A RTSX Command Reference

This appendix contains RTSX commands and their usages. This material is also available at http://controlsystemslab.com/rtsx/online-help/

The online-help page classifies commands into the following categories. The commands are then listed alphabetically.

Mathematics

Homogeneous Transformation, 3-D Position/Orientation

- ***Rotations***
  - angvec2r, angvec2t, tr2angvec — rotation about an arbitrary axis
  - eul2r, eul2t, tr2eul — euler angle representation
  - rotx, roty, rotz, trotx, troty, trotz — basic rotation matrix about a coordinate axis.
  - rpy2r, rpy2t, tr2rpy — Roll-Pitch-Yaw representation
  - trotw — rotation about a world coordinate
- ***Translations***
  - transl — translation in 3-D
- ***Homogeneous Transformations***
  - Frame — create a coordinate frame structure
  - InsertFrame, DeleteFrame, ReplaceFrame — modify frame in a transformation chain
  - PlotFrame — plot frames in a trasformation chain
  - ResolveFrame, PlotResolveFrame — find/plot homogeneous transformation of missing link in a chain
  - SerialFrame — form a transformation chain from a frame structure
  - t2d, t2r, r2t, rd2t — conversion between rotation matrix R, position vector P and homogeneous transformation matrix T
  - tranimate — frame animation in 3-D

- trinterp — interpolate homogeneous transformation
- trplot — plot a transformed frame in 3-dimension

- ***Quaternion***
  - isquaternion, isqequal — test functions on quaternions
  - q2str, q2tr, q2vec, tr2q — conversion between quaternion and other types of data
  - qadd, qsubtract, qmult, qdivide, et.al — quaternion math operations
  - qinterp — interpolate quaternions
  - Quaternion — create a quaternion data structure

Kinematics

Robot Model Creation and Graphics

- AnimateRobot, AnimateRobotFrame — show a movie of commanded robot movements
- AppendLink, RemoveLink, ReplaceLink — append, remove, or replace a link
- AttachBase, DetachBase, AttachTool, DetachTool — attach/detach a base or tool frame to robot model
- Link — create a robot link from DH parameters
- PlotRobot, PlotRobotFrame — display robot structure/ DH coordinate frames
- Robotinfo — display robot information
- SerialLink — construct a robot from link structure
- UpdateRobot, UpdateRobotLink — add/change robot model and link data

Forward Kinematics

- FKine — Compute forward kinematics from robot model
- Link2AT — Compute homogenous matrices from link data
- Robot2AT — Compute homogenous matrices from robot model
- Robot2hAT — Compute sequence of homogeneous matrices from robot model

Inverse Kinematics

- Ikine — Compute an inverse kinematics solution for a robot model numerically
- Ikine6s — Compute an inverse kinematics solution for a robot with 6 revolute joints such as Puma 560

Velocity Kinematics

- delta2tr, tr2delta — Convert between differential motion and homogeneous transform
- jacob0, jacobn — Compute Jacobian that maps joint velocity to base or end-effector spatial velocity
- tr2jac, eul2jac, rpy2jac — Compute Jacobian from homogeneous transform, euler, or RPY angles

Path Generation

- cpoly, qpoly, lspb — trajectory generation using cubic, quintic polynomials or linear-segment parabolic blend
- ctraj – cartesian trajectory
- jtraj — joint space trajectory
- mtraj, mstraj – multi-axis/multi-segment trajectory

Dynamics

- accel – robot manipulator forward dynamics
- coriolis — compute coriolis matrix
- gravload — compute gravity load in robot dynamics
- inertia — compute inertia matrix
- jfriction, nofriction — get /remove robot friction
- payload — add a payload to robot
- rne — recursive Newton Euler algorithm

Vision

- CamPlot — plot projection of world point on image plane
- CamProject — project world points
- CentralCamera — create a central-projection perspective camera model
- IBVS4 — Image-based visual servoing control for 4-feature points

Robot Control

- Xcos models are included in subdirectory /xcos. (See Chapter 5.)

accel

RTSX Category: Dynamics

Compute forward dynamics of a robot manipulator

Syntax

- `qdd = accel(robot, q, qd, torque)`

Input Arguments

- robot — n-link robot data structure created with `SerialLink()`
- q — 1 x n joint position
- qd — 1 x n joint velocity
- torque — 1 x n torque input

Output Arguments

- qdd — n x 1 joint acceleration

Description

`accel()` computes the forward dynamics, the motion of manipulator in response to the forces/torques applied to its joints.

