quaternioms are seen in the following example

```
q=[ 1.0000         0         0         0
    0.0000    1.0000   -0.0000         0
    0.7071         0    0.7071         0
    1.0000         0         0         0];
s=linspace(0,1,4)';
t=(0:0.01:1)';
q_slerp=interpQuaternionPath(s,q,t);
[xi,ti]=pathoverpoints([zeros(4,3) q],'Interp','spline');
q_spline=xi(:,4:7);
q_squad=qspline(q,t,'squad');
q_hermite=qspline(q,t,'hermite_cubic');
```



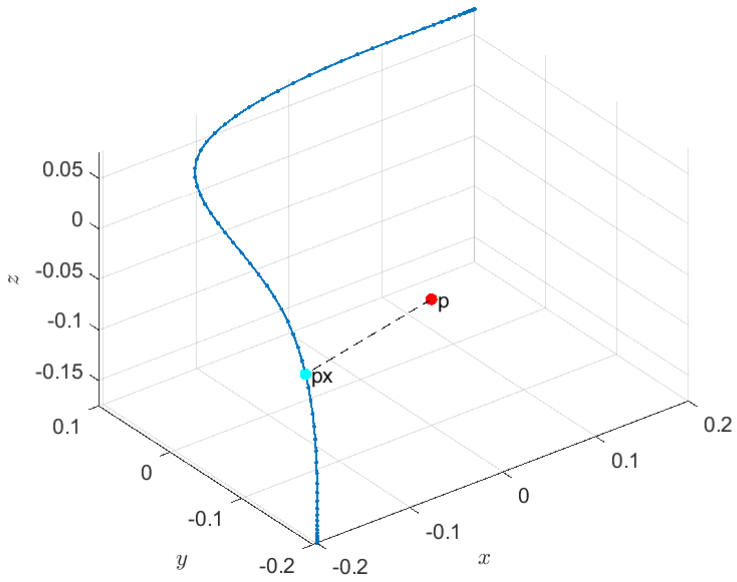
Path utilities

Utility functions are used to get some information about paths like path length or distance to path. The path length is calculated as a weighted sum of distances between path points

$$s(k) = \sum_{i=2}^{k} \sqrt{s_p^2 \|p_k - p_{k-1}\|^2 + s_Q^2 \|2\log(Q_k * Q_{k-1}^{-1})\|^2}$$

where p_k are the positions and Q_k the orientations of generated path points, and s_p and s_Q are weights for position and orientation distances.

```
>> pte=[-0.2 -0.2 -0.175   0     0     0;...
    -0.2 -0.2  0.075   0     0   -pi/2;...
    -0.2  0.1  0.075 -pi/2   0   -pi/2;...
     0.2  0.1  0.075 -pi     0   -pi/2];
>> pt=prpy2x(pte);
>> [path,s]=pathoverpoints(pt,'Npoints',51);
>> len=pathlen(path,[1 0])  % path length considering only positions
len =
    0.7906
>> p=[0 -0.1 -0.05]';
>> [px,d,sx]=distance2path(p,path(:,1:3),s) % closest point on path and distance
px =
   -0.2000
   -0.1839
   -0.0153
d =
    0.2196
sx =
    0.3415
>> plotcpos_ori(s,path,'Type','Pos','Graph','3D','LineWidth',1,'Marker','.')
>> plotpoint(p,0.005,'Color','r')
>> text(p(1),p(2),p(3),' p')
>> plotpoint(px,0.005,'Color','c')
>> text(px(1),px(2),px(3),' px')
>> plotline(p,px,'LineStyle','--','Color','k')
```

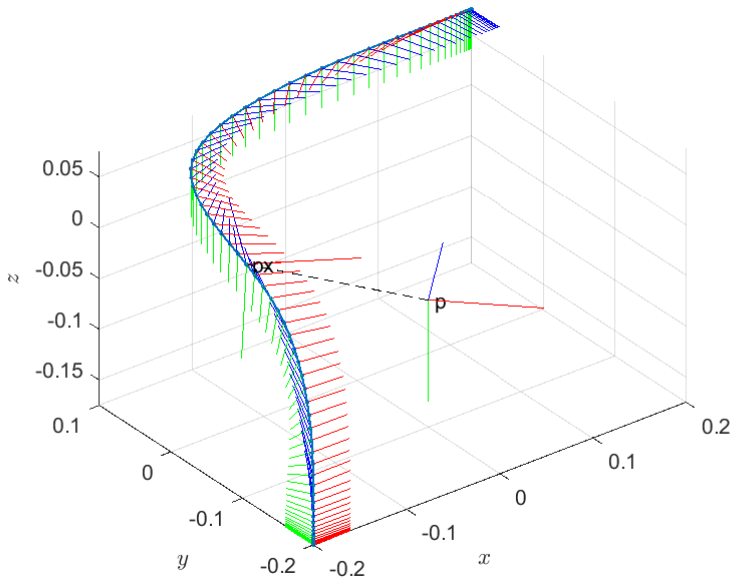


Considering also orientations the distance is defined as

$$d = \sqrt{s_p^2 \|p_e\|^2 + s_Q^2 \|Q_e\|^2}$$

where p_e and Q_e are position and orientation errors between the point `p` and a point on path `px`, and s is a weighting factor for orientatios.

```
>> p=prpy2x([0 -0.1 -0.05 -pi/4   0   -pi/2]);
>> [px,d,sx]=distance2path(p,path,s,[1 0.1]) % closest point on path and distance
px =
   -0.1990   -0.1048    0.0575    0.7225   -0.6461    0.1741   -0.1741
d =
    0.2283
sx =
    0.4512
>>
plotcpos_ori(s,path,'Type','Pos','Graph','3D','LineWidth',1,'Marker','.','UCS','o
n')
>> plotucs(p,0.005,'UCSLength',0.1,'Color','r')
>> text(p(1),p(2),p(3),' p')
>> plotucs(px,'UCSLength',0.1,'Color','c')
>> text(px(1),px(2),px(3),' px')
>> plotline(p,px,'LineStyle','--','Color','k')
```



Path approximations

Radial basis function (RBF)

If a path or trajectory is obtained by capturing motion of a robot, e.q. by using kinestetic guidance, it is possible to approximate the captured path by using radial basis functions (RBF). Among many possible radial basis functions we have selected ones with Gaussian kernels defined as

$$\Psi(x) = e^{-\frac{(x-c)^2}{2\sigma}}$$

which is centred at c and h is defining the width of the kernel function. An important feature of this kernel function is that it expresses $\mathcal{C}^\infty$ continuity.

The approximation function f of captured data $\mathbf{y}$ can be defined as a linear combination of weighted radial basis functions. We use a normalized version of this function, which yields

$$f(x) = \frac{\sum_{j=1}^{m} w_j \Psi_j(x)}{\sum_{j=1}^{m} \Psi_j(x)}$$

where m is the number of kernel functions. The kernel functions are centred at c_i. Using $\mathbf{\Phi}$ defined as a row vector with components

$$\mathbf{\Phi}_k(x) = \frac{\Psi_k(x)}{\sum_{j=1}^{m} \Psi_j(x)}$$

yields

$$f(x) = \mathbf{\Phi}(x)\mathbf{w}$$

where $\mathbf{w}$ is a vector with elements w_j. Applying this to the given data set of points we obtain a set of linear independent equations

$$\mathbf{A}\mathbf{w} = \mathbf{y}$$

where $\mathbf{y}$ is a vector of captured data with elements y_i, and $\mathbf{A}$ is a matrix with rows $\mathbf{\Phi}(x_i)$. The corresponding weights $\mathbf{w}$ can be found as

$$\mathbf{w} = \mathbf{A}^{-1}\mathbf{y}$$

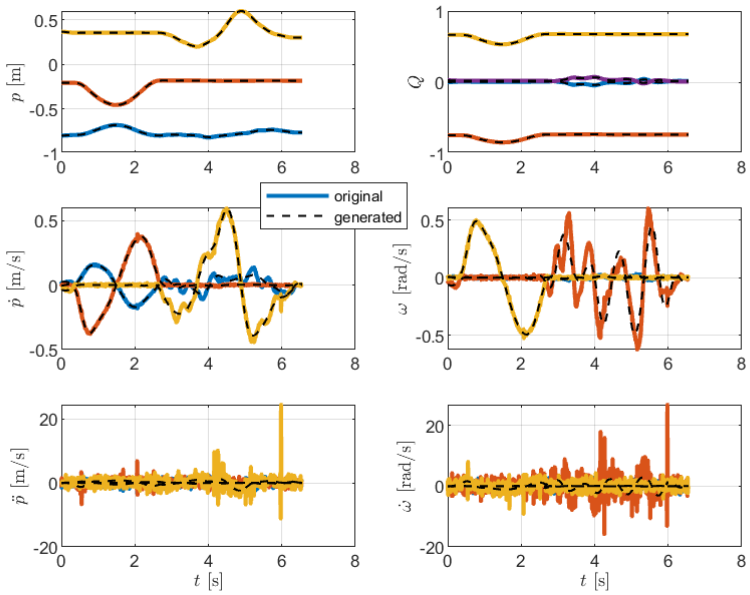
The functions to work with RBF approximations are

Function	Description
`[RBF,A]=encodeRBF(x,y,N,c,sigma2,bc,coff,sfac)`	Calculate weights for Gaussian radial basis functions optionally allowing also specification of initial and final velocities and accelerations
`[y,ydot,yddot,ydddot,A,Ad,Add,Addd]=decodeRBF(x,RBF)`	Calculate path in point x using Gaussian radial basis functions
`[y,ydot,yddot]=decodeCartesianRBF(x,RBF)`	Calculate Cartesian path in point x using Gaussian radial basis functions
`[q,w,wdot]=decodeQuaternionRBF(x,RBF)`	Calculate quaternion path in point x using Gaussian radial basis functions
`[J,Jdot]=jacobiRBF(x,RBF)`	Calculates the Jacobian and its derivative using numeric differentiation
`[y,RBF]=updateRBF(x,y,RBF)`	Update weights for (RBF) using recursive regression

The RBF encoding for quaternion data is done for each component independently, but RBF decoding assures that returned quaternions are unit quaternions.

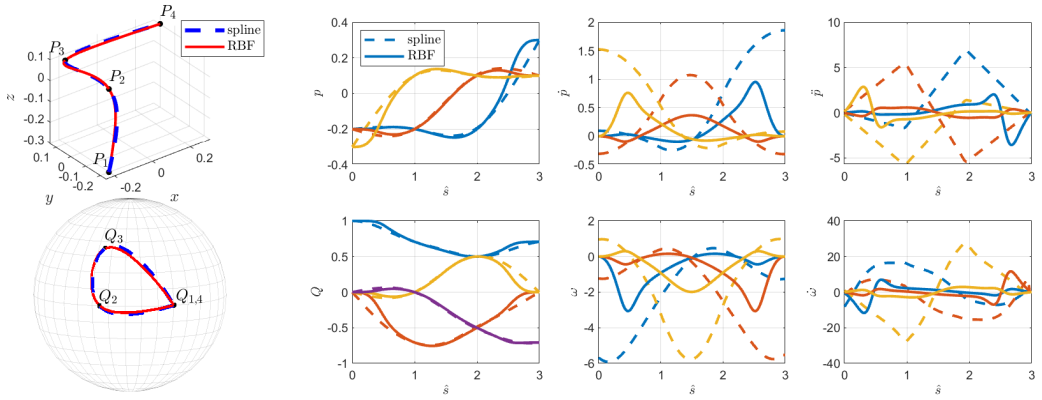
For example, captured Cartesian path of a robot end-effector can be approximated using RBF using `encodeRBF` and `decodeCartesianRBF` functions

```
load('sample_traj.mat','cart_traj','time');
x=uniqueCartesianPath(cart_traj);
RBF=encodeRBF(time,x,25);  % other RBF parameters (c and sigma) have default
values
[x_rbf,v_rbf,a_rbf]=decodeCartesianRBF(time,RBF);
```



We can use RBF parametrization path generation over multiple points. Compared to spline interpolation we ca define in the case of RBF also initial and final conditions.

```
points=[-0.2 -0.2  -0.30   0       0     0;...
    -0.2 -0.2   0.1   0       0   -pi/2;...
    -0.2  0.1   0.1 -pi/2     0   -pi/2;...%    0.1   0.1   0.075   0    -pi/2
0;...
    0.3   0.1   0.1 -pi/2     0   0;...%    0.3   0.1   0.1  0     pi/2    0;...
    ]; %    ;...
points=prpy2x(points);
[N,m]=size(points);
tmax=3;
dt=0.01;
[xts,ts,xdts,~,xddts]=pathoverpoints(points,'Interp','spline','Order',5);
ts=ts*tmax/ts(end);
sp=(0:N-1)'; sp=sp/sp(end)*tmax;
c=sp;
sigma2=(diff(c)*1).^2; sigma2=[sigma2;sigma2(end)];
init_cond=zeros(4,size(points,2))  % initial and final vel and acc
coff=0.05; % additinal kernels offset
sfac=3; % sigma fac for aux RBF
t=(0:dt:sp(end))';
RBF=encodeRBF(sp,points,N,c,sigma2*0.6^2,init_cond,coff,sfac);
[xtr,xdtr,xddtr,A]=decodeCartesianRBF(t,RBF);
```



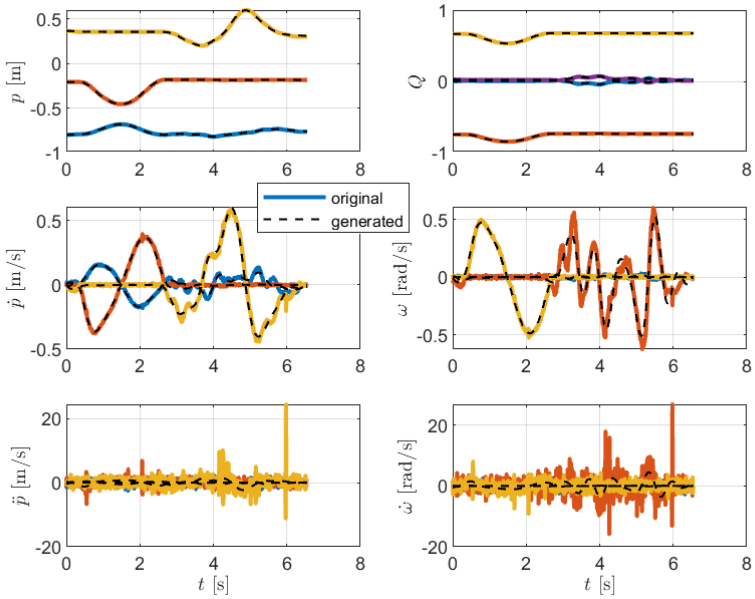
Dynamic motion primitives (DMP)

The RBS Toolbox provides functions for generation and use of DMPs.

Function	Description
`DMP=encodeDMP(t,x,DMP)`	Encoding a DMP
`DMP=encodeCartesianDMP(t,x,DMP)`	Encoding a Cartesian space DMP
`DMP=encodeQuaterionDMP(t,Q,DMP)`	Encoding a quaternion DMP
`[y,dy,ddy]=decodeDMP(t,DMP)`	Calculate path for path time using DMPs
`[y,dy,ddy]decodeCartesianDMP(t,DMP)`	Calculate path for path time using Cartesian DMPs
`[y,dy,ddy]decodeQuaternionDMP(t,DMP)`	Calculate path for path time using Quaternion DMPs
`y=DMP2Path(DMP,x)`	Generate trajectory using dmp
`DMP=Path2DMP(p,N)`	dmp parametrization of sampled trajectory
`S=integrateStepDMP(DMP,S,phaseStoppingSignal)`	Does one step of dmp integration
`S=integrateStepCartesianDMP(DMP,S,phaseStoppingSignal)`	Does one step of dmp integration
`S=integrateStepQuaternionDMP(DMP,S,phaseStoppingSignal)`	integrateStepQDMP: Does one step of quaternion dmp integration

```
load('sample_traj.mat','cart_traj','time');
cart_ori=uniqueCartesianPath(cart_traj);
DMP.N=25;
DMP.a_z=48;
DMP.b_z=12;
[DMP]=encodeCartesianDMP(time,cart_ori,DMP);
T_f = (length(cart_traj)+1)*DMP.dt;
Xmin = exp(-DMP.a_x*T_f/DMP.tau);
%init. states for DMP
S.y = DMP.y0;
S.z = zeros(1,6);
S.x = 1;
y = [];
dy = [];
ddy = [];
i=1;
while S.x > Xmin
    [S]=integrateStepCartesianDMP(DMP,S,0);
    y(i,:)=S.y;
```

```
    dy(i,:)=S.dy;
    ddy(i,:)=S.ddy;
    i=i+1;
end
cart_gen_dmp=y;
vel_gen_dmp=dy;
acc_gen_dmp=ddy;
```



Plotting utilities

The RBS Toolbox provides several plotting utilities, which extend the basic MATLAB plot functions

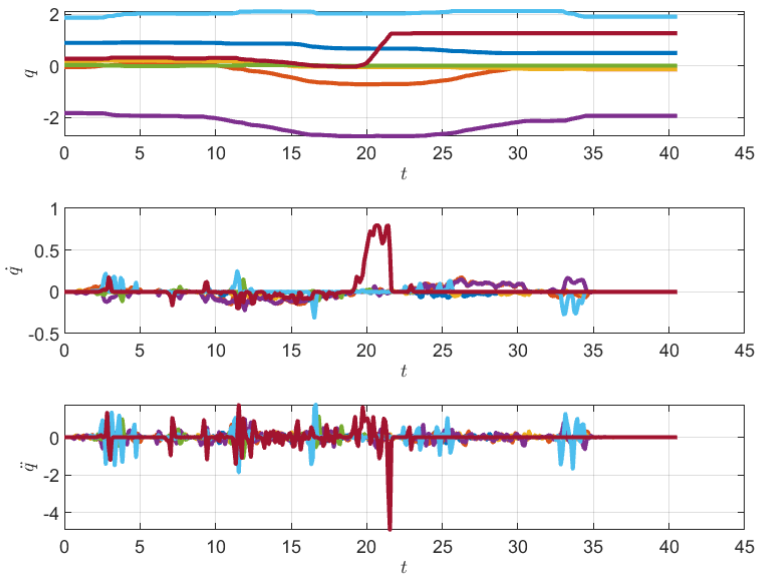
Function	Description
`[h,ax]==plotjtraj(t,qt,qdt,qddt)`	Time plot of joint trajectory
`[h,ax]==plotjpos(t,qt,q_max,q_min)`	Plot positions of joint trajectory with borders
`ud=plotcpath(si,xi)`	Combined plot of Cartesian trajectory
`ud=plotctraj(t,xt,xdt,xddt)`	Time plot of Cartesian trajectory
`h=plotcpos_ori(t,T,...)`	Plot positions or orientations of Cartesian trajectory
`h=plotpathpoints(points)`	Plot point sequence defining a path
`h=plotline(p1,p2,...)`	Plot 2D or 3D line
`h=plotarc(p,R,r,a1,a2,...)`	Plot 3D arc in (x,y) plane
`h=plotplane(p,R,...)`	Plot 3D plane in (x,y)
`h=plotdisc(p,R,...)`	Plot 3D disc in (x,y) plane
`h=plotpath(p,R,...)`	Plot path using line or surface
`h=plotpoint(p,r,...)`	Plot spherical point

Function	Description
`h=plotarrow(p1,p2,...)`	Plot 3D arrow using lines or patch
`h=plotsphere(p,r,...)`	Plot sphere as patch
`h=plotbox(p,R,a,...)`	Plot box as patch
`h=plotcylinder(p1,p2,r,...)`	Plot 3D cylinder as patch
`h=plotcone(p1,p2,r,...)`	Plot 3D cone as patch
`h=plotucs(p,R,...)`	Plot coordinate frame
`h=plotspheregrid(...)`	Plot 3d sphere grid
`nice3Dgraphics`	Script to set camera for nice graphics

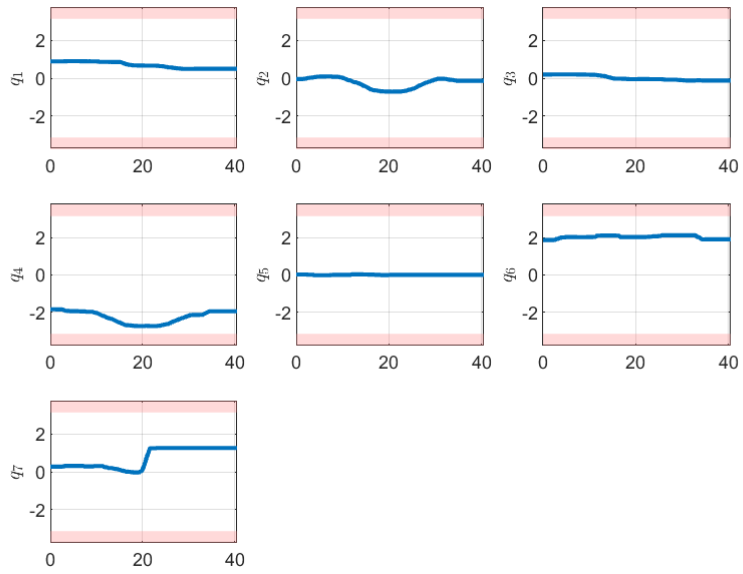
*Many of plotting functions can use optional parameters and also support some of standard MATLAB options for plotting like `'Color'` or `'LineWidth'`. For more information use `help` or `doc` for each function.