Examples

```
exec('./models/mdl_puma560.sce',-1);
qdd = accel(p560, q_n, q_z, [1 0 0 0 0 0])
```

angvec2r, angvec2t

tr2angvec

RTSX Category: Mathematics → Rotations

Rotation about an arbitrary vector.

Syntax

- `[R] = angvec2r(theta, k)`
- `[T] = angvec2t(theta, k)`
- `[theta, k] = tr2angvec(R)`
- `[theta, k] = tr2angvec(T)`

Input/Output Arguments

- theta – rotation angle (default is radian. See options below.)
- k – a 3×1 vector representing rotation axis
- R — 3×3 rotation matrix
- T — 4×4 homogeneous transformation matrix

Options

- Put `'deg'` as last argument to input angle value in degree. For example,
  - `T=angvec2t(90,[0 -1 1]‵, ‘deg’)`
  - `[theta, v] = tr2angvec(T,’deg’)`

Description

`angvec2r(), angvec2t()` returns a rotation and homogeneous transformation matrix representing a rotation of angle theta around vector `k`. `tr2angvec()` just does the opposite. It receives a rotation or homogeneous transformation matrix and solves for a set of rotation axis vector and angle.

Examples

```
-->R=angvec2r(pi/4,[1 1 1]')
 R  =
    0.8047379  - 0.3106172    0.5058794
    0.5058794    0.8047379  - 0.3106172
  - 0.3106172    0.5058794    0.8047379

-->[theta,v]=tr2angvec(R)
 v  =
    0.5773503
    0.5773503
    0.5773503
 theta  =
    0.7853982
```

AnimateRobot

AnimateRobotFrame

RTSX Category: Kinematics → Robot Model Creation and Graphics

`AnimateRobot()` shows robot movement corresponding to a sequence of joint variable values. `AnimateRobotFrame()` does the same thing but displays coordinate frames in place of robot joints.

Syntax

- `AnimateRobot(robot, q, options)`
- `AnimateRobotFrame(robot, q, options)`

Input Arguments

- robot –a robot model
- q –a ns x nq matrix of joint variable values, where ns = number of setpoints (move steps) and nq = number of joint variables

Output Arguments

- none

Options

Note : options in [ ] must be passed to the function in pairs.

- 'grid' — show grid
- 'noworld' — do not plot world coordinate frame
- 'notool' — do not plot tool coordinate frame
- ['figure', fnum] — command the plot to a particular window indicated by fnum

Description

This pair of functions is designed to animate movement of a robot model, or DH frames, corresponding to a given sequence of joint variable values, or setpoints, which could be created by a simple script file or from a path generation routine.

Examples

This example demonstrates how to generate a simple command sequence and animate a 2-link manipulator.

```
clear L;
a1 = 1.2; a2 = 1;
L(1)=Link([0 0 a1 0]);
L(2)=Link([0 0 a2 0]);
twolink=SerialLink(L);  // a 2-link manipulator
// generate simple setpoints
// both joints move full circle
t = [0:0.01:1]';        // "time" data
qs = [2*pi*t 2*pi*t];
AnimateRobot(twolink,qs);
```

AppendLink

RemoveLink

ReplaceLink

RTSX Category: Kinematics → Robot Model Creation and Graphics

This set of auxiliary functions is designed to conveniently add, remove, or replace a link in a robot model.

Syntax

- `robot = AppendLink(robot, L, li)`
- `robot = RemoveLink(robot, li)`
- `robot = ReplaceLink(robot, L, li)`

Input Arguments

- robot –a robot model to modify
- L –a link to append/replace
- li –index of robot link

Output Arguments

- robot — updated robot model

Description

Suppose a robot model is already built and one later wants to add/delete/change a link at specific location, this set of support functions is designed for such purpose. When appending/replacing robot with a new link, the link is checked whether it has same DH convention with the robot. If not, the function shows error message and the link is not added/changed.

Examples

```
L(1)=Link([0 0 1.2 0]);
L(2)=Link([0 0 1 0]);
twolink=SerialLink(L); // a 2-link manipulator
newL1 = Link([0 0 0.5 0]);
threelink = AppendLink(twolink,newL1,2);  // insert new link at
location 2
fourlink = AppendLink(threelink, newL1); // if location is
omitted, append on top
threelink1 = RemoveLink(fourlink,3); // remove link 3
newL2 = Link([0 1 0 pi/2],'P');        // new prismatic link
threelink2 = ReplaceLink(threelink1, newL2, 2);  // replace link
2
```

AttachBase, DetachBase

AttachTool, DetachTool

RTSX Category: Kinematics → Robot Model Creation and Graphics

This set of auxiliary functions is designed to conveniently attach/detach base and tool frame in a robot model.

Syntax

- `robot = AttachBase(robot, Tb)`
- `[robot,Tb] = DetachBase(robot)`
- `robot = AttachTool(robot, Tt)`
- `[robot,Tt] = DetachTool(robot)`

Input Arguments

- robot –a robot model to modify
- Tb –base frame, a 4 x 4 homogeneous transformation matrix
- Tt –tool frame, a 4 x 4 homogeneous transformation matrix

Output Arguments

- robot — updated robot model
- Tb – detached base frame, a 4 x 4 homogeneous transformation matrix
- Tt – detached tool frame, a 4 x 4 homogeneous transformation matrix

Description

Suppose a robot model is already built and one later wants to add/remove its base/tool frame, this set of support functions is designed for such purpose. `AttachBase()` and `AttachTool()` checks wheter a user inputs a valid homogeneous transformation matrix, so it is a prefered approach to accessing robot.base or robot.tool directly.

Examples

```
// build a robot first
robot = AttachBase(robot, transl([1 1 0])); // translate robot to
location [1,1,0]'
robot=AttachTool(robot,trotx(pi/2)); // rotate original tool frame
about x 90 degree
robot=DetachBase(robot);
robot=DetachTool(robot);  // back to original
```

CamPlot

RTSX Category: Vision

Syntax

- `uv = CamPlot(cam, P, options)`

Input Arguments

- cam –a camera model
- P –world point coordinates (3 x N)

Output Arguments

- uv — image plane coordinates (2xN) corresponding to the world points P (3xN).

Options

Note : options in [ ] must be passed to the function in pairs.

- ['Tobj', T] — Transform all points by the homogeneous transformation T before projecting them to the camera image plane.
- ['Tcam', T] — Set the camera pose to the homogeneous transformation T before projecting points to the camera image plane. Temporarily overrides the current camera pose .

Examples

```
-->cam = CentralCamera('focal', 0.015, 'pixel', 10e-
6, ...
'resolution', [1280 1024], 'centre', [640 512],
'name', 'mycamera')
--> P = mkgrid(3, 0.2, 'T', transl(0, 0, 1.0));
-->CamPlot(cam,P)
```

CamProject

RTSX Category: Vision

Syntax

- `uv = CamProject(cam, P, options)`

Input Arguments

- cam –a camera model
- P –world point coordinates (3 x N)

Output Arguments

- uv — image plane coordinates (2xN) corresponding to the world points P (3xN).

Options

Note : options in [ ] must be passed to the function in pairs.

- ['Tobj', T] — Transform all points by the homogeneous transformation T before projecting them to the camera image plane.
- ['Tcam', T] — Set the camera pose to the homogeneous transformation T before projecting points to the camera image plane. Temporarily overrides the current camera pose .

Examples

```
-->cam = CentralCamera('focal', 0.015, 'pixel', 10e-6, ...
'resolution', [1280 1024], 'centre', [640 512], 'name',
'mycamera')
--> P = mkgrid(3, 0.2, 'T', transl(0, 0, 1.0));
-->CamProject(cam,P)
 ans  =
    490.    490.    490.    640.    640.    640.    790.
790.    790.
    362.    512.    662.    362.    512.    662.    362.
512.    662.
```

CentralCamera

RTSX Category: Vision

`CentralCamera()` creates a data structure that contains central-projection perspective camera parameters. The camera model assumes central projection; i.e., the focal point is at z=0 and the image plane is at z=f. The image is not inverted.

Syntax

- `cam = CentralCamera(options)`

Output Arguments

- cam — a camera data structure containing the following fields
  - name — camera name

- f — focal length (meters)
- rho — pixel size [W, H]
- npix — image plane resolution [W, H]
- pp — principal point (2 x 1)
- T — camera pose (4 x 4 homogeneous transformation matrix)
- comment — general comment string for this camera

Options

Note : options in [ ] must be passed to the function in pairs.