Here are some examples how to use plot functions.

```
load('captured_traj.mat','time','joint_traj')
plotjtraj(time,joint_traj)
```

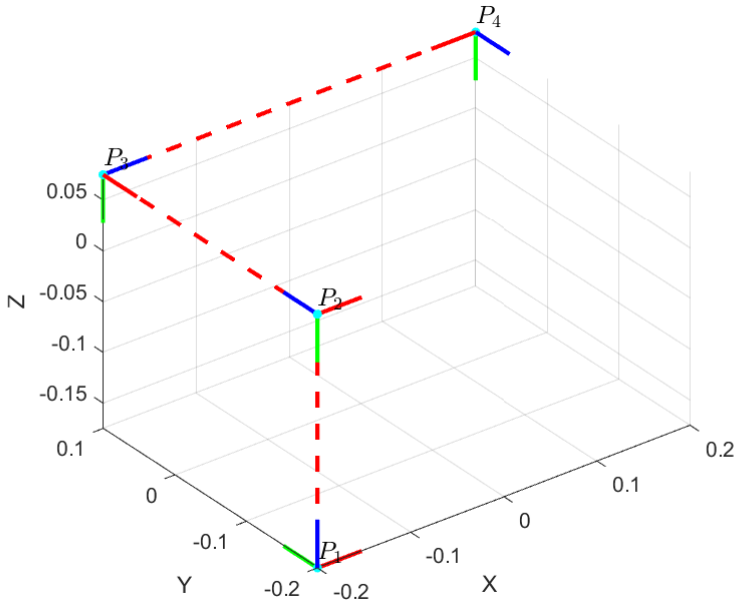



```
plotjpos(time,joint_traj,ones(7,1)*pi,-ones(7,1)*pi)
```



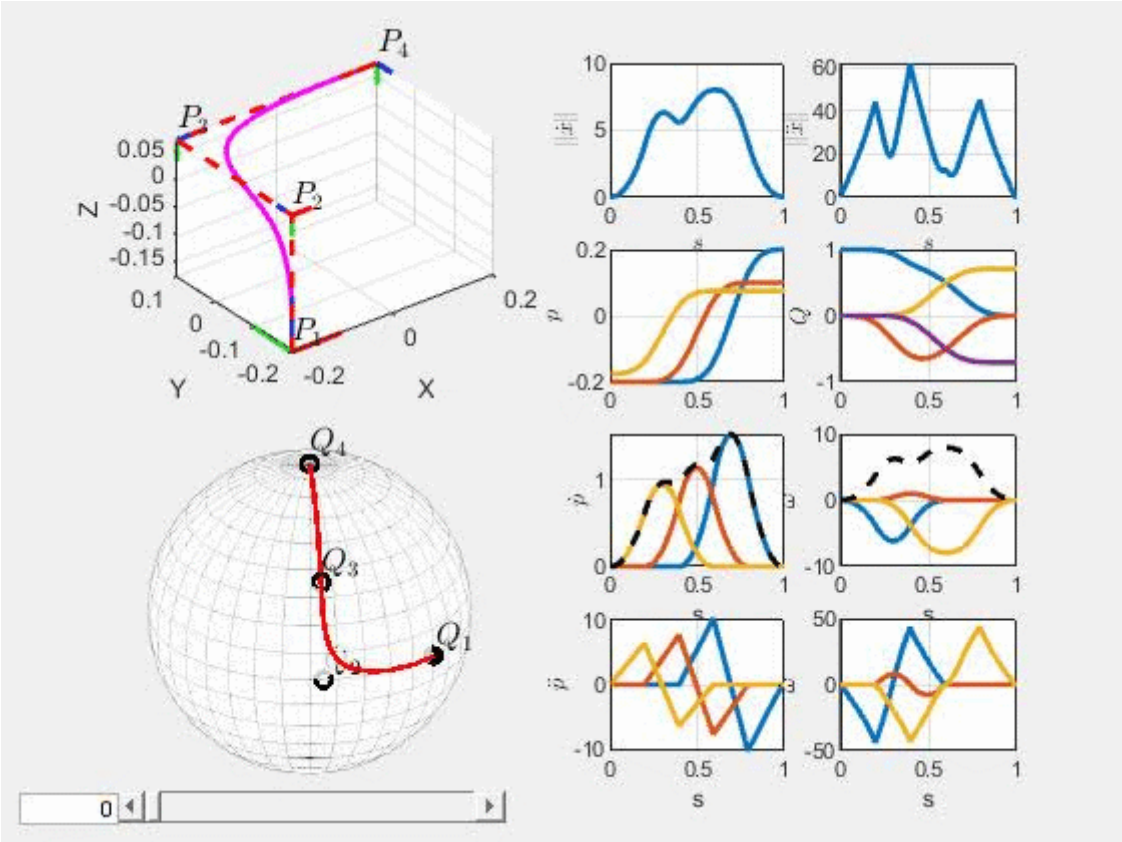
For Cartesian paths and trajectories, there are different plotting utilities. The first one plots the Cartesian poses, represented as frames at positions with pose orientation

```
pte=[-0.2 -0.2 -0.175   0     0     0;...
    -0.2 -0.2  0.075   0     0   -pi/2;...
    -0.2  0.1  0.075 -pi/2   0   -pi/2;...
     0.2  0.1  0.075 -pi     0   -pi/2];
pt=prpy2x(pte);
figure(1)
plotpathpoints(pt)
```



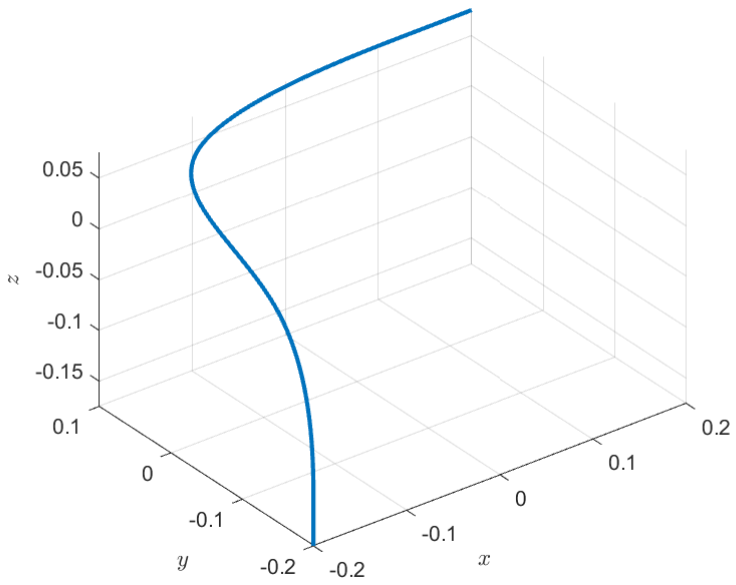
To show the path one can use complex path plot, where the path is plotted in 3D, orientations on a unit sphere and additionally, positions/orientations and derivatives versus path parameter. Using slider in the generated figure, one can check the path by moving along the path.

```
[xi,si,xid,sid,xidd,sidd]=pathoverpoints(pt,'Step',0.01);
ud=plotcpath(si,xi,'Points',pt);
set(ud.sgrid,'FaceAlpha',0.8)
view(ud.axS3,[40 25])
```



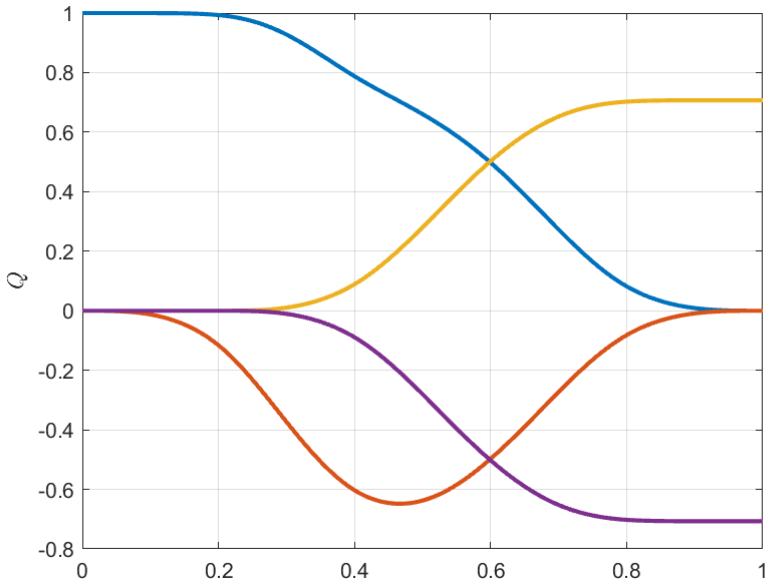
For partial plots use other functions like

```
plotcpos_ori(si,xi,'Type','Pos','Graph','3D','LineWidth',2)
```



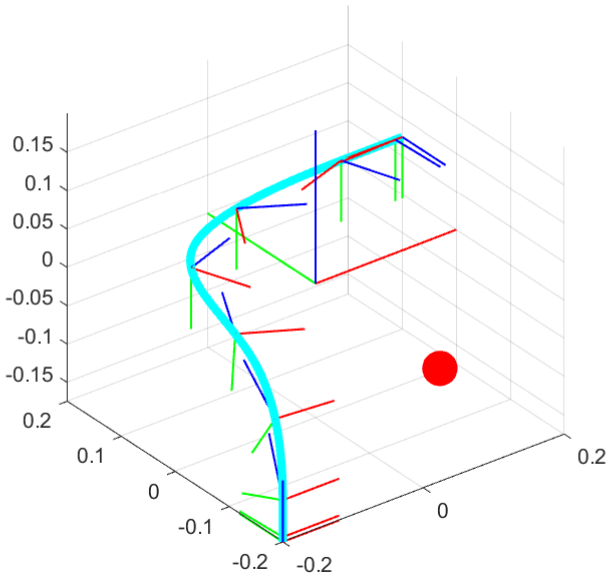
or

```
plotcpos_ori(si,xi,'Type','Ori','Graph','Time','LineWidth',2)
```



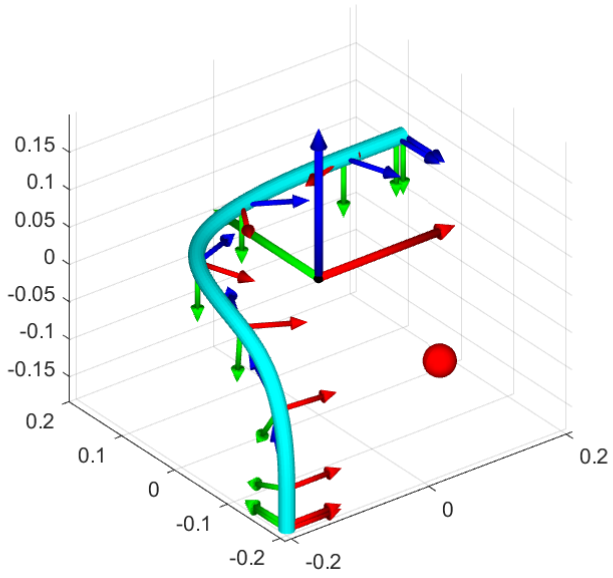
Additional possibilities for paths are

```
plotucs(eye(4),'UCSLength',0.2,'UCSLineWidth',1)
plotpath(xi,'Color','c','LineWidth',4,'UCSIndex',10,'UCSLength',0.08,'UCSLineWidt
h',1)
plotpoint([0.1 -0.1 -0.1]',0.02,'Color','r')
view(3)
grid on
```



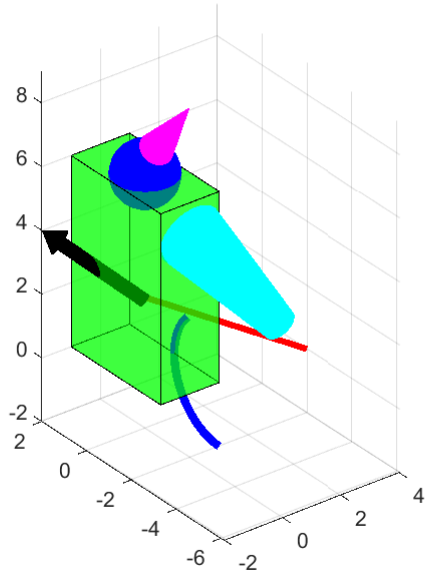
or nicer graphics with the same functions

```
plotucs(eye(4),'UCSLength',0.2,'UCSLineWidth',0.005,'Patch','on')
plotpath(xi,'Color','c','LineWidth',0.01,'UCSIndex',10,'UCSLength',0.08,'UCSLineW
idth',0.005,'Patch','on')
plotpoint([0.1 -0.1 -0.1]',0.02,'Color','r')
view(3)
grid on
axis equal
nice3Dgraphics
```



Finally, there are functions to plot different objects.

```
plotbox([0 0 3]',eye(3),[1 2 3],'Color','g','FaceAlpha',0.5,'EdgeColor','k')
plotsphere([0 0 6]',1,'Color','b')
plotcone([0 -0.5 7]',[0 -2 9]',0.5,'Color','m')
plotcylinder([0 -2 5]',[0 -6 4]',[1 0.5],'Color','c')
plotarrow([0 0 2]',[-2 2 4],'Radius',0.2,'HeadLength',0.6,'HeadRadius',0.4)
plotline([1 2 1]',[4 -2 0]','Color','r','LineWidth',3)
plotarc([1 -2 0]',rot_y(pi/2),2,0,3*pi/4,'Color','b','Linewidth',4)
view(3), axis equal
grid on
```



Generation of kinematic models

To be able to control the motion of a robot we use in RTB kinematic models of robot manipulators. Kinematic models can be generated using the Denavit-Hartenberg (DH) notation, the Modified Denavit-Hartenberg (MDH) notation, or the model is imported from the URDF file describing the robot.

Using DH-parameters we have to define the structure (as a function or a variable) with required fields

```
function [robot]=dh_panda()
% DH Parameters for FRANKA-Emika Panda
robot.name='panda';
robot.description='FRANKA-Emika Panda';
robot.nj=7;
robot.a=[0 0 0.0825 -0.0825 0 0.088 0];
robot.alpha=[-1 1 1 -1 1 1 0]/2*sym(pi);
robot.d=[0.333 0 0.316 0 0.384 0 0.107];
```

Then, using the provided function `gen_kinmodel_dh_matlab` the function `kinmodel_<robot.name>` for the direct kinematics in symbolic form is generated.

```
gen_kinmodel_dh_matlab(dh_panda)
```

If an URDF description file for a robot exists, we can use function `gen_kinmodel_urdf_matlab` to generate the kinematic model. In this case, we can generate a model for only part of the serial kinematic chain, For that, we have to select the initial and the final link in the kinematic chain. For example, if we want to generate a model of a Talos left arm from shoulder to the end of left arm, we execute

```
>> gen_kinmodel_urdf_matlab('talos_arm', 'TALOS', 'Talos.urdf', 'torso_2_link',
'arm_left_7_link');
All joints included in the model (60): 13 14 15 16 17 18 19
Active joints included in the model: 7
Generatig kinmodel_talos_arm for TALOS
```

To show the robot structure given in the URDF file we can use

```
showURDF('Talos.urdf')
```

The generated kinematic model function has a standard from

```
[p,R,J]=kinmodel_<robot.name>(q,tcp)
```

Optionally we can define the tool-center-point (tcp).

A related function is `manipulability`, which calculates the manipulability of a robot

```
>> q=[0 -0.2 0 -1.5 0 1.5 0.7854]';
>> tcp=rp2t(rot_z(-pi/4),[0 0 0.2]');
>> [p,R,J]=kinmodel_panda(q,tcp)
p =
    0.5133
         0
    0.5579
R =
    0.9801   -0.0000    0.1987
   -0.0000   -1.0000         0
    0.1987   -0.0000   -0.9801
J =
         0    0.2249         0    0.1012         0    0.2834         0
    0.5133         0    0.5477         0    0.3125         0         0
         0   -0.5133         0    0.4952         0    0.1472         0
         0         0   -0.1987         0    0.9636         0    0.1987
         0    1.0000         0   -1.0000         0   -1.0000         0
    1.0000         0    0.9801         0    0.2675         0   -0.9801
>>  man=manipulability(J)
man =
    0.0793
```

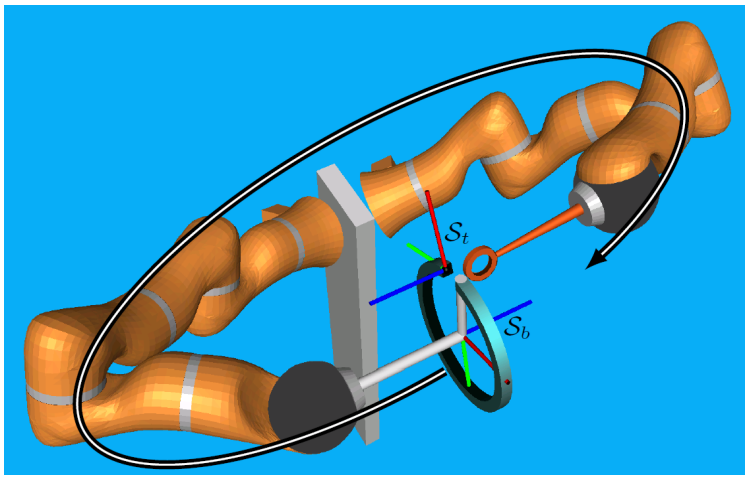
Joining serial kinematic chains

The Toolbox provides functions to join serial kinematic chains into one kinematic chain.

Function	Description
`[x,R,J]=join_robot(x1,R1,J1,x2,R2,J2)`	Joining two robot kinematics
`[x,R,J]=join_reverse_robot(x1,R1,J1,x2,R2,J2)`	Joining two robot kinematics, first in reverse order
`[x,R,J]=join_robot_reverse(x1,R1,J1,x2,R2,J2)`	Joining two robot kinematics, second in reverse order

Function	Description
`[x,R,J]=join_robot_fixed(x1,R1,J1,x2,R2)`	Join kinematic model with fixed kinematic structure
`[x,R,J]=join_fixed_robot(x1,R1,x2,R2,J2)`	Join fixed kinematic structure with robot kinematic model
`[x,R,J]=robot_reverse(x1,R1,J1)`	Robot kinematics from end-effector to base

For example, if we want to joint two KUKA LWR robot arms to a dual-arm robot which starts in the end-effector of the first robot ($\mathcal{S}b$) and ends at the end-effector of the second robot ($\mathcal{S}t$).



then we get the kinematic model of joined dual-arm robot as

```
>> q1=[-0.3460 0.1384 -0.2517 -1.6195 0.5419 1.1927 -0.3092]';
>> tcp1=rp2t([0 0 0.2]');
>> xb1=[0 -0.045/2-0.054 0.92]';  % Robot1 base pos
>> Rb1=rot_y(35/180*pi)*rot_x(60/180*pi)*rot_z(-120/180*pi); % Robot1 base ori
>> q2=[0.0828 1.0148 -0.1233 -1.3104 -0.4346 0.4044 1.5402]';
>> tcp2=rp2t([0 0 0.2]');
>> xb2=[0 0.045/2+0.054 0.92]'; % Robot2 base pos
>> Rb2=rot_y(35/180*pi)*rot_x(-60/180*pi)*rot_z(120/180*pi); % Robot2 base ori
>> xb=Rb1'*(xb2-xb1); % Distance between bases
>> Rb=Rb1'*Rb2; % rotation between bases
>> [x1,R1,J1]=kinmodel_lwr(q1,tcp1);
>> [x1r,R1r,J1r]=robot_reverse(x1,R1,J1);  % From EE to base
>> [x1f,R1f,J1f]=join_robot_fixed(x1r,R1r,J1r,xb,Rb); % Added distance between
bases
>> [x2,R2,J2]=kinmodel_lwr(q2,tcp2);
>> [xx,RR,JJ]=join_robot(x1f,R1f,J1f,x2,R2,J2) % Added second robot
xx =
    0.0001
   -0.1250
   -0.0001
RR =
    0.0008   -1.0000   -0.0001
   -1.0000   -0.0008   -0.0000
    0.0000    0.0001   -1.0000
JJ =
  Columns 1 through 7
```

```
    0.4813    0.4677    0.4630   -0.7745   -0.3300    0.7572   -0.2172
    0.4555   -0.5383    0.4838    0.3734    0.6251   -0.0982   -0.0129
    0.3869    0.6426    0.3538   -0.6885   -0.6245    1.1193   -0.1583
   -0.2868    0.1489   -0.4146    0.0793   -0.8853   -0.3043   -0.0000
    0.4096    0.9121    0.4084   -0.8740   -0.2828    0.9526    0.0000
    0.8660   -0.3821    0.8132    0.4793   -0.3692   -0.0000   -1.0000
  Columns 8 through 14
    0.3972   -0.4688    0.2765    0.5690    0.0033   -0.2779         0
    0.6128    0.3771    0.5860   -0.2889    0.1093    0.0083         0
    0.1583   -0.5170    0.0624    0.1391   -0.0000    0.0000         0
    0.2868    0.3900    0.8946   -0.4139    0.3932    0.0298   -0.0001
   -0.4096    0.8756   -0.4337   -0.8951   -0.0117    0.9996   -0.0000
    0.8660    0.2849    0.1081   -0.1656   -0.9194   -0.0000   -1.0000
```

Robot objects

The essential part of the RTB is to provide robot classes to control different robots in the same manner. To achieve this, we have defined a master robot class where all general properties and methods needed to control a robot are defined

- motion generation methods,
- kinematic controllers,
- transformation functions between different representations of robots states,
- transformation functions between different spaces,
- utility functions, etc.

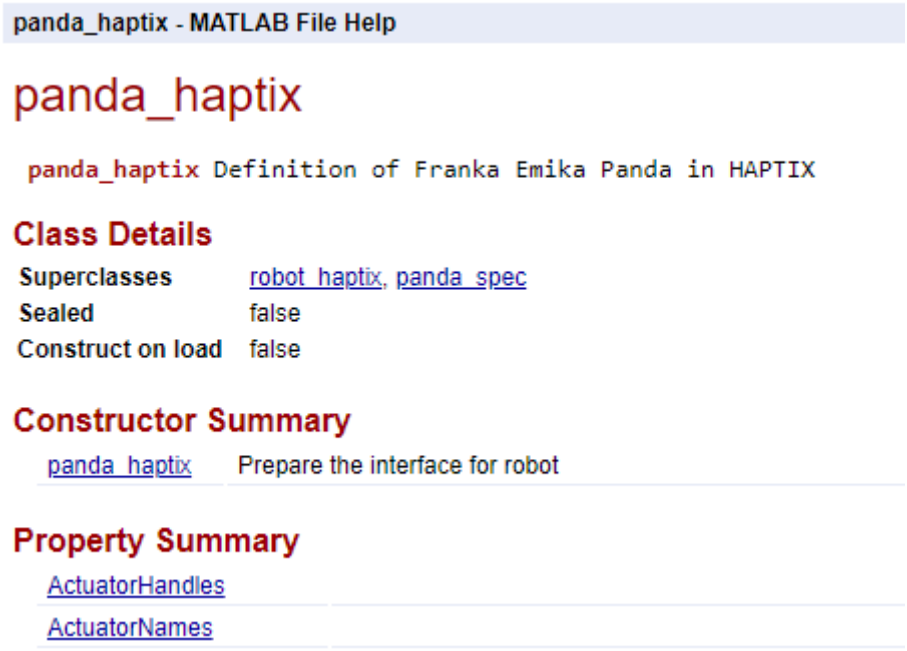
In the implementation classes all specific functionalities of a robots and their interfaces are considered. Additionally, the implementation classes define all abstract methods defined in master class and can redefine some already defined methods to adapt them to special requirements of a particular robot.

To be able to control the motion of a robot we use in RTB kinematic models of robot manipulators. Kinematic models can be generated using the Denavit-Hartenberg (DH) notation or the model is imported from the URDF file describing the robot.

As it is hard for a user to know all methods and properties of a class, all classes in RBS are documented and the information about them can be obtained by using MATLAB `doc` command. For example, using

```
doc panda_haptix
```

we get the complete structure of `panda_haptix` class with all it's properties and methods (here only the first part is shown)



panda_haptix - MATLAB File Help

panda_haptix

panda_haptix Definition of Franka Emika Panda in HAPTIX

Class Details

Superclasses	robot_haptix, panda_spec
Sealed	false
Construct on load	false

Constructor Summary

panda_haptix	Prepare the interface for robot

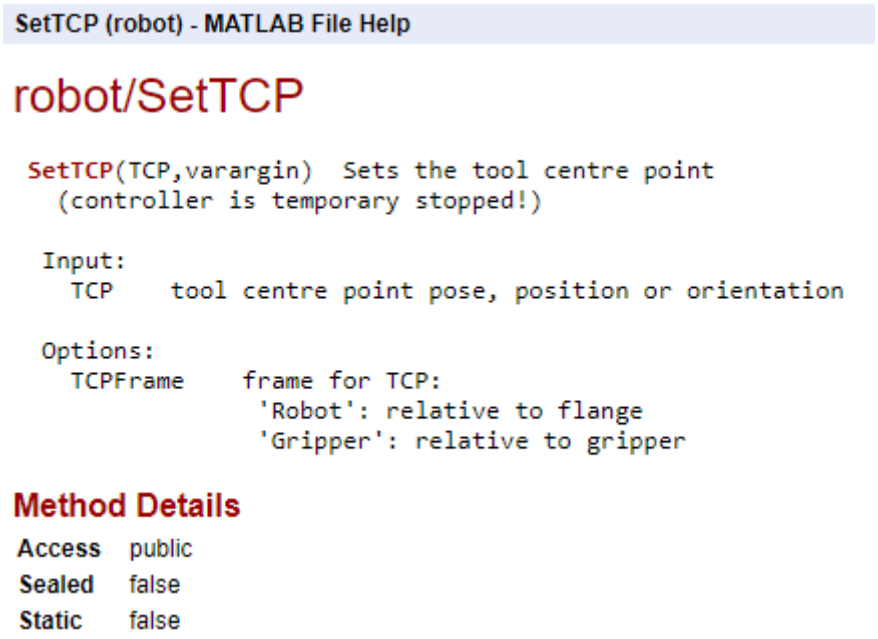
Property Summary

ActuatorHandles
ActuatorNames

To get the information about the object or it's class method (what the method does and the description it's of input/output parameters and options) can use

```
doc r.SetTCP
```

which shows information about `SetTCP` method of object `r`



SetTCP (robot) - MATLAB File Help

robot/SetTCP

SetTCP(TCP,varargin) Sets the tool centre point
(controller is temporary stopped!)

Input:
TCP tool centre point pose, position or orientation

Options:
TCPFrame frame for TCP:
'Robot': relative to flange
'Gripper': relative to gripper

Method Details

Access	public
Sealed	false
Static	false

Creation of a robot object

The creation of a robot object depends on the target robot platform. For example, to create a Panda robot object which will be connected to a real robot using ROS we use the following command

```
r=panda_ros;
```

Note that we can define during creation some optional parameters by adding Name-Value input parameters to the above commands. For example, to change the default TCP transformation, we can use the command

```
r=panda_ros('TCP',rp2t([0 0 0.1]'));  % TCP is 0.1m in z-direction
```

Of course, we can define new TCP later by using a corresponding method

Method	Description
`SetTCP(TCP,...)`	Sets the tool centre point
`GetTCP(...)`	Get robot TCP relative to flange

with optional parameter

Option	Value	Comment
`'TCPFrame'`	`'Gripper'`, `'Robot'`	TCP is relative to gripper TCP is relative to flange

For example,

```
r.SetTCP([0 0 0.1]','TCPFrame','Gripper');  % TCP is relative to gripper
```

To create a robot object for the Haptix simulator, we need first open a corresponding scene (model) with a Panda robot in Haptix , and then we can use the command.

```
r=panda_haptix;
```

Similar commands are for creation of object for other simulation environments (RoboWorks, CoppeliaSim, Gazebo, ...). However, they may require additional input parameters.

We can define the location of the robot base using methods

Method	Description
`SetBase(T)`	Set the robot base Cartesian pose
`T=GetBase(...)`	Get the robot base Cartesian pose

where optional paramaters define the output form.

Robot states

All robot states are read from the robot using method

Method	Description
`GetState`	`Update robot states`

updated by using

To get joint positions and velocities we use methods

Method	Description
`q=GetJointPos(...)`	Get robot configuration
`qdot=GetJointVel(...)`	Get robot joint velocities
`trq=GetJointTrq(...)`	Get robot joint torques

The optional parameter can have two values defining the state output

Parameter	Values	Comment
`'State'`	`'Actual'`,`'Commanded'`	Select type of state

The default value when the parameter is not used is defined by the corresponding defaut property, e.g.

```
>> r.Default_State
ans =
    'Actual'
```

To get the same information we can use robot object's dependent properties.

Property	Comment
`<robot>.q`	Joint positions
`<robot>.qdot`	Joint velocities

Adding suffix `_ref` we get the commanded (reference) values and adding the suffix `_err` we get the errors. For example, to get robot commanded (reference) position the two commands are equivalent

```
r.q_ref
r.GetJointPos('State','Commanded')
```

The joint position error can be obained by

```
r.q_err
```

To get the robot end-effector state there are more possibilities and options. The basic methods are

Method	Description
`T=GetPose(...)`	Get robot task space pose
`R=GetOri(...)`	Get robot task space orientation

Method	Description
`p=GetPos(...)`	Get robot task space position
`v=GetVel(...)`	Get robot task space velocity
`FT=GetFT(...)`	Get robot task space force/torque

These commands have several optional parameters

Parameters	Values	Comment
`'State'`	`'Actual'`, `'Commanded'`	Select type of state
`'TaskSpace'`	`'World'`, `'Robot'`, `'Object'`, `'Tool'`	Coordinate system for output (`'Tool'` is used only for `GetFT`)
`'Kinematics'`	`'Calculated'`, `'Robot'`	`'Calculated'` means that task variable is calculated from joint state using direct kinematics
`'Source'`	`'Robot'`, `'External'`	Used to select source for a signal (`'External'` means that external sensor is used)
`'TaskPoseForm'`	`'TransformationMatrix'`, `'Pose'`,`'Quaternion'`, `'Euler'`, `'RotationMatrix'`,`'Position'`	Output form
`'TaskVelForm'`	`'Twist'`, `'Linear'`, `'Angular'`	`'Twist'` =[`'Linear'` ,`'Angular'`]
`'TaskFTForm'`	`'Wrench'`, `'Force'`, `'Torque'`	`'Wrench'` = [`'Force'`, `'Torque'`]

In RBS we support in general four task coordinate systems:

Space	Description
`'World'`	Cartesian space
`'Robot'`	Cartesian space with origin in the robot base defined by property `<robot>.TBase`
`'Object'`	Custom defined cartesian space defined by `<robot>.TObject`
`'Tool'`	Cartesian space aligned with robot end-effector

The object frame `TObject` can be set using methods

Method	Description
`SetObject(T)`	Set the object frame
`T=GetObject(...)`	Get the object frame

Variables can be transformed from one space to another using methods

Method	Description
`BaseToWorld(...)`	Map variable form base CS to world CS
`WorldToBase(...)`	Map variable form world CS to base CS
`ObjectToWorld(...)`	Map variable form object CS to world CS
`WorldToObject(...)`	Map variable form oworld CS to object CS

For example, to transform task position from Object CS to World CS we use

```
>> r.TObject
ans =
    0.0000   -1.0000         0    1.0000
    1.0000    0.0000         0         0
         0         0    1.0000         0
         0         0         0    1.0000

>> r.ObjectToWorld([0 0.1 0]')
ans =
    0.9000
    0.0000
         0
```

> Note that only fixed frames are supported and that the above transformations consider only rotation beween frames for twists and wrenches.

Beside above methods to get the task position and orientation of the robot end-effector there are several dependent properties for states in the world (Cartesian) coordinate system

Property	Comment
`<robot>.p`	cartesian position
`<robot>.R`	cartesian orientation matrix
`<robot>.T`	cartesian homogenous matrix
`<robot>.Q`	cartesian orientation quaternion
`<robot>.x`	cartesian pose [p Q]
`<robot>.pdot`	cartesian linear velocity
`<robot>.w`	cartesian rotational velocity

Property	Comment
`<robot>.v`	cartesian twist [pdot;w]
`<robot>.F`	cartesian external end-effector forces
`<robot>.Trq`	cartesian external end-effector torques
`<robot>.FT`	cartesian external end-effector forces and torques

For some of them, adding suffix `_ref` we get the commanded (reference) values, adding the suffix `_err` we get the errors, and adding the suffix `_add` we get the values added to the control signal. For example,

```
>> r.x_ref
ans =
    0.4941         0    0.6526    0.0000   -0.9950    0.0000   -0.0998
>> r.x
ans =
    0.4942   -0.0000    0.6520    0.0000    0.9951   -0.0000    0.0992
>> r.x_err
ans =
   -0.0002
    0.0000
    0.0006
    0.0000
   -0.0013
    0.0000
```

Robot states are updated during motion or wait period at sampling frequency defined by object sample time `<robot>.tsamp`. When idle, the state is updated when any of the state viariables is read and the elapsed time since last update is greater than the default update time defined by parameter `<robot>.Default_UpdateTime`. If for some reason faster reading of robot states is needed, then it is necessary to use method `GetState`.

Motion methods

The RBS toolbox allows to control the motion in joint and task-space. For each type of motion RBS provides high-level motion methods, which include trajectory generation and low-level methods, which execute the motion.

When a high-level motion method is executed, first the trajectory for the desired movement is calculated (positions and velocities). Then, the desired values are sent to low-level motion method in a loop for each sample until the end of the trajectory is reached.

The purpose of low-level motion methods is to send the motion command to the specific robot target. These methods send desired positions, velocities and forces/torques to the target controller at specified update rate.

Joint space motion

Methods for motion in joint space are

Method	Description
`JMove(q,t,...)`	Basic joint position movement
`JMoveFor(dq,t,...)`	Relative joint position movement
`JLine(q,t,...)`	Joint position movement with trapezoidal velocity profile
`JPath(path,t,...)`	Joint position movement along path
`JRBFPath(RBF,t,...)`	Joint position movement along RBF encoded path

Basic method to move to robot in joint space to desired position `q` in time `t` is

```
r.JMove(q,t)
```

Note that when trajectory is generated, it is checked whether the generated velocities exceed the maximum values. If this is the case, the execution time is prolonged so that the velocities are lower than the maximal velocities.

In `JMove`, we can use following optional parameters

Option	Value	Comment
`'Wait'`	scalar	Wait time in the final position (sec)
`'Trq'`	vector	Joint torques added in the low-level motion method
`'Traj'`	`'Poly'`, `'Trap'`	Trajectory type: 5th order polynom or trapezoidal velocity
`'State'`	`'Actual'`, `'Commanded'`	State for the initial trajectory position. Default is `'Commanded'`

*Options default values are defined by parameters `<robot>.Defualt_<option>.`

Additionaly, there is a special method to generate trajectory with trapezoidal velocity profile. The following two commands generate the same motion

```
r.JMove(q,t,'Traj','Trap')
r.JLine(q,t)
```

If we need to move the robot for some distance from the current position we can use relative movement command

```
r.JMoveFor(dq,t)  %  relative movement for dq in time t
```

There is also a method which allows to define a special trajectory profile

```
r.JPath(Path,t)
```

where `Path` is a equidistant sequence of positions. The second parameter `t` is defining the time to execute the whole motion. If the path is encoded using radial basis functions (RBF), we can use

```
r.JPathRBF(RBF,t)
```

where `RBF` is parametric description of the path (e.q. obtaind by using `encodeRBF` function). This method has an additional option `'Direction'` which defines how the robot moves along the path.

Option	Value	Comment
`'Direction'`	`'Forward'`, `'Backward'`	Defines if robot moves from initial path position to final path position or in oposite direction.

Task-space motion

For motion in task-space, we use the same concept as for the joint space motion. However, in tasks space we can move the end-effector to a target pose using different spatial pose representations and the pose can be defined in different task spaces.

Method	Description
`CMove(T,t,...)`	Basic Cartesian position movement
`CMoveFor(dT,t,...)` `CDepart(dT,t,...)` `Depart(dT,t,...)` `CMoveRelative(dT,t,...)`	Relative Cartesian position movement
`CApproach(T,dp,t,...)`	Approach Cartesian position movement
`CArc(T,pC,t,...)`	Cartesian position movement along circular path
`CPath(path,t,...)`	Cartesian position movement along path
`CRBFPath(RBF,t,...)`	Cartesian position movement along RBF encoded path

The basic command to move a robot in task-space is

```
r.CMove(T,t)
```

which generates the trajectory in task-space to move the robot from current pose to pose `T` in time `t`. The target pose `T` can be defined as a complete spatial pose or as task-space position or orientation (assuming that unspecified part is not changed).

```
r.CMove(rp2t(rot_z(pi/4),[1 0.5 0.5]'),t)
r.CMove([1 0.5 0.5 0.9239 0 0 0.3827],t)
r.CMove([1 0.5 0.5]',t)  % robot preserves current orientation
r.CMove(rot_z(pi/4),t)   % robot stay in current position
```

Here the first two commans are equivalent and also the final pose after using last two commands is the same.

Method CMove can have optional parameters specifing the trajectory generation or task-space:

Option	Values	Comment
`'TaskSpace'`	`'World'`, `'Robot'`, `'Object'`, `'Tool'`	Coordinate system for the input pose
`'Wait'`	scalar	Wait time in the final position (sec)
`'FT'`	vector	EE force/torques added in the low-level motion method
`'State'`	`'Actual'`, `'Commanded'`	State for the initial trajectory position. Default is `'Commanded'`
`'RotDir'`	`'Short'`, `'Long'`	Rotation angle direction from current to target orientation
`'Traj'`	`'Poly'`, `'Trap'`	Trajectory type: 5th order polynomic or trapezoidal velocity

*Options default values are defined by parameters `<robot>.Defualt_<option>.`

For relative motion in task-space we can use methods `CMoveFor` or aliases

```
r.CMoveFor([0.2 -0.1 0]',1')
r.CDepart([0.2 -0.1 0]',1')
r.Depart([0.2 -0.1 0]',1')
r.CMoveRelative([0.2 -0.1 0]',1')
```

All these commands are equivalent, but is recommended to use the first one. Another relative movement method is `CApproach` (or `Approach`). To approach point, which is at distance `dp` from the target pose `T` we use command

```
r.CApproach(T,dp,t)
```

In practice, motion commands can be expressed in different task spaces. For example, if we want to move the end-effector along the tool z-axis we can use a command

```
r.CMove([0 0 0.1]',1,'TaskSpace','Tool')
```

To use trapezoidal velocity profile, we can use `CLine` instead of command `CMove` with option `'Traj'` (or command `CLineFor` instead of command `CMoveFor`).

Circular motion can be generated using

```
r.CArc(T,pC,t)  % T: target pose; pC center of arc position
```

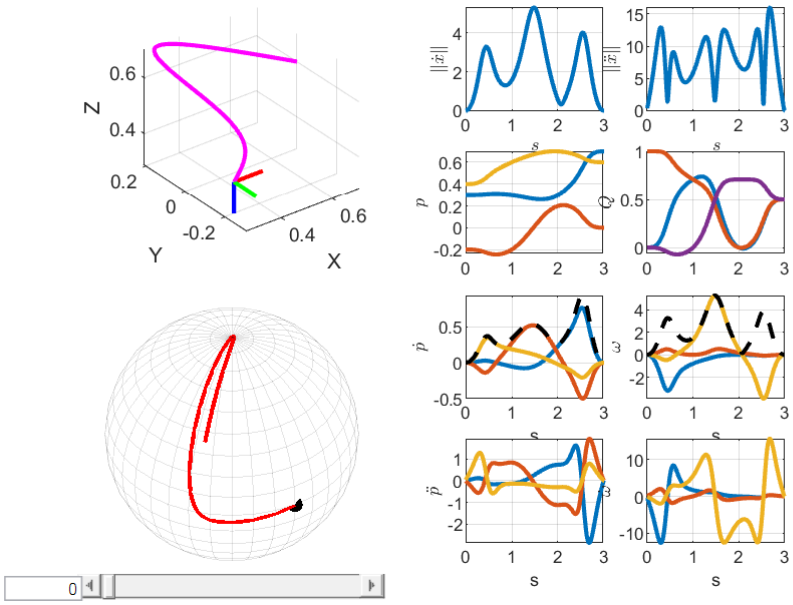
For task-space motion we can define custom path as

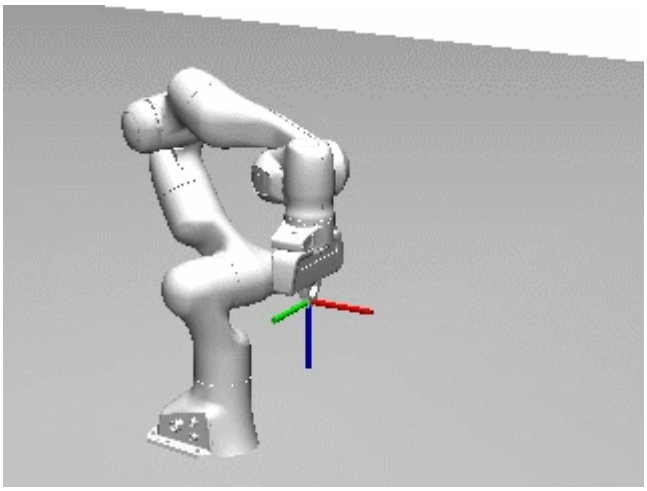
```
r.CPath(path,t)
r.CRBFPath(RBF,t)
```

Here, the input parameter `path` (for `CPath`) can be an array of homogenous matrices or array of spatial poses (pos+quaternion). On the other hand, for method `RBFPath` we can use only representations based on spatial poses. Also method `CRBFPath` has an additional option for motion direction

Option	Value	Comment
`'Direction'`	`'Forward'`, `'Backward'`	Defines if robot moves from initial path position to final path position or in oposite direction.

```
points=[...
    0.3 -0.2   0.4  0    0 pi;...
    0.3 -0.2   0.6   0    0 pi/2;...
    0.3  0.2   0.7   pi   0 pi/2;...
    0.7  0.0   0.6   pi/2 0 pi/2;...
    ]; %    ;...
points=prpy2x(points);
[N,~]=size(points);
tmax=3;
dt=0.01;
sp=(0:N-1)'; sp=sp/sp(end)*tmax;
c=sp;
sigma2=(diff(c)*1).^2; sigma2=[sigma2;sigma2(end)];
init_cond=zeros(4,size(points,2));  % initial and final vel and acc
coff=0.05; % additinal kernels offset
sfac=3; % sigma fac for aux RBF
t=(0:dt:sp(end))';
RBF=encodeRBF(sp,points,N,c,sigma2*0.6^2,init_cond,coff,sfac);
% Path
[xtr,xdtr,xddtr]=decodeCartesianRBF(t,RBF);
plotcpath(t,xtr);
% Robot motion
r=panda_haptix;
r.JMove(r.q_home,1)
r.CRBFPath(RBF,10)
r.CRBFPath(RBF,10,'Direction','Backward')
```





To make motion programming easier we have defined additional methods for motion in commonly used task spaces (Tool And Object):

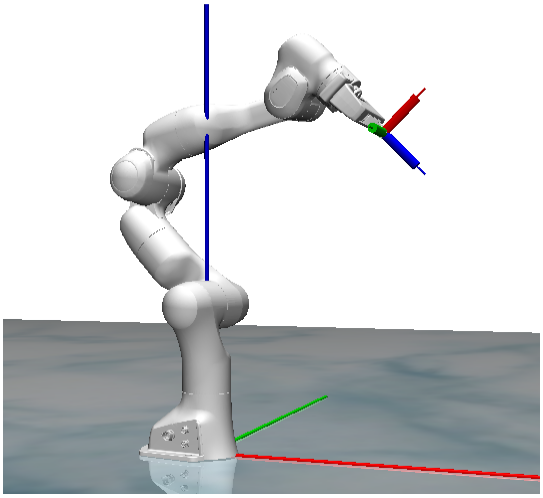
Task-space	Commands
Tool	`TMove`, `TLine`
Object	`OMove`, `OMoveFor`, `OApproach`, `OLine`, `OLineFor`, `OArc`, `OPath`, `ORBFPath`

These methods have the same optional parameters as listed before (if they are appropriate, of course).

> Note that the Tool CS is attached to the robot end-effector. On the other side, the origin and orientation of the Object CS frame can be defined by user. In the next example we define new Object CS and move to the origin.

In the following example, the last two commands specify the same end-effector pose.

```
>> r.SetObject(rp2t(rot_y(3*pi/4)*rot_z(pi),[0.4 0 0.7]'))
>> r.TObject
ans =
    0.7071    0.0000    0.7071    0.4000
    0.0000   -1.0000         0         0
    0.7071    0.0000   -0.7071    0.7000
         0         0         0    1.0000
>> r.OMove(eye(4),1)
>> r.CMove(T,1)
```



Control strategies and low-level movement methods

All high-level motion commands use in the loop low-level methods, which actually send the commanded positions, velocities and torques to the robot controller. Which low level method is used, depends on the control strategy used.

Control strategies

Available control strategies are

Space	Strategies
Joint	`'JointPosition'`, `'JointVelocity'`, `'JointImpedance'`
Task	`'CartesianPose'`, `'CartesianVelocity'`, `'CartesianImpedance'`

However, which can be applied depends on the target robot system. For all simulation targets only `'JointPosition'` strategy can be used (actually, the strategy can not be changed). On the other side, if the target is a real robot, then strategies can be selected if a corresponding controller is implemented on the target controller. We can get, set and verify current strategy using commands:

Method	Description
`[strategy]=GetStrategy`	Get control strategy
`SetStrategy(strategy)`	Set current control strategy
`val=isStrategy(strategy')`	Check if strategy is active

Method	Description
`AvailableStrategies`	List of all avalable control strategies

To check available strategies, use

```
>> panda_ros.AvailableStrategies
ans =
  4×18 char array
    'JointPosition     '
    'JointImpedance    '
    'CartesianVelocity '
    'CartesianImpedance'
```

and change it use

```
>> r.verbose=2;  % select more messages
>> r.GetStrategy
ans =
    'JointPosition'

>> r.SetStrategy('JointImpedance')
Panda:Haptix: Strategy can not be set
>> r.isStrategy('JointImpedance')
ans =
  logical
   0
```

Robot compliance

If one of the impedance control strategies is selected, then we can get or set the stiffness (compliance) of the robot using methods

Method	Description
`val=GetJointStiffness`	Get current joint stiffness
`SetJointStiffness(stiffness)`	Set joint stiffness
`val=GetJointDamping`	Get current joint damping
`SetJointDamping(damping)`	Set joint damping
`SetJointSoft(softness)`	Set stiffness level: 0:compliant; 1:stiff
`SetJointCompliant`	Sets zero joint stiffness
`SetJointStiff`	Sets maximal joint stiffness
`val=GetCartesianStiffness`	Get current cartesian stiffness
`SetCartesianStiffness(stiffness)`	Set cartesian stiffness
`val=GetCartesianDamping`	Get current cartesian damping

Method	Description
`SetCartesianDamping(damping)`	Set cartesian damping
`SetCartesianSoft(softness)`	Set stiffness level: 0:compliant; 1:stiff
`SetCartesianCompliant`	Sets zero Cartesian stiffness
`SetCartesianStiff`	Sets maximal Cartesian stiffness

Note that the stiffness and damping values depend on the particular implementation of impedance controllers on robots and that in some of the robot class implementations additional methods related to compliance are available.

Some of the above method have optional parameters. One of the important options is `'HoldPose'` used for example in `SetJointStiffness` method

Option	Values	Comment
`'HoldPose'`	`'on'`, `'off'`	Value `'on'` means that the robot holds its position when it becomes more stiff. Value `'off'` means that stiffness will increase slowly to prevent sudden movements of the robot.

> The stiffnes and damping parameters are used when the selected target controller is impedance controller, e.g. `'JointImpedance'` or `'CartesianImpedance'`.

Low-level joint movement method

Regardless of the selected control strategy, joint motion is executed using the following low-level method

Method	Description
`GoTo_q(q,qdot,trq,wait)`	Direct joint motion command
`GoToActual`	Move to actual configuration in 1sec

which sends to the target controller the desired positions, desired velocities and additional joint torques. This method is called by the high-level motion methods periodically at frequency defined by the object property `<robot>.tsamp`. Sampling time can be changed by using method

Method	Description
`SetTsamp(tsamp)`	Set sampling time

Due to MATLAB limitation `tsamp` cannot be less than 0.01ms. When GoTo_q is used directly, we have to set the input parameter `wait` to 0 or `tsamp`, except when want to allow the target controller to reach the desired state.

Low-level task-space movement methods

When using task-space movement, the situation is slightly different. Namely, the low-level motion method depends on the control strategy used on the target platform. Therefore, all high-level task-space movement methods call an intermediate method

Method	Description
`GoTo_T(T,v,FT,...)`	Move to task pose T

which selects the corresponding low-level method based on current control strategy and implemented target controller. This method allows also to pass options from the high-level methods to the low-level methods.

The master robot class provides a sophisticated kinematic controller `GoTo_TC` which allows task-space control of redundant robots.

Method	Description
`GoTo_TC(T,v,FT,...)`	Move to task pose T and v using task space position kinematic controller

Method `GoTo_T` uses `GoTo_TC` whenever the target does not support task-space control. The kinematic controller implemented in `GoTo_TC` method uses a lot of parameters, which are passed as `options`:

Option	Values	Comment
`'TaskSpace'`	`'World'`, `'Robot'`, `'Object'`	Task-space of the input pose `T` and options `T_opt` and `v_ns`
`'Wait'`	`tsamp`	Maximal wait time in final position
`'PosErr'`	`inf`	Maximal task position error in final position
`'OriErr'`	`inf`	Maximal task orientation error in final position
'TaskDOF'	`[1 1 1 1 1 1]`	Selection of DOF in task-space (1: included in task-space)
`'TaskContSpace'`	`'World'`, `'Robot'`, `'Tool'`, `'Object'`	Task-space for the controller (used to align task DOF)

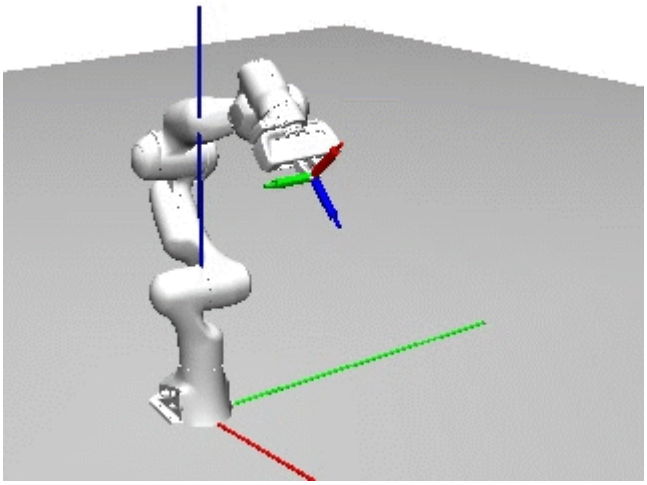
Option	Values	Comment
`'NullSpaceTask'`	`'None'`, `'Manipulability'`, `'JointLimits'`, `'ConfOptimization'`, `'PoseOptimization'`, `'TaskVelocity'`, `'JointVelocity'`	Secondary task used in redundancy resolution control
`'q_opt'`	`q`	Optimal joint configuration (used for `'ConfOptimization'`)
`'T_opt'`	`T_opt`	Optimal pose (osed for `'PoseOptimization'`)
`'v_ns'`	`v_ns`	Task velocity for NS (used for `'TaskVelocity'`)
`'Kp'`	`10`	Task-space position gain
`'Kff'`	`1`	Task-space velocity feedforward gain
`'Kns'`	`10`	Null-space gain

> Note that these options have to be defined in high-level motion commands and then they are passed to the `GoTo_TC`.

An example of using functional redundancy is

```
>> r.T % Initial pose
ans =
    0.7065   -0.0049    0.7077    0.4000
   -0.0050   -1.0000   -0.0019   -0.0012
    0.7077   -0.0022   -0.7066    0.7000
         0         0         0    1.0000

>> r.CMoveFor([0 0.3 0]',1)
>> r.T % Intermidiate pose
ans =
    0.7065   -0.0048    0.7077    0.3999
   -0.0051   -1.0000   -0.0018    0.3004
    0.7077   -0.0023   -0.7065    0.7000
         0         0         0    1.0000
>> r.CMoveFor([0 -0.3 0]',2,'TaskDOF',[1 1 1 0 0 0])
>> r.T % Final pose
ans =
    0.6730    0.6682    0.3171    0.3999
    0.7048   -0.7094   -0.0007   -0.0003
    0.2245    0.2240   -0.9484    0.7000
         0         0         0    1.0000
```



where the robot task-space in the second movement includes only position of the end-effector. Consequently, the final orientation is not the same as the initial.

When task-space controller is implemented on the target platform (e.g. real Panda on ROS, KUKA LWR), then `GoTo_T` used method `GoTo_X`

```
r.GoTo_X(x,v,FT,wait,options)
```

Checks during motion

RBS Toolbox provides utilities to perform actions during robot motion, which can influence the execution of the commanded motion.

First we have to prepare a callback function, which will be called in each sample of trajectory execution. For example, if we wan to push into an object the robot until a specified force is achieved the call back function can be like this

```
function stat=Check_Fz_tool(robot)
%Check force in object space z-axis
if isobject(robot) &&  isrobot(robot)
    FT=robot.WorldToObject(robot.FT);
    p=robot.GetPos('TaskSpace','Object');
    robot.user.Test.i=robot.user.PritisniGumb.i+1;
    robot.user.Test.F(robot.user.PritisniGumb.i)=FT(3);
    robot.user.Test.p(robot.user.PritisniGumb.i)=p(3);
    stat=(FT(3)>robot.user.PritisniGumb.Fmax);
end
```

Here, the output status changes when the external force exceeds the maximal value. In this example we also capture the position and force in desired direction.

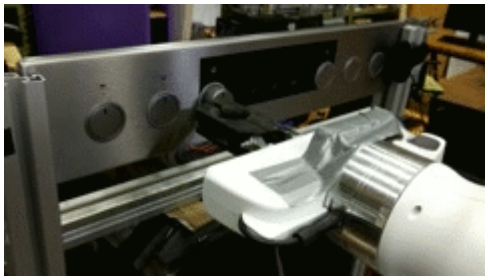
> Note that the only input parameter has to be the `robot` object and all additional parameters have to be defined in the object's user property `robot.user`. Next, the output variable `stat` interrupts the motion execution if it's value is not 0.

> The callback function has to have short execution time so that the overall execution time in one step is less the the selected sampling time.

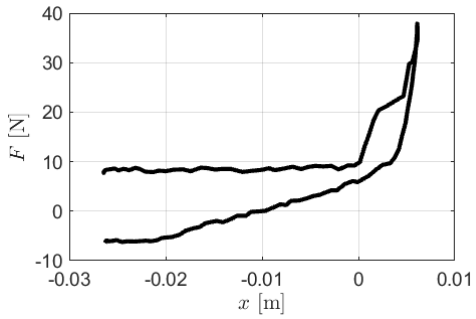
This callback function is then applied to the high-level motion method in the following way

```
r.OLine(T,1);
r.SetMotionCheckCallback(@Check_Fz_tool);
r.user.Test.i=0;
r.user.Test.Fmax=30;
r.EnableMotionCheck;
if r.OLine([0 0 0.1]',5)==0
    r.ShowMsg('Maximal force reached')
end
r.OLine(T,1);
r.DisableMotionCheck;
plot(r.user.Test.p,r.user.Test.F)
```

The robot is commanded to move toward the object and when the maximal force is reached, motion is aborted and the robot moves back.



The captured forces during motion are



Capturing robot motion

Often it necessary to capture some robot states for later analysis. RBS Toolbox provides methods to capture different variables during robot motion.

Method	Description
`Update`	Update robot states (using `GetState`) and calls callback function
`flag=GetUpdateStatus`	Returns Update calling flag
`EnableUpdate`	Enables calling Update
`DisableUpdate`	Disables calling Update
`SetCaptureCallback(fun)`	Define function to capture desired variables
`StartCapture`	Start calling capturing function
`StopCapture`	Stop calling capturing function

Method	Description
`SetUserData(data)`	Set value of custom variable, which can be captured
`data=GetUserData`	Get value of custom variable

Low-level motion methods call the `Update` method, which updates states by calling basic method `GetState` and when capturing call back is set and enabled, it calls a user defined callback function. The user has to prepare a callback function and assign it to the `Update` method with `SetCaptureCallback` method .

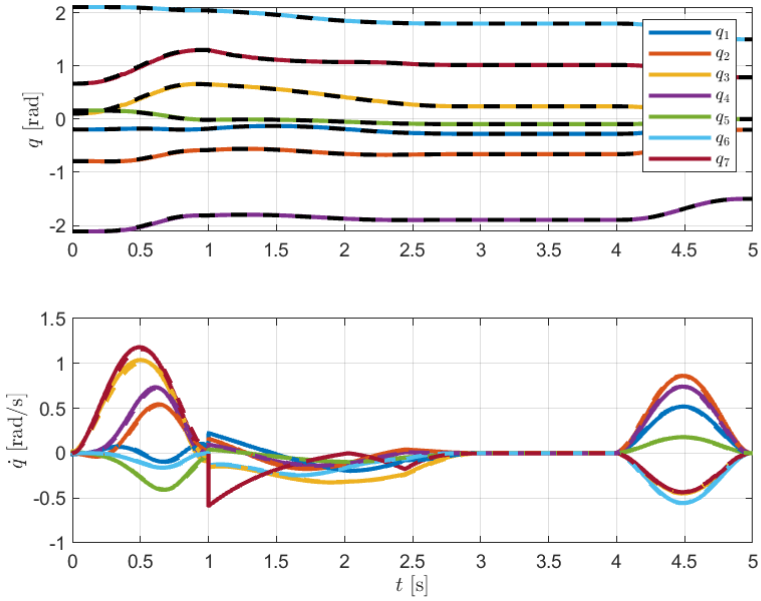
In the Toolbox there is predefined capture function do_capture, which can serve as a template. Before starting capture, it may be necessary to initialize capturing. For that, function init_do_capture is predefined (to be used as a template). Then capturing can be enabled and disabled by using methods `StartCapture` and `StopCapture`.

```
% Initialization
n=10000;
init_do_capture
r.SetCaptureCallback(@do_capture);
r.StartCapture;
% Motion
r.SetUserData(0)
r.CMoveFor([0 0.3 0]',1)
r.SetUserData(1)
r.CMoveFor([0 -0.3 0]',2,'TaskDOF',[1 1 1 0 0 0])
r.SetUserData(2)
r.Wait(1)
r.SetUserData(3)
r.StopCapture
```

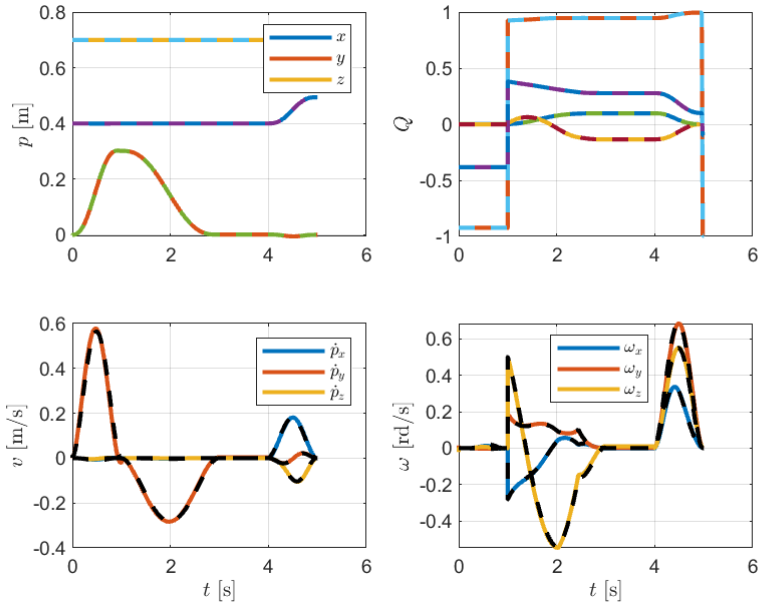
Using the predefined functions, the variables are saved into the structure `states`. Method `SetUserData` allows to mark desired situations during the capturing process for easier analysis of captured data.

To plot the captured states we can use utility functions plot_joint, plot_task, plot_force and plot_user. To delete duplicate states and NaN we can use function `capture_data_delete_same_time`.

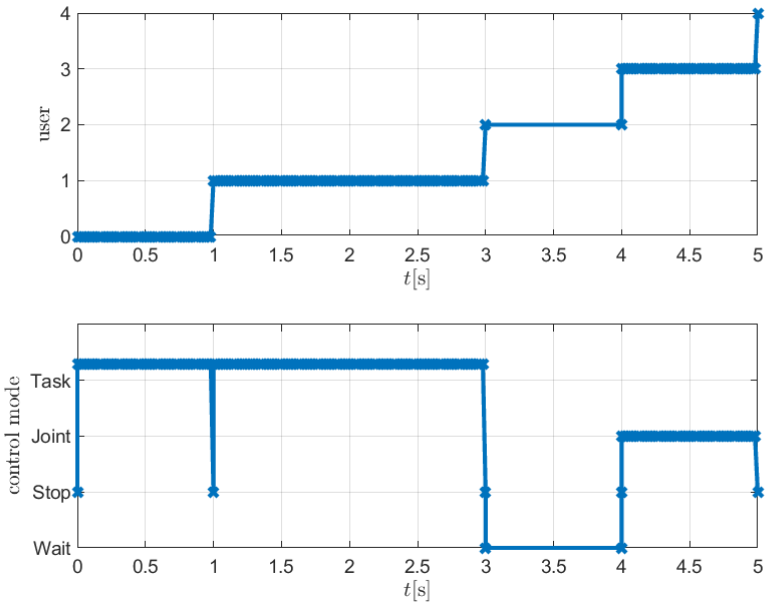
```
states=capture_data_delete_same_time(states);
plot_joint(states,1)
```




```
plot_task(states,2)
```




```
plot_user(states,3)
```

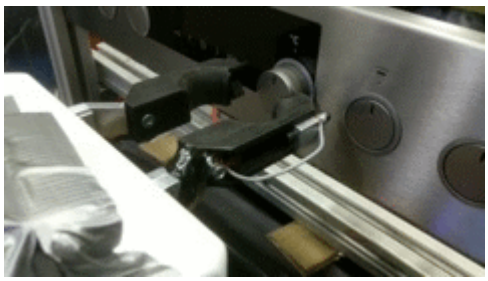


Grippers

The RBS Toolbox provides beside robot classes also classes for grippers . Using gripper object, we can control actions of a gripper. Currently, only Franka Emika Panda gripper is supported.

In the following example we define gripper, attach it to the robot and perform grasping and rotation of a button

```
r=panda_ros;
g=panda_gripper_ros;
g.Homing;
r.SetGripper(g)
r.OApproach(T_Button,[0 0 -0.03]',2);
r.OMove(T_Button,0.5);
q_0=r.q;
q_0(7)=-Hod/2;
r.JMove(q_0,1)
r.gripper.Grasp('Width',0.005,'Force',10,'Eps',0.02)
if r.gripper.isClosed
    r.JLine(q_0+[0 0 0 0 0 0 3*pi/4]',3)
    r.JLine(q_0,3)
else
    ShowMsg('Not grasped')
end
r.Wait(0.2)
r.gripper.Open;
r.OMove(T_Button,0.5);
r.OApproach(T_Button,[0 0 -0.03]',0.5);
```



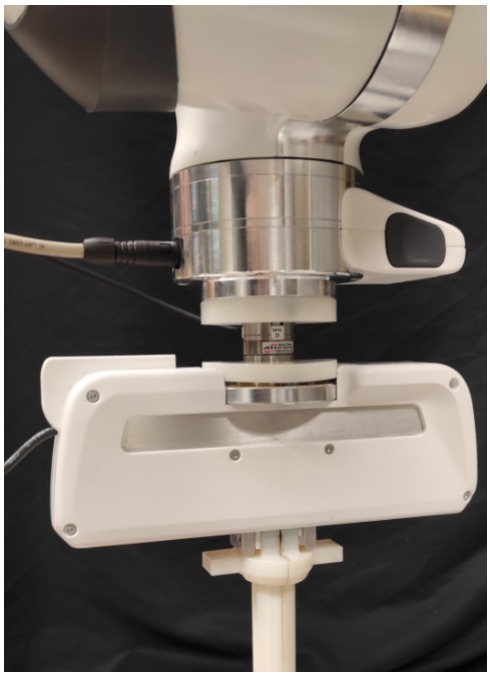
Force sensors

The RBS Toolbox provides classes for sensors. Currently only ATI F7T sensors are supported. They can be used independently or they can be attached to a robot. The for FT sensors the following properties and methods are available.

Property	Comment
`<sensor>.F`	force vector
`<sensor>.Trq`	torque vector
`<sensor>.FT`	wrench vector

Method	Description
`FT=GetFT`	Get forces and torques and substract offsets (zeroing)
`FT=GetRawFT`	Get forces and torques
`ZeroingFT(time)`	Zeroing FT sensor

A F/T sensor can be attached to the robot as shown on the Figure.



In we want to use the attached sensor to measure forces and torques at the end-effector, we have to attach the sensor to the robot object. To transform measured forces and torques to the end-effector we have to define also the location of the sensors relative to the robot flange. Additionally, to compensate the mass of the elements mounted on the force sensor, we have to define the mass and the COM of the load. The RBS Toolbox provides the following methods.

Method	Description
`SetFTsensor(sensor)`	Assign force/torque sensor object to robot object

Method	Description
`[name,obj]=GetFTsensor`	Get force/torque sensor assigned to robot object
`SetFTsensorFrame(T)`	Define force/torque sensor frame in robot flange CS
`T=GetFTsensorFrame(...)`	Get force/torque sensor frame in robot flange CS
`T=GetFTSensorPose(...)`	Get pose of the force/torque sensor
`SetFTsensorLoad(...)`	Set the load parameters (Mass, COM, Offset)
`[mass,COM]=GetFTsensorLoad`	Get the load parameters
`UpdateFTsensorOffset(...)`	Updates Force/torque sensor offsets

An example of measuring forces using external F7T sensor is

```
>> r=panda_ros;
>> sen=ati_ft;
>> sen.FT
ans =
   -0.0000
    0.0000
   32.8736
    0.0017
    1.3149
   -0.0000

>> r.SetFTsensor(sen)
>> r.SetFTsensorFrame([0 0 0.03]');
>> r.SetFTSensorLoad('Mass',0.7,'COM',[0 0 0.38]','Offset',[0.1 -0.2 0.1 0.1 -0.1
-0.1])
>> r.GetFT('Source','External');
ans =
   -0.0000
    0.0000
   32.8736
    0.0017
    1.3149
   -0.0000
```

For estimation of load mass and COM it is necessary to measure sensor forces/torques for different orientations of the sensor. One possibility is to use random joint configurations (as used in function `Motion4LoadEst.m`)

```
n=50;
Ft=nan(n,6);
Rt=nan(3,3,n);
r.JMove(r.q_home,5)
r.FTsensor.SetOffset(zeros(6,1))
for i=1:n
    qx=r.q_home+[0 0 0 0 1 1 1]'.*(rand(r.nj,1)-0.5);
    r.JMove(qx,1)
    r.Wait(0.2)
    Ft(i,:)=r.FTsensor.FT';
    Rt(:,:,i)=t2r(r.GetFTsensorPose);
end
```

and then the mass, COM and sensor offsets can be calculated

```
[mass,COM,Off]=load_est(Ft,Rt)
r.SetFTSensorLoad('Mass',mass,'COM',COM','Offset',Off)
```

As sensor offsets may change during time, its offsets can be updated. Note that is offset can be updated using command

```
r.UpdateFTsensorOffset
```

when no external forces/torques are acting on the sensor.

As the property `<robot>.FT` returns the forces and torques using default options, it is possible to globally change which F/T data is returned by `<robot>.FT`. For example, the same date as above we get using the following commands.

```
>>r.SetDefault('Source','External')
>>r.FT
ans =
   -0.0000
    0.0000
   32.8736
    0.0017
    1.3149
   -0.0000
```

Utility methods

In RBS Toolbox, there are several utility methods.

Method	Description
`T=DKin(q,...)`	Direct kinematics
`q=IKin(T,...)`	Numeric iterative inverse kinematics solution
`J=jacobi(q)`	Calculates the Jacobian matrix from the kinematic model

Method	Description
`[Idx,Grad]=Manipulability(q,...)`	Manipulability measure
`d=JointDistance(q,...)`	Distance between current position and q
`d=TaskDistance(T,...)`	Distance between current position and T
`[stat,qlim]=CheckJointLimits(q)`	Check if q in joint range
`[dq,dqLow,dqUp]=DistToJointLimits(q)`	Calcualte distance to joint limits
`q=WrapToJointLimit(q)`	Try to wrap q to joint range
`ResetCurrentTarget`	Resets current commanded values to actual
`Wait(t,dt)`	Wait and update states while waiting
`WaitUntilStopped(eps)`	Wait as long as robot is not stopped
`ResetTime`	Resets the time
`SetFTsensor`	(sensor) Assignes force/torque sensor to robot EE
`SetGripper`	(grip) Assignes gripper to robot EE
`SetDefault`	Set the value of a default parameter
`SetDefaultGains(Kp,Kff,Kns)`	Set default kinematic controller gains
`Message(msg,verb)`	Show messages
`Warning(msg)`	Show warning

If for some reason, we want to set the reference values to current actual values, we can use

```
r.ResetCurrentTarget
```

We can get the distance between the current position and a target.

```
>> r.CMove(T,1)
>> r.CMoveFor([0 0.3 0]',1)
>> dist=r.JointDistance(r.q_home)
dist =
    0.1863
    0.3870
   -0.6566
    0.3138
    0.0133
   -0.5465
   -0.5166

>> xdist=r.TaskDistance(T)
xdist =
    0.0005   -0.3016   -0.0002   -0.0001    0.0032    0.0000
```

We can wait or wait until robot is stopped.

```
r.Wait(1)
r.WaitUntilStopped(eps) % until norm(qdot)<eps
```

We can print messages or warninigs.

```
>> r.Message('Task finished')
Panda:Haptix: Task finished
>> r.Warning('Gripper not opened')
Warning: Panda:Haptix: Gripper not opened

> In robot/Warning (line 3487)
>>
```

For information about other methods use MATLAB `doc` functionality.

Creation of robot classes and robot objects

The essential part of the RTB is to provide robot classes to control different robots in the same manner. To achieve this, we have defined a master robot class `robot.m` where all general properties and methods needed to control a robot are defined. This class is parent of all implementation classes for particular robots. In `robot.m` all methods common to all robots are defined like

- motion generation methods,
- kinematic controllers,
- transformation functions between different representations of robots states,
- transformation functions between different spaces,
- utility functions, etc.

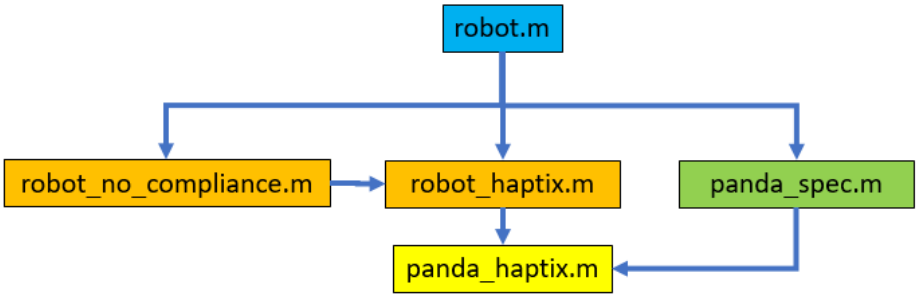
Some of the methods are abstract method, which have to be defined in the child class. Superclass `robot.m` defines also all common robot properties.

In the implementation classes all specific functionalities of a robots and their interfaces are considered. Additionally, the implementation classes define all abstract methods defined in master class `robot.m` and can redefine some already defined methods to adapt them to special requirements of a particular robot.

Some of the implementation methods depend also on the way how MATLAB is communicating with the target platform, i.e. a real robot platform or the environment used to simulate the robot. Therefore, special subclasses are defined including all specific functionality of the target environment.

To avoid duplicating the code we are using subclasses, which group related properties and methods. Note that subclasses extend inherited methods to provide specialized functionality, while reusing the common aspects. They can add also new methods and properties. Using such hierarchical organization of classes allows easier and more transparent robot object creation.

The following figure explains the structure of a robot class used to define a robot object (e.g. Panda robot object for Haptix target), which inherits basic `robot.m` class and several subclasses in several levels.



The topmost class `robot.m` is the superclass of all robot classes.

Next are the subclasses defining the parameters and methods for specific targets (explained later for each target platform). In the above example, properties and methods related to the Haptix simulator are defined in two subclasses: `robot_haptix.m` (main subclass for Haptix) and `robot_no_compliance.m` (subclass to define dummy compliance related methods).

The specific robot parameters and methods are in subclasses `<robot>_spec.m` specific.

Finally, the robot class can be defined as a subclass of all above mentioned classes. Typically it has the name `<robot>_<target>`. For example, the Panda class for Haptix environment is

```
classdef panda_haptix < robot_haptix & panda_spec
    %PANDA_HAPTIX Definition of Franka Emika Panda in HAPTIX
    methods
        function robot=panda_haptix(varargin)
            %Prepare the interface for robot
            opt=set_options({},varargin);
            robot=robot@robot_haptix('Panda',opt);
            robot.Specifications;
            robot.control_strategy='JointPosition';
            robot.Init;
        end
    end % methods
end % classdef
```

The class `panda_haptix` is then used to define a robot object simulated in Haptix simulator.

```
r=panda_haptix;
```

To define the robot class for another target, we have to use corresponding subclasses. For example, for Panda robot with the RoboWorks as target only the `robot_haptix.m` subclass has to be replaced by `robot_roboworks.m` subclass.

To define the class for another robot, we have to use corresponding `<robot>_spec.m`. Currently, the RBS Toolbox provides classes for the following commercial robots :

- Franka Emika Panda,
- Kuka LWR,

- KUka iiwa
- Universal robots UR10,
- Universal robots UR5,
- Universal robots UR10e,
- Universal robots UR5e,
- Mitsubishi PA10,
- Kinova Jaco2, and
- Pal Talos humanoid robot.

The supported target platforms are:

- real robot:
    - Franka Emika Panda over ROS
    - Kuka LWR over FRI and IJS xPC Target server
    - UR10 over ROS
- MuJoCo Haptix simulator: all robots
- RoboWorks simulator: Franka Emika Panda, Kuka LWR, PA10, UR10, UR5 and UR5e
- CoppeliaSim simulator: Franka Emika Panda, Kuka LWR, Kinova Jaco2 and UR10.

Beside robots there are also classes for grippers and sensors. Currently, only the gripper for Franka Emika Panda is supported for target platforms:

- real robot over ROS,
- MuJoCo Haptix simulator, and
- RoboWorks simulator:

Robot parameters subclass `<robot>_spec.m`

In subclass `<robot>_spec.m` parameters and direct kinematics methods for a particular robot are defined. These classes have to define at least the parameters:

Parameter	Description
`nj`	number of joints
`q_home`	home joint configuration,
`q_max` and `q_min`	joint position limits,
`qdot_max`	maximal joint velocities, and
`v_max`	maximal Cartesian velocity of end-effector

and method

Method	Description
`kinmodel`	Direct kinematic model

Beside them, the class can define also other robot specific parameters or methods. For example, for Panda robot additional parameters defining the default gripper transformation are included in the `panda_spec.m` class

```
classdef panda_spec < robot
    %Franka Emika Panda specifications
    methods (Access=protected)
        function Specifications(robot)
            robot.nj=7;
            robot.TCPGripper=[0.7071 0.7071 0 0;-0.7071 0.7071 0 0;0 0 1 0.1034;0
0 0 1];
            robot.q_home= [0 -0.2 0 -1.5 0 1.5 0.7854]'; % home joint
configuration
            robot.q_max=[2.8973 1.7628 2.8973 -0.0698 2.8973 3.7525 2.8973]';   %
upper joint limits
            robot.q_min=[-2.8973 -1.7628 -2.8973 -3.0718 -2.8973 -0.0175
-2.8973]';  % lower joint limits
            robot.qdot_max=[2.1750 2.1750 2.1750 2.1750 2.6100 2.6100 2.6100]'; %
maximal joint velocities
            robot.v_max=[1.5 1.5 1.5 2 2 2]'; % maximal task velocities
        end
    end

    methods
        %-----------------------------------------------------------------
        % Kinematic model methods
        %
        function [p,R,J]=kinmodel(robot,q)
            %[p,R,J]=kinmodel(q) Kinematic model
            if nargin==1
                q=robot.q;
            end
            tcp=[robot.TCP(1:3,4)' r2rpy(robot.TCP(1:3,1:3))];
            [p,R,J]=kinmodel_panda(q,tcp);
        end
    end % methods
end
```

Robot classes for ROS

One possibility to communicate with robots and robot model in different simulation environments is by using ROS framwork. ROS is an open source robotics middleware suite which provides a collection of tools that aim to simplify the task controlling the robots across a wide variety of robotic platforms.

RBS does not use all the possibilities offered by ROS, but only those that are necessary to communicate with the robot, i.e. reading the status of the robot and sending the desired movement to the robot. In addition, it uses ROS tools to launch and change controllers. For this functionality RBS Toolbox uses ROS subscribers, publishers and services. In the communication it is not using only standard message and service types, but also some custom messages, especially for controllers.

For example, to receive robot state we can use a ROS subscriber

```
robot.states_sub =
rossubscriber('/franka_state_controller/franka_states',@robot.GetStateCallback);
```

where the callback function is defined as

```
function GetStateCallback(robot,~,msg)
    robot.state = msg;
    robot.t_last_states_call=toc(robot.last_callback);
    robot.last_states_callback=tic;
end
```

and the robot state is defined using the message `msg`.

To control the motion of a robot we define a ROS publisher:

```
robot.command_pub =
rospublisher('/joint_impedance_controller/command','robot_module_msgs/JointComman
d');
```

use the custom message defined for the particular control

```
cmd_msg = rosmessage(robot.command_pub);
```

and when all variables in the message are updated, we send the m,essage to the controller

```
cmd_msg.Pos = q;
cmd_msg.Vel = qdot;
cmd_msg.Trq = trq;
cmd_msg.Impedance.N = 7;
cmd_msg.Impedance.K = robot.joint_compliance.K;
cmd_msg.Impedance.D = robot.joint_compliance.D;
send(robot.command_pub,cmd_msg);
```

> Note that custom ROS messages have to be built in Matlab (see E1Wiki-Custom ROS messages Matlab).

Currently RBS provides classes for:

- robot Franka Emika Panda,
- gripper for Franka Emika Panda,
- robot UR10,
- Intel RealSense camera,

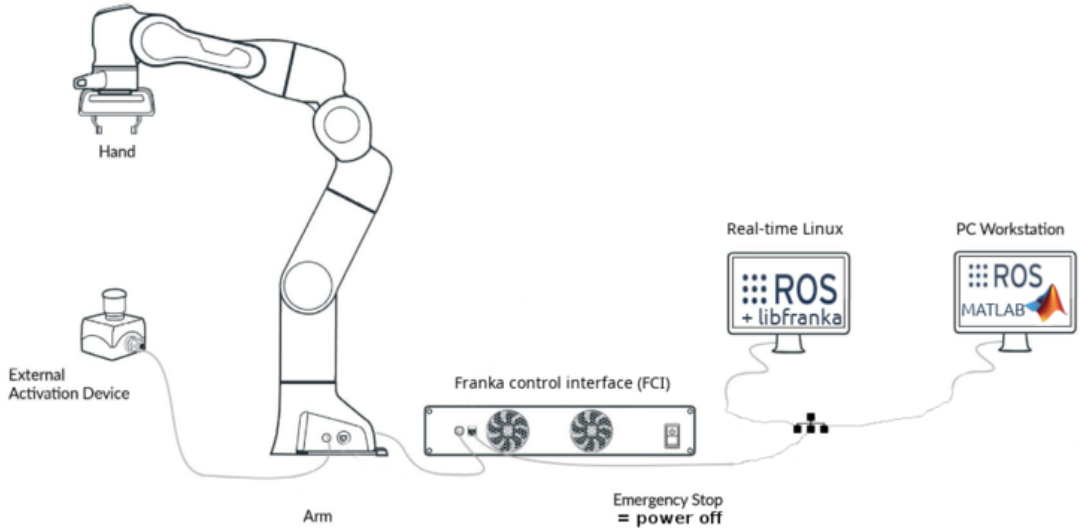
- standard USB camera, and
- power switch.

Before working with ROS in Matlab it is necessary to connect to to the ROS master using the command

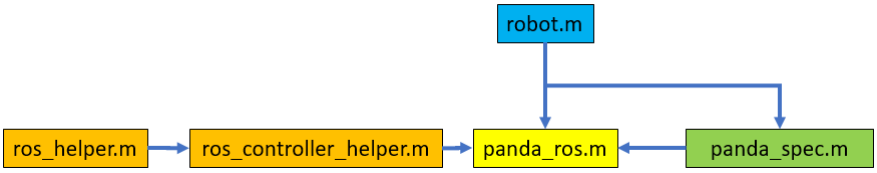
```
rosinit('host')  % host is host name or IP address of ROS master
```

Robot class for ROS controlled Franka Emika Panda

Collaborative robot Franka Emika Panda has been integrated into RBS Toolbox using the Franka Control Interface (FCI) ROS integration. FCI provides the current status of the robot and enables its direct control with an external workstation PC connected via Ethernet. In addition, `franka_ros` connects robots with the entire ROS.



The `panda_ros` class has the structure as shown on figure



To use the Franka Emika Panda robot it is necessary to launch the ROS controller (for more information see E1Wiki). Currently, the `panda_ros` class supports four controllers:

- joint position controller (`JointPosition` , `JointVelocity`),
- joint impedance controller (`JointImpedance`),
- Cartesian position controller (`CartesianPose` , `CartesianVelocity`), and
- Cartesian impedance controller (`CartesianImpedance`).

For example, to lauch joint impedance controller use

```
roslauch ijs_controllers joint_impedance.launch load_gripper:=true
```

Switch between different controllers is done by using methods `SetStrategy`, `GetStrategy` and `isStrategy`, which use ROS controller manager and methods defined in subclass `ros_controller_helper.m`. The Joint-space controllers support joint movements (`JMove`, ...). For all task-space movements (`CMove`, `OMove`, ...) internal RBS kinematic controller is used (see section Task-space movement methods). On the other side, the Cartesian controlles support only task-space movement commands. So, to move in joint-space, it is necessary to switch to one of joint controllers.

The impedance controllers are not standard controllers defined in *franka_control* but custom made controllers. To control the stiffness and damping of these controlles several methods are available

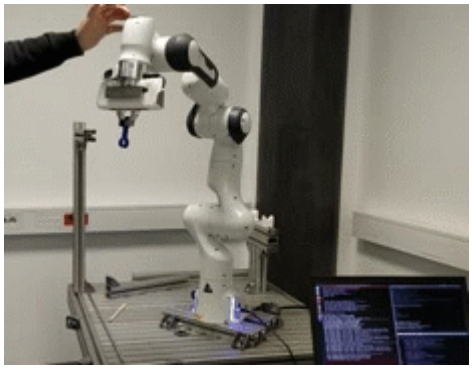
Method	Description
`[val]=GetJointStiffness`	Get current joint stiffness
`SetJointStiffness(stiffness,...)`	Set joint stiffness
`[val]=GetJointDamping`	Get current joint damping
`SetJointDamping(damping,...)`	Set joint damping
`SetJointSoft(softness,...)	Set stiffness level: 0:compliant; 1:stiff
`[K,D]=GetJointCompliance`	Get joint compliance gains
`SetJointCompliance(K,D,...)`	Set joint compliance gains
`[val]=GetCartesianStiffness`	Get current cartesian stiffness
`SetCartesianStiffness(stiffness,...)`	Set cartesian stiffness
`[val]=GetCartesianDamping`	Get current cartesian damping
`SetCartesianDamping(damping,...)`	Set cartesian damping
`SetCartesianSoft(softness,...)`	Set stiffness level: 0:compliant; 1:stiff
`[Kp,Kr,R,D]=GetCartesianCompliance`	Get Cartesian compliance gains
`SetCartesianCompliance(Kp,Kr,...)`	Set Cartesian compliance gains
`SetPositionCompliant`	Set Cartesian position compliant

and some of them can have optional parameters

Parameter	Description
`HoldPose`	Holds current pose (`on` - default) or slowly increases stiffness (`off`)
`R`	Rotation of position stiffness frame (only for Cartesian impedance)
`D`	damping factor for rotaions (only for Cartesian impedance)

For example, when the robot is made more compliant, it can move more due to external forces. In practice, the steady-state displacement from the commanded position depends on the stiffness gain, friction and external forces. When the robot is made more stiff, the robot will decrease the displacement from commanded positions. The behaviour when the robot becomes stiffer is controlled by optional parameter `HoldPose`. When `HoldPose` is `on` then the robot does not move when stiffnes is increased. On the other hand, when `HoldPose` is `off` then the robot moves toward the commanded position.

```
r=panda_ros;
r.JMove(r.q_home,1)
r.SetJointSoft(0.001) % Robot is compliant
r.Wait(1)
r.SetJointSoft(1) % Robot becomes stiff and preserves the current position
r.Wait(1)
r.JMove(r.q_home,1)
r.SetJointSoft(0.001) % Robot is compliant
r.Wait(1)
r.SetJointSoft(1,'HoldPose','off') % Robot becomes stepwise stiff and moves
toward q1
```



In subclass `panda_ros.m` additional methods are included by redefining standard methods or new methods for specific functionality of Panda robot.

When the robot is very compliant, its behavior depends significantly on quality of gravity compensation. For that it is necessary to set the correct load of the robot. The TCP of the end-effector and the load can be defined using the methods

Parameter	Description
`SetTCP(TCP,...)`	Sets the tool centre point
`SetLoad(...)`	Sets the end-effector load parameters

Method `SetTCP` has two optional parameters how the desired TCP is applied.

Parameters	Values	Comment
`TCPFrame`	`'Robot'`,`'Gripper'`	TCP is set relative to flange or gripper
`EEFrame`	`'Flange'`, `'Nominal'`	Frame used for TCP in `franka_lib` (`'Nominal'` has to be used for `franka_lib` ver 8)

*Defaults are bold.

The `TCPFrame` parameter is used to select wheather TCP input parameter is relative to the frame defined by the robot parameter `TCPGripper` or not. So, the following two commands yield the same TCP

```
>> r.SetTCP(rp2t(rot_x(pi/2),[0 0.1 0]'))
>> r.TCP
ans =
    0.7071    0.0000   -0.7071    0.0707
   -0.7071    0.0000   -0.7071    0.0707
         0    1.0000    0.0000    0.1034
         0         0         0    1.0000
>> r.SetTCP(r.TCPGripper*rp2t(rot_x(pi/2),[0 0.1 0]'),'TCPFrame','Robot')
>> r.TCP
ans =
    0.7071    0.0000   -0.7071    0.0707
   -0.7071    0.0000   -0.7071    0.0707
         0    1.0000    0.0000    0.1034
         0         0         0    1.0000
```

Note that the method `SetTCP` in class `panda_ros.m` changes also the TCP in the robot controller. As in the new version of `franka_lib` the behavior of setting the TCP has been changed, it is necessary to use for version 0.8.0 the `EEFrame` parameter with value `'Nominal'`.

The load of the robot end-effector can be defined using the method `SetLoad` with three optional parameters defining the load:

Parameter	Description
`Mass`	Load mass
`COM`	Center of mass
`Inertia`	Load inertial tensor

Since human operators are expected to physically interact with the Panda robot, strict safety measures ar important. The Panda robot provides on the low level safety-like feature, i.e. a possibility to set torque boundaries for each joint of the robot and force/torques boundaries defined in Cartesian space. These boundaries are defined for contact and collision separately and the difference lies in the way the robotic arm reacts. Contact is triggered only when the user applies a force or torque whose magnitude lies between the lower and upper thresholds, and collision is triggered when that force or torque is higher the higher threshold. To set/get these forces and torques, we provide following methods

Method	Description
`[F,T,tq]=GetCollisionBehaviour`	Get current collision thresholds
`SetCollisionBehaviour(F,T,tq)`	Set nominal collision thresholds
`[qcol,xcol]=GetCollisions(robot)`	Get current joint and Cartesian collision status
`[F,T,tq]=GetContactBehaviour`	Get current contact thresholds

Method	Description
`SetContactBehaviour(F,T,tq)`	Set nominal contact thresholds
`[qcon,xcon]=GetContacts(robot)`	Get current joint and Cartesian contact status

When a collision is registered, the robotic arm stops and a reset is required. Whenever the Panda robot stops in an error condition, it is possible to recover from errors and reflexes by calling the `ErrorRecoveryAction`. The methods for checking for errors and to recover from them aremethod

Method	Description
`[val,msg]=Check`	Check for controller errors and throw error
`status = HasError`	Check if in Move mode without error
`ErrorRecovery`	Resets error or reflex and eables robot control

Robot class for gripper for Franka Emika Panda

The RBS Toolbox provides following methods to control the Panda gripper :

Method	Description
`succ=Open(gripper,...)`	Open gripper
`succ=Close(gripper,...)`	Close gripper
`Homing(gripper)`	Homes the gripper and updates the maximum width given by the mounted fingers.
`Stop(gripper)`	Stops a currently running gripper move or grasp
`succ=Move(gripper,d,...)`	Moves to a target width with the defined speed
`succ=Grasp(gripper,...)`	Tries to grasp at the desired width with desired force while closing with the given speed
`succ=GraspInwards(gripper,...)`	Tries to grasp at the desired width with desired force while closing with the given speed
`succ=GraspOutwards(gripper,...)`	Tries to grasp at the desired width with desired force while closing with the given speed

Some of the methods have optional parameters to define the motion of the gripper fingers or grasp:

Parameter	Description	Default value
`Width`	Gripper width	0 [m]
`Force`	Grasping force	10 [N]
`Speed`	Moving speed	0.1 [m/s]

Parameter	Description	Default value
`Eps`	Width tolerance in grasp	0.005 [m]
`Timeout`	ROS timeout for reporting sucsess	2 [s]
`Check`	Check final width between fingers	`'on'`

Methods `GraspInwards` and `GraspInwards` are equal to `GraspInwards`, except for additional offset of 0.005m in target gripper width regarding the grasp direction.

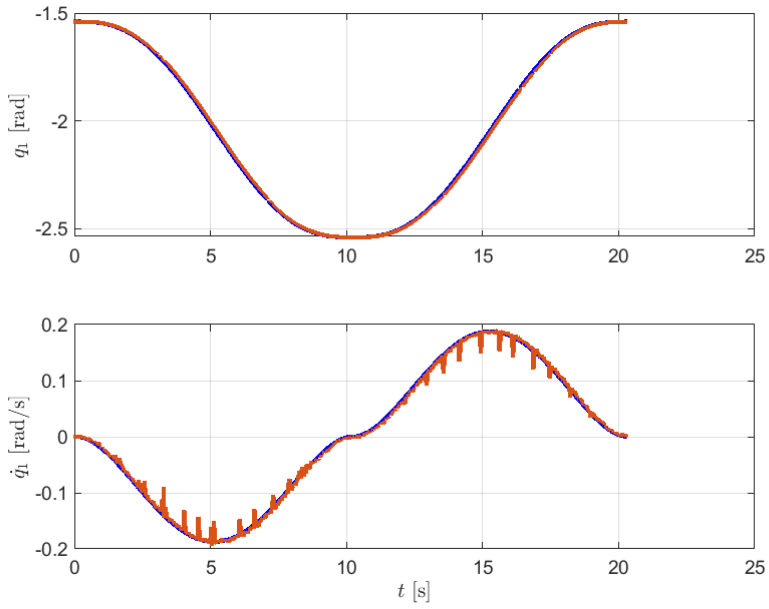
Robot class for ROS controlled UR10

To use the UR10 robot with ROS control it is necessary to launch the corresponding ROS controller. Currently, the `ur10_ros` class supports only one controller, i.e., `pos_joint_traj_controller`. To switch to this controller use a corresponding RAD command, e.g.

```
root@bambus-3:/ur_ws# rosservice call /controller_manager/switch_controller
"start_controllers: ['pos_joint_traj_controller'] stop_controllers:
['scaled_pos_joint_traj_controller']
strictness: 2
start_asap: false
timeout: 0.0"
```

The subclass `ur10_ros.m` redefines standard methods and defines some new methods for specific functionality of UR10 robot.

The RBS uses for joint motion and for task-space motion based on RBS kinematic controller the low-level function `GoTo_q`, which sends reference joint positions at each sample to the robot controller. When using internal UR10 internal controller `pos_joint_traj_controller` there might occur some vibrations during motion execution, e.g. see joint 1 velocity on the figure



To minimize vibrations we propose to set the sampling rate to 125Hz, which is the UR10 rate to receive commands.

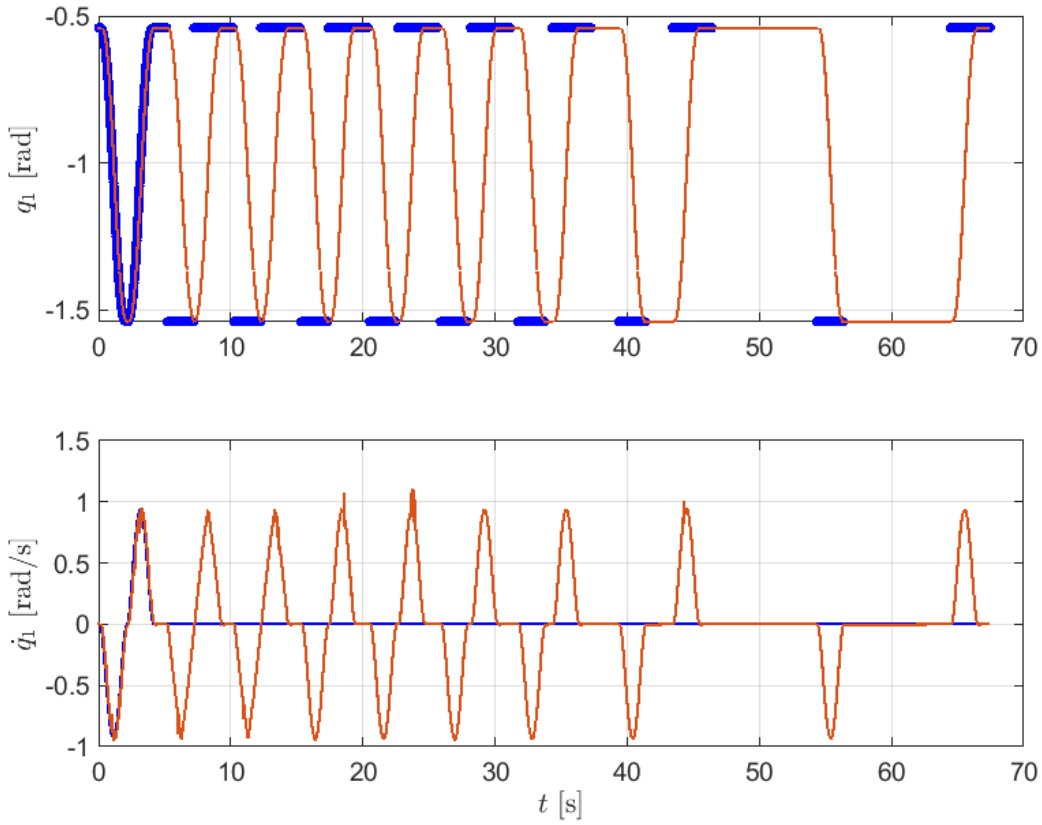
For executing motion in joint space motion using commands based on `JMove` command there is possibility to exploit the UR10 joint position trajectory controller. For that it is necessary to send to the `pos_joint_traj_controller` all the points defining the joint position trajectory. For that, `JMove` and related commands have an additional option `'Ref'`.

Option	Values	Comment
`'Ref'`	`'Points'`,`'Traj'`,`'PTP'`	Value `'Points'` (default) means that RBS generates joint position for each sample, value `Traj` means that selected number of points on the generated trajectory are sent to the UR10 controller as trajectory points (additional option 'NumPoints' defines the number of points), and the value `'PTP'` means that the motion to the final position is generated by the UR10 trajectory controller.
`'NumPoints'`	[2 .. n]	Number of points on trajectory

For example, the following commands show how to use these options.

```
r=ur10_ros;
q1=r.q;
q2=q1-[1 0 0 0 0 0]';
t_move=2;
r.JMove(q2,t_move)
r.JMove(q1,t_move)
r.Wait(1)
r.JMove(q2,t_move,'Ref','PTP')
r.JMove(q1,t_move,'Ref','PTP')
r.Wait(1)
np=[2 10 20 30 50 100 200];
for i=1:length(np)
    r.JMove(q2,t_move,'Ref','Traj','NumPoints',np(i))
    r.JMove(q1,t_move,'Ref','Traj','NumPoints',np(i))
    r.Wait(1)
end
```

First the motion between q1 and q2 is made using default trajectory generation, the PTP motion is used and the last sequences define the trajectory using different number of points. The resulting motion is:



Note the there is a delay between the `JMove` command and the actual start of the motion due to the preparation of the ROS message for the `pos_joint_traj_controller`. In avarage, each point generation for ROS message takes about 5ms,

ROS class for Intel RealSense camera

To use the Intel RealSense camera in Matlab over ROS it is necessary first to run the corresponding ROS aplication:

```
roslaunch realsense2_camera rs_camera.launch
```

Then we can create the Matlab object and capture an image. Currently, the RBS class supports ony raw images.

```
cam=realsense_ros;
image=cam.getsnapshot;
```

ROS class for USB camera

To use the USB camera in Matlab over ROS it is necessary first to run the corresponding ROS aplication:

```
roslaunch usb_cam usb_cam-test.launch video_device:=/dev/video1
```

Then we can create the Matlab object and capture an image.

```
cam=usbcam_ros;
image=cam.getsnapshot;
```

Robot class for power switch

The RBS Toolbox provides a class to work with power-switch module, i.e. is a periphery device that controls mains power switch using raspberry GPIO interface. To use the power-switch module in Matlab over ROS it is necessary first to run the corresponding ROS aplication:

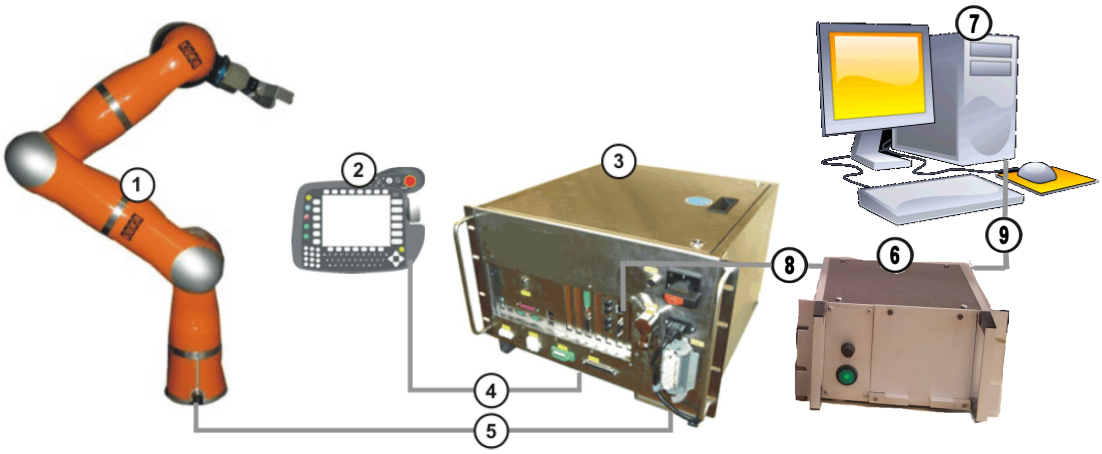
```
sshpass -p ubuntu ssh ubuntu@power-switch-module
catkin_ws/src/rpigpio_ros/util/supervisor.sh
```

Then we can create the Matlab object and control the switch.

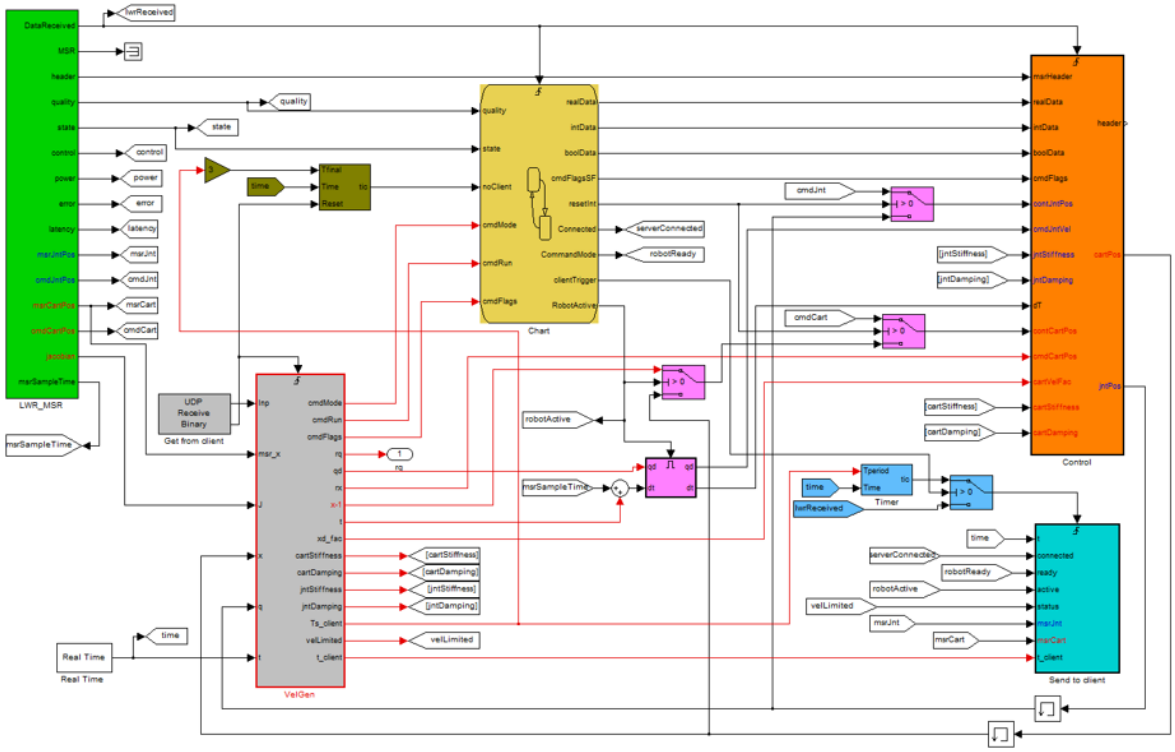
```
sw=gpio_power_ros;
sw.SetPower(1)
sw.SetPower(0)
```

Robot classes for KUKA LWR

The robot is controlled by the KUKA controller using the integrated scripting language KRL. Additionally, it has implemented the Fast Research Interface (FRI), which enables to interact with the robot controller. The FRI enables basically only the position control of the robot, but the user can add to the control signal also additional joint torques or task force/torques. The FRI communicates with the KUKA controller using UDP socket communication. The user has to establish a reliable UDP socket communication at prescribed sampling rate. Although the FRI allows different sampling rates, a computer with real-time operating system that generates UDP packets in real-time with selected sampling rate is required. For many robotics research topics, like motion planning or task generation at higher levels the update rates are usually relatively low and they are not always in "real-time", i.e. the time between two samples may vary more than it is required by the FRI communication protocol. To make the interaction with LWR over FRI possible also in such situations, we have developed a FRI-xPC server, which operates in real-time. The following figure shows the KUKA LWR robot system overview (1: LWR Robot, 2: Teach pendant KCP, 3: Robot controller with FRI, 4: Connection controller/pendant, 5: Connection controller/robot, 6: FRI-xPC server, 7: Client computer where MATLAB and RBS Toolbox are running, 8: connection FRI-xPC/controller (500Hz UDP), and 9: connection FRI-xPC/Client (approx. 100 HZ UDP):

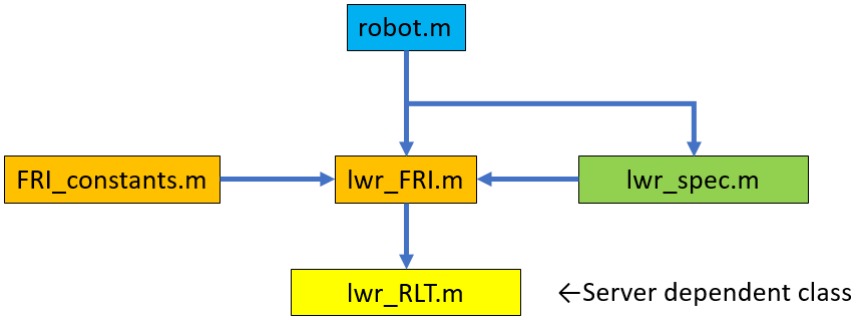


The FRI-xPC server is implemented on a computer using MATLAB xPC Target real-time platform. The server is build using the Simulink, Stateflow and some user defined blocks as shown in the figure



The RBS Toolbox provides robot classes to connect to the xPC Target server. The IJS platform consists of two LWR arms and a Torso

There exist several xPC Target servers, which are prepared for different robot configurations and types of control (e.g. independent control of left and right LWR robot, both LWR robots as dual-arm robot system, or robots and torso as one system, etc.). Each of these servers uses different structure of communication packets. Hence, it is necessary to have different robot classes. All classes have the structure as shown on figure



All common specific parameters and methods for LWR robots classes are defined in subclass `lwr_FRI.m` (sublass `FRI_constants.m` defines only common FRI constants). The difference is only in server dependent class, which defines the particular robot class, e.g. `lwr_RLT.m`. This subclass includes the definition of communication packets used by a particular server and parsing functions for these packets. Currently, RBS toolbox provides classes for lwr_RLT and lwr_RL xPC target.

The KUKA LWR robots support over FRI different control modes (strategies):

- Monitor mode (0) and
- Control mode:
  - Joint position (1)
  - Cartesian Impedance (2), and
  - Joint impedance (3).

For control modes operations the Toolbox provides several functions.

Method	Description
`[strategy,mode]=GetStrategy` `[mode,strategy]=GetCommandMode`	Get control strategy
`SetStrategy(strategy,flags)` `SetCommandMode(mode,flags)`	Set current control strategy and flags (if flags are not defined, default flags for each mode are used)
`val=isStrategy(strategy)`	Check if strategy is active
`val=isMonitorMode`	Check if in Monitor mode
`SetMonitorMode`	Set the robot into Monitor mode
`val=isCommandMode`	Check if in Command mode
`[flags,s]=GetCommandFlags`	Returns current command flags

When changing control modes it is necessary to select proper flags. The main control fags are:

Flag	Value	Data used for control	Feasible for control modes
`FRI_CMD_JNTPOS`	`0x0001`	Desired joint positions	1, 2, 3
`FRI_CMD_JNTTRQ`	`0x0004`	Superposed joint torques	3

Flag	Value	Data used for control	Feasible for control modes
`FRI_CMD_JNTSTIFF`	`0x0010`	Desired joint stiffness	2, 3
`FRI_CMD_JNTDAMP`	`0x0020`	Desired joint damping	2, 3
`FRI_CMD_CARTPOS`	`0x0100`	Desired Cartesian position	2
`FRI_CMD_TCPFT`	`0x0400`	Superposed end-effector forces and torques	2
`FRI_CMD_CARTSTIFF`	`0x1000`	Desired Cartesian stiffness	2
`FRI_CMD_CARTDAMP`	`0x2000`	Desired Cartesian damping	2

For security reasons the xPC target server has additional flag for allowing motion commands, which is controlled using following methods.

Method	Description
`val=isReady`	Checks if connection to robot controller is ready
`val=isActive`	Checks if robot is prepared to be moved
`Stop`	Disables update of robot motion
`Start`	Enables update of robot motion

These methods are used internally in all motion commands to assure secure operation of the robot and it is not necessary to use them for common operations. However, if for some reason the communication form client to xPC target server is interrupted for more the 0.25s or the motion command has been interrupted before the `Stop` has been executed (typically, at the end of motion), it is necessary to enable motion update by using `Stop` before next movemenet method. The proper use of `Start` and `Stop` has to be considered espweciall when using directly the low level motion commands:

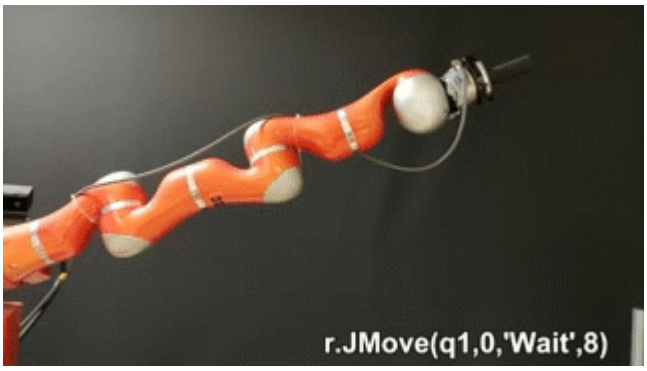
```
r.Stop  % Optional, to assure that xPC target goes into correct internal state
r.Start % Enables motion update
r.GoTo_q(r.q,0*r.q,0*r.q,0) % Low-level motion command
% ... Additional motion commands, with no inter-command delay
r.Stop  % Disebles motion update - xPC target is waiting for next Start
```

Whenever some delay between motion commands is expected, it is necessary to use `Stop` and `Start` combination between motion methods.

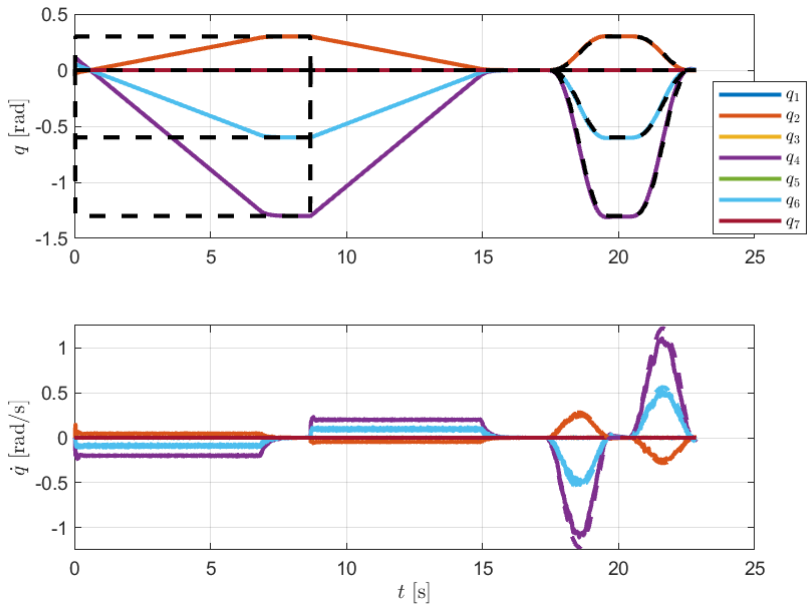
If FRI is in Monitor mode, it is also nesessary to select before the motion commands the desired control strategy.

The xPC target server includes also safety mechanism preventing fast motion when large chages in commanded configuration is sent to robot. In the following example we first command two "discontinous movements" and then smooth movements to the same configurations

```
r=lwr_RLT('TCP',rt2tr(eye(3),[0 0 0.295]'));
q0=zeros(7,1);
q1=[0 0.3 0 -1.3 0 -0.6 0]';
r.SetStrategy('JointPosition')
r.JMove(q1,0,'Wait',8) % Similar to r.GoTo_q(q1,zeros(r.nj,1),zeros(r.nj,1),8)
r.JMove(q0,0,'Wait',8)
r.JMove(q1,2)
r.JMove(q0,2)
```



In the resposes, we can see that joint motion velocity in "discontinous movements" is bounded to relatively low value



The LWR robots support compliant motion. The stiffness and damping can be defined in joint space or in task space using the standard methods

Method	Description
`val=GetJointStiffness`	Get current joint stiffness
`SetJointStiffness(stiffness)`	Set joint stiffness
`val=GetJointDamping`	Get current joint damping
`SetJointDamping(damping)`	Set joint damping
`SetJointSoft(softness)`	Set stiffness level: 0:compliant; 1:stiff

Method	Description
`val=GetCartesianStiffness`	Get current cartesian stiffness
`SetCartesianStiffness(stiffness)`	Set cartesian stiffness
`val=rGetCartesianDamping`	Get current cartesian damping
`SetCartesianDamping(damping)`	Set cartesian damping
`SetCartesianSoft(softness)`	Set stiffness level: 0:compliant; 1:stiff

The behaviour of the robot (stiffness and damping) is changed only if appropriate control flag is set.

When the command flag `FRI_CMD_CARTPOS` is set in control strategy (applicable only for `CartesianImpedance` strategy), then the RBS Toolbox task-space motion commands (like CMove) use internal KUKA robot inherited task-space controller. In all other cases, the task-space motion is controlled by using the internal RBS kinematic controller. The optional parameters regarding the null-space behavoiur of LWR robot only apply to the internal kinematic controller (see section Task-space movement methods).

Note that using `SetTCP` method defines only the TCP of the robot object in Matlab and the TCP has to be set also in the parameters of the KUKA robot controller. If TCP of the robot object and TCP on the real robot are not the same, the task space motion using internal controller and the KUKA inherited controller do not result in the same motion. To check if the TCP is set as on the KUKA robot controller, the outputs of the following commands have to be the same:

```
>> T1=r.GetPose('TaskSpace','Robot','Kinematics','Robot')
T1 =
    0.5370   -0.0198   -0.8433   -0.5737
    0.0123    0.9998   -0.0156   -0.0105
    0.8435   -0.0020    0.5371    0.7212
         0         0         0    1.0000
>> T2=r.GetPose('TaskSpace','Robot','Kinematics','Calculated')
T2 =
    0.5370   -0.0198   -0.8433   -0.5737
    0.0123    0.9998   -0.0156   -0.0105
    0.8435   -0.0020    0.5371    0.7212
         0         0         0    1.0000
>>  T1\T2
ans =
    1.0000         0         0         0
   -0.0000    1.0000         0         0
    0.0000         0    1.0000         0
         0         0         0    1.0000
```

The defualt sampling time is 0.02s and minimal sampling time for xPC target is around 0.0125s (80Hz). Namely the low level motion command takes around 10ms

```
>> r.Stop,r.Start,tic,r.GoTo_q(r.q,0*r.q,0*r.q,0),toc,r.Stop
Elapsed time is 0.009845 seconds.
```

We can check if the selected sampling time is feasible by verifyinf the execution time of a motion command like this

```
>> tic,r.JMove(q0,10),toc  % Expected time 10sec
Elapsed time is 10.402400 seconds.
```

where the elapsed time shoud not be much longer as the commanded motion time (actually, it is little longer due to process time before starting and after finishing the motion).

Robot classes for Haptix simulator

The robot models for MuJoCo Haptix simulator are defined in a XML using MJCF modelling language. To control a robot model in Haptix using RBS Toolbox we have to send to the Haptix simulator commands to actuators and get the information about the robot reading the robot states and sensors.

All common specific parameters and methods for Haptix target are defined in subclass `robot_haptix.m`. Currently, all robot joints in Haptix simulator have to connected to actuators, which are internally position controlled. Therefore, the Haptix targets support on *joint position control strategy*. As a robot model has to define all methods related to control strategies, we use for Haptix target a subclass `robot_no_compliance.m`.

To connect internal variables to Haptix states and sensors we use names of Haptix objects. In the following table are the robot properties and the default Haptix object names based on the robot name `<robot>` defined as the first parameter of `robot_haptix.m` class.

Property	Defualt Haptix name	Description
`JointNames`	`<robot>/joint<i>`	Joint names in Haptix model
`BaseName`	`<robot>`	Base link name in Haptix model (defines robot base frame)
`EEName`	`<robot>/EE`	Site name on the last link representing the end-effector (used to calculate TCP transformation)
`TCPName`	`<robot>/hand`	Site name at the TCP location (used to calculate TCP transformation)
`ActuatorNames`	`<robot>/pos_joint<i>`	Actuator names for joint position control in Haptix model
`SensorPosName`	`<robot>/pos`	EE position sensor name in Haptix model
`SensorOriName`	`<robot>/ori`	EE orientation sensor name in Haptix model
`SensorLinVelName`	`<robot>/v`	EE linear velocity sensor name in Haptix model
`SensorRotVelName`	`<robot>/w`	EE rotational velocity sensor name in Haptix model

Property	Defualt Haptix name	Description
`SensorForceName`	`<robot>/force`	EE force sensor name in Haptix model
`SensorTorqueName`	`<robot>/torque`	EE torque sensor name in Haptix model

If the Haptix robot model does not use the default object names, then it is necessary to define the corresponding properties in the robot class. This can be done in the subclass `<robot>_haptix.m` defining a particular robot using

```
robot=robot@robot_haptix('Panda','TCPName','Panda_gripper');
```

or when defining a robot object

```
r=panda_haptix('TCPName','Panda_gripper')
```

A typical definision of sensors looks like this

```
<sensor>
    <framepos name="Panda/pos" objtype="site" objname="Panda/hand" />
    <framequat name="Panda/ori" objtype="site" objname="Panda/hand" />
    <framelinvel name="Panda/v" objtype="site" objname="Panda/hand" />
    <frameangvel name="Panda/w" objtype="site" objname="Panda/hand" />
    <force name="Panda/force" site="Panda/hand" noise="0.5" />
    <torque name="Panda/torque" site="Panda/hand" noise="0.5" />
</sensor>
```

When the robot model and Haptix model are based on the URDF then the joint names are read from the URDF file. In this case, we can define the robot object as

```
r.talos_arm_haptix('URDF','Talos.urdf','InitialLink','torso_2_link','FinalLink','
arm_left_7_link')
```

or include the URDF parameters in the class `<robot>_haptix.m`

```
classdef talos_arm_haptix < robot_haptix & talos_arm_spec
    methods
        function robot=talos_arm_haptix(varargin)
            opt.URDF='Talos.urdf';
            opt.InitialLink='torso_2_link';
            opt.FinalLink='arm_left_7_link';
            opt=set_options(opt,varargin);
            robot=robot@robot_haptix('Talos',opt);
            robot.Specifications;
            robot.Init;
        end
    end % methods
end % classdef
```

The `robot_haptix.m` class defines also some for Haptix target specific methods

Method	Description
`T=GetRobotBase`	Get the pose of robot base and update `TBase` using body "RobotBase"
`SendRobot_q(q)`	Send desired position to actuators. In simulation mode one intergration step in initiated.
`SendCtrl(u)`	Update all actuators. In simulation mode one integration step in initiated.
`SendAuxCtrl(idx,val)`	Send specific control values
`GetAuxJointPos(id)`	Get joint positions for specific joints
`SetAuxJointPos(id,q)`	Send specific joint positions
`GetSensor(id)`	Get sensor data for specific sensor
`F=GetContacts(id)`	Get contacts for specific geom
`SetMocapPose(id,T)`	Set the pose of a mocap body
`GetMocapPose(id,...)`	Get pose of a mocap body
`GetObjectPose(typ,id,...)`	Get object pose
`AnimationMode`	Haptix is used only to animate robot
`SimulationMode`	Dynamic simulation in Haptix without integration. Integration steps are activated by robot object.
`RealTimeMode`	Dynamic simulation in Haptix.
`GetSimulationMode`	Get current simulation mode
`HaptixMessage(msg)`	Message to be displayed in Haptix simulator

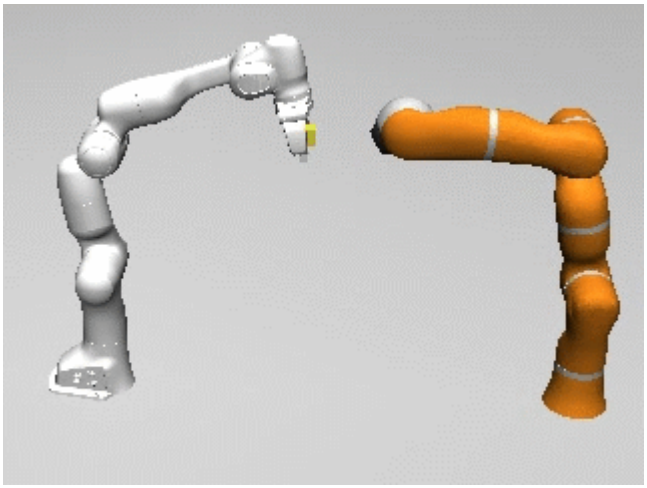
Such naming convention allows to control more robots in one scene (due to MATLAB limitation only one at a time)

```
r1=panda_haptix;
r2=lwr_haptix;
r2.SetMocapPose(r2.BaseName,rp2t(rot_z(pi),[1.2 0 0]')) % Position robot r2
r2.GetRobotBase; % Update the new robot base

r1.JMove(r1.q_home,1)
r2.JMove(r2.q_home,1)
T0=rp2t(rot_y(pi/2)*rot_z(pi),[0.5 0 0.7]');
T1=rp2t(rot_y(pi/2)*rot_z(pi),[0.4 0 0.7]');
T2=rp2t(rot_y(-pi/2)*rot_z(pi),[0.6 0 0.7]');
r1.SetMocapPose('Target',T0)  % Position the target mocap body

r1.CMove(T1,2)
r2.CMove(T2,2)
r1.TMove([0 0 0.1]',1)
```

```
r2.TMove([0 0 0.1]',1)
```

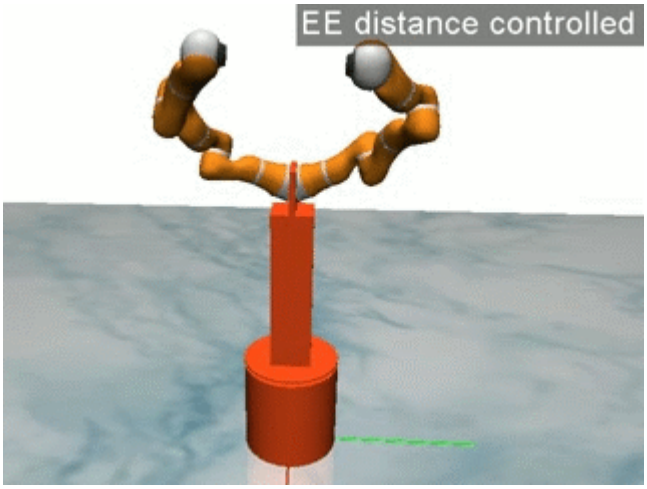


To move two robot arms simultaneously it is necessary to define a new robot class. For example, for a dual arm robot where only a relative motion between end-effectors is important, we can define the robot system by joining two robot arms. The base and the end-effector of dual arm robot is defined in the kinematics method, which is one of the important methods of a new robot class (the whole class definition is given in `duallwr_haptix.m`.

```
function [p,R,J]=kinmodel(robot,q)
    qR=q(1:7);
    qL=q(8:14);
    [Rb,xb]=t2rp(robot.T_RL);
    [p1,R1,J1]=kinmodel_lwr(qR,robot.rR.TCP); % First robot
    [p1r,R1r,J1r]=RobotReverse(p1,R1,J1);  % From EE to base
    [p1f,R1f,J1f]=JoinRobotFixed(p1r,R1r,J1r,xb,Rb); % bases
    [p2,R2,J2]=kinmodel_lwr(qL,robot.TCP); % Second robot
    [p,R,J]=JoinRobot(p1f,R1f,J1f,p2,R2,J2); % Added second robot
end
```

This robot class allows to control the relative motion between the robot end-effectors.

```
r=duallwr_haptix;
r.JMove(r.q_home,1)
r.HaptixMessage('EE distance controlled')
r.CMove(rp2t(rot_x(pi),[0 0 0.4]'),1)
r.CMove(rp2t(rot_x(pi),[0 0 0.2]'),1)
r.Wait(0.5)
r.HaptixMessage('NS motion')
qopt=[-0.32 1.36 0.4 -0.96 0.08 0.5 -0.63 0.5 0.32 -0.1 -1.55 -0.4 0.96 0.8]';
r.CMove(rp2t(rot_x(pi),[0 0
0.2]'),1,'NullSpaceTask','ConfOptimization','q_opt',qopt)
r.Wait(0.5)
r.CMove(rp2t(rot_x(pi),[0 0 0.2]'),1)
r.HaptixMessage('EE distance controlled')
r.CMove(rp2t(rot_x(pi),[0 0 0.4]'),1)
```



Robot classes for RoboWorks simulator

The robot models for RobotWorks simulator are modelled directly in RoboWorks. RobotWorks simulator is not a dynamic simulator but can be used to animate the motion of objects. To control a robot model in RoboWorks using RBS Toolbox we have to send to the RoboWorks simulator commands to control the translations and rotations representing the joints of a robot mechanism. Each transformation is identified by a *tag*. To get the information about the robot configuration it is possible to read the values transformations.

All common specific parameters and methods for RoboWorks target are defined in subclass `robot_roboworks.m`. As RoboWorks targets allow only animation, the only allowed control strategy is *joint position control strategy*. As a robot model has to define all methods related to control strategies, we use for RoboWorks target a subclass `robot_no_compliance.m`.

The communication between MATALB and Roboworks is over TCP on port defined by the name of the simulated scene. Therefore, it is necessary to define the RoboWorks scene when creating a robot object (an instance of robot class) .

```
r=panda_roboworks('Panda_Oven');
```

To connect internal variables to RoboWorks states we use tag names of RoboWorks transformations. In the following table are the robot properties and the default RoboWorks tag names.

Property	Defualt RoboWorks name	Description
`JointTagNames`	`'q1,q2,q3,q4,q5,q6,q7'`	Joint tag names
`BaseTagNames`	`'x,y,z,roll,pitch,yaw'`	Robot base transformation tags (define robot base frame)
`TCPTagNames`	`'tcpx,tcpy,tcpz,tcproll,tcppitch,tcpyaw'`	Tag names of transformations between robot end-effector and TCP (used to calculate TCP transformation)

If the RoboWorks robot model does not use the default object names, then it is necessary to define the corresponding properties in the robot class. This can be done in the subclass `<robot>_roboworks.m` defining a particular robot using

```
robot=robot=robot@robot_roboworks(scn,'JointTagNames','q11,q21,q31,q41,q51,q61');
```

or when defining a robot object

```
r=panda_roboworks('Panda_Oven','JointTagNames','q11,q21,q31,q41,q51,q61')
```

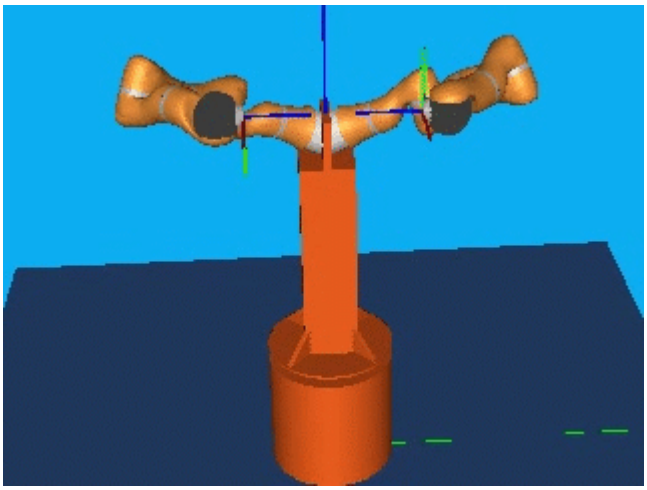
The `robot_roboworks.m` class defines also some for RoboWorks target specific methods

Method	Description
`T=GetRobotBase`	Get the pose of robot base and update `TBase` using "BaseTagNames"
`SetTCP(TCP,...)`	Set the TCP pose
`send_cmd(q)`	Send joint positions
`vals=get_val(tags)`	Get tag values
`set_val(tags,vals)`	Set tag values

Such naming convention allows to control more robots in one scene (due to MATLAB limitation only one at a time). To define two or more robot objects for one scene it is necessary to use for all robots optional parameter `'Connected','yes'`, except the first one.

```
rR=lwr_roboworks('LWR_2_Torso');
rL=lwr_roboworks('LWR_2_Torso','Connected','yes',...
    'JointTagNames','q11,q21,q31,q41,q51,q61,q71',...
    'BaseTagNames','x1,y1,z1,roll1,pitch1,yaw1',...
    'TCPTagNames','tcpx1,tcpy1,tcpz1,tcproll1,tcppitch1,tcpyaw1');
rR.q_home=[-0.1658 0.4065 0.0641 -1.4771 0.3629 0.8605 0.1421]';
rL.q_home=[-0.0190 0.4179 0.1912 -1.4809 -0.5190 0.8841 -0.0365]';

rR.JMove(rR.q_home,1)
rL.JMove(rL.q_home,1)
rL.CApproach(rR.T*rp2t(rot_x(pi)),[0 0.25 0]',1)
rL.CApproach(rR.T*rp2t(rot_x(pi)),[0 0.05 0]',1)
rL.TMove([0 0 -0.2]',1)
rR.CApproach(rL.T*rp2t(rot_x(pi)),[0 -0.05 0]',1)
rR.TMove([0 0 -0.2]',1)
```

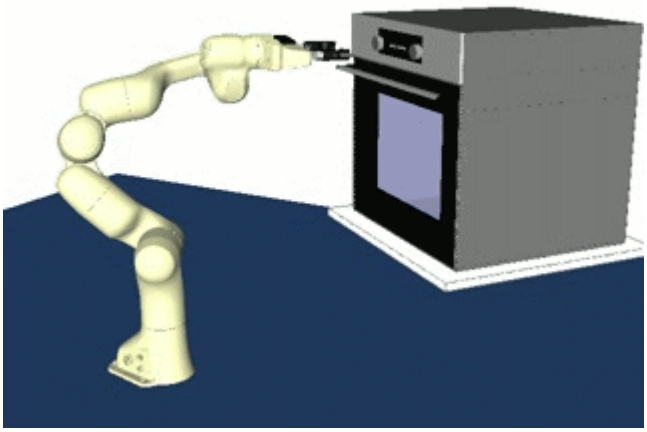


For synchronous motion of robots it is necessary to define a new robot class, where robots are treated as one system (as for Haptix target).

Using auxiliary methods it is possible to control the motion of other object in RoboWorks scene. For example, using `set_val` method and motion callback

```
function stat=Vrata_tracking(robot)
rp=robot.GetPos('TaskSpace','Object');
dp=rp-robot.user.Vrata.PosTecaj;
dp=dp/norm(dp);
fi=-atan2(dp(3),dp(1));
fi=fi+robot.user.Vrata.Offset;
robot.set_val('vrata',fi);
stat=0;
```

we can control an object during robot motion (see demo `Demo_Panda_Oven.m`.



Robot classes for CoppeliaSim simulator

The robot models for CoppeliaSim simulator are modelled directly in CoppeliaSim. To control a robot model in CoppeliaSim using RBS Toolbox we have to send to the CoppeliaSim simulator commands to control the translations and rotations representing the joints of a robot mechanism. Each transformation is identified by a *tag*. To get the information about the robot configuration it

is possible to read the joint values using *tags*.

All common specific parameters and methods for CoppeliaSim target are defined in subclass `robot_coppelia.m`. Currently, the only allowed control strategy is *joint position control strategy*. As a robot model has to define all methods related to control strategies, we use for CoppeliaSim target a subclass `robot_no_compliance.m`.

The communication between MATALB and CoppeliaSim is over TCP (for more info got to CoppeliaSimHelp).

```
r=panda_coppelia;
```

To connect internal variables to CoppeliaSim states we use tag names of CoppeliaSim. In the following table are the robot properties and the default CoppeliaSim tag names.

Property	Defualt CoppeliaSim name	Description
`BaseTagName`	`'<robot>'`	Robot base tag (defines robot base frame)
`JointTagNames`	`<BaseTagName>_joint<i>`	Joint tag names

If the CoppeliaSim robot model does not use the default object names, then it is necessary to define the corresponding properties in the robot class. It possible to define only `BaseTagName` and then the `JointTagNames` are generated automatically. If this is not appropriate, we have to define `JointTagNames`. This can be done in the subclass `<robot>_coppelia.m` defining a particular robot using

```
opt.JointTagNames='Jaco_joint1','Jaco_joint2','Jaco_joint3','Jaco_joint4','Jaco_j
oint5','Jaco_joint6'};
robot=robot@robot_coppelia(client,opt);
```

or

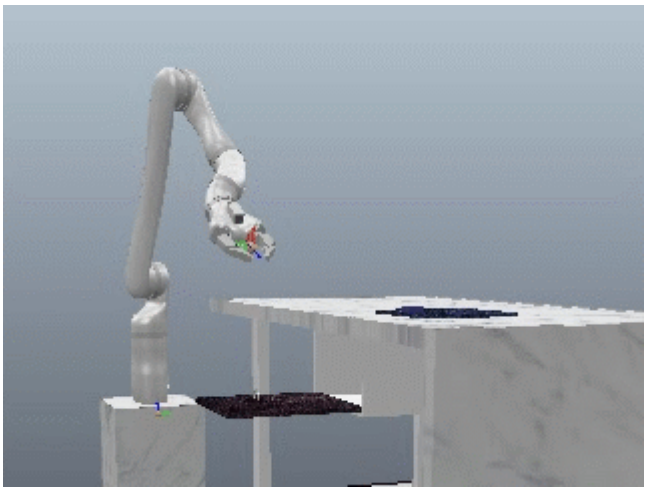
```
robot=robot@robot_coppelia(client,'JointTagNames',{'Joint1',...,'Joint7'}); %
Cell array
```

or when defining a robot object

```
r=panda_coppelia([],'BaseTagName','Panda1','JointTagNames',
{'Joint1',...,'Joint7'})
```

Here is an examle of simulation in CoppeliaSim:

```
r=jaco2_coppelia('BaseTagName','Jaco',IP','178.172.42.81');
r.SetTCP(rp2t(rot_z(pi/2)));
r.JMove(r.q_home,1)
X=[-0.1354 0.2031 1.0343 -0.0280 0.8347 0.0253 0.5494;...
    -0.1354 0.2031 0.8300 -0.0280 0.8347 0.0253 0.5494;...
    -0.1354 0.2031 0.7793 -0.0280 0.8347 0.0253 0.5494;...
    -0.1351 0.5162 1.0003 -0.0280 0.8347 0.0253 0.5494;...
    -0.1351 0.5162 0.9463 0.0900 -0.8322 0.0694 -0.5427];
for i=[1 2 3 2 1 4 5 4 1 2 3 2 1]
    r.CMove(X(i,:),2)
end
```



The `<robot>_coppelia.m` class defines also some for CoppeliaSim target specific methods (Actually, the CoppeliaSim provides more functionality, which is currently not supported by RBS Toolbox).

Method	Description
`T=GetRobotBase`	Get the pose of robot base and update `TBase` using "BaseTagName"
`send_joint_pos(q)`	Send joint positions
`T=GetObjectPose(TagName)`	Get object pose
`SetObjectPose(TagName,T)`	Set object pose

Such naming convention allows to control more robots in one scene (due to MATLAB limitation only one at a time). To define two or more robot objects for one scene it is necessary to use the `Client` parameter of the first defined robot for all other robots as input parameter when defining a new robot object.

```
r1=panda_coppelia;
r2=panda_coppelia('Client',r1.Client,'BaseTageName','Panda1');
```

> Note that some models in CoppeliaSim do not have the same kinematics as real robots. Therefore, the kinematic models have to be modified accordingly.

Simulink library

Simulink is a MATLAB-based graphical programming environment for modeling, simulating and analyzing dynamical systems. Unlike Matlab, which is a script-oriented interpreter, Simulink's primary interface is a graphical block diagramming tool and a customizable set of block libraries. Of course, It offers tight integration with the rest of the MATLAB environment and can either drive MATLAB or be scripted from it.

You can create block diagrams, where blocks represent parts of a system. A block can represent a physical component, a small system, or a function. An input/output relationship fully characterizes a block. In this way Simulink, makes simulations easier and the model of the system you want to simulate is more readable, because it's represented by graphics.

The RBS Simulink `robotblockset` library is a collection of blocks for simulation of robot systems including blocks for:

- robot kinematic and dynamic models,
- joining kinematic models of serial robot arms,
- robot transformations between different task spaces,
- different controllers,
- interfacesfor different animation targets,
- rotations, quaternion operations, and transformations between different rotation representations,
- some matrix transformations, and
- utility functions for real-time UDP communication (only for Matlab 2015b!) and real-time sychronization.