- 'default' — Default camera parameters: 1024×1024, f=8mm, 10um pixels, camera at origin, optical axis is z-axis, u- and v-axes parallel to x- and y-axes respectively.
- ['name', 'robot name'] — give the camera model some name for reference.
- ['comment', 'comment string'] — a comment for this camera.
- ['focal', f] — camera focal length (meters)
- ['resolution', N] — image plane resolution [W, H]
- ['center', P] — principal point [u,v]
- ['pixel', S] — pixel size: Ss = [W H]
- ['pose', T] — specify a pose for camera, T must be a 4 x 4 homogeneous transformation matrix

Examples

```
cam = CentralCamera('default');
cam = CentralCamera('focal', 0.015);
cam = CentralCamera('focal', 0.015, 'pixel', 10e-6,
...
'resolution', [1280 1024], 'centre', [640 512],
'name', 'mycamera')
```

coriolis

RTSX Category: Dynamics

Compute Coriolis/centripetal matrix in robot dynamics

Syntax

- `C = coriolis(robot, q, qd)`

Input Arguments

- robot — n-link robot data structure created with `SerialLink()`
- q — 1 x n joint position
- q — 1 x n joint velocity

Output Arguments

- C — n x n Coriolis/centripetal matrix

Description

`C = coriolis(robot,q,qd)` computes an n x n Coriolis/centripetal matrix for a robot in joint configuration q and velocity qd, where n is the number of joints. The diagonal elements are due to centripetal effects and the off-diagonal elements are due to Coriolis effects. Joint frictions are eliminated in the computation.

Examples

```
-->exec('./models/mdl_puma560.sce',-1);
-->qd = 0.5*[1 1 1 1 1 1];
-->C = coriolis(p560, q_n, qd)
 C  =
    0.          - 0.9115459    0.2172555    0.0012865  -
0.0025932    0.00006
    0.3140112  - 5.551D-17    0.5786335  - 0.0010762  -
0.0001034  - 0.0000059
  - 0.1803736  - 0.1928778    0.         - 0.0004544  -
0.0023017  - 0.0000059
  - 0.0001528    0.0005860  - 0.0000358    0.
0.0002566  - 0.0000424
    0.           0.0000207    0.0013810  - 0.0002061    0.
- 0.0000059
    0.           0.00002      0.00002      0.0000283
0.0000059    0.
```

cpoly, cpolyplot

qpoly, qpolyplot

lspb, lspbplot

RTSX Category: Path Generation

trajectory generation using cubic, quintic polynomials or linear-segment parabolic blend

Syntax

- `[s, sd, sdd] = cpoly(q0, qf, t, qd0, qdf)`
- `[s, sd, sdd] = qpoly(q0, qf, t, qd0, qdf)`
- `[s, sd, sdd] = lspb(q0, qf, t, v)`

Remark: `cpolyplot(), qpolyplot(), lspbplot()` share the same syntax.

Input Arguments

- q0, qf — start and end position, respectively
- qd0, qdf — start and end velocity, respectively
- t — number of time steps, or time vector
- v — velocity of constant segment in LSPB trajectory

Output Arguments

- s — generated trajectory array
- sd — velocity array
- sdd — acceleration array

Description

`cpoly(), qpoly(), lspb()` generate trajectory from q0 to qf using cubic, quintic polynomial, and linear segment with parabolic blend, respectively. The 3rd argument can be an integer (number of steps) or a time vector. Specifying start and end velocity is optional.

`cpolyplot(), qpolyplot(), lspbplot()` behaves similarly except that the resulting trajectory, velocity, and acceleration are also plotted.

Examples

```
[s, sd, sdd] = cpoly(0, 1, 50);
s = qpoly(0, 1, 50, 0.5, 0);
s = lspbplot(0, 1, 50, 0.035);
```

ctraj

RTSX Category: Path Generation

Compute cartesian trajectory between two points

Syntax

- `Ts = ctraj(T0, T1, t)`

Input Arguments

- T0, T1 — 4 x 4 homogeneous.transform matrix of the start and end frame, respectively
- t — number of time steps, or time vector

Output Arguments

- Ts — 4 x 4 x n cartesian trajectory

Description

`traj()` computes cartesian trajectory between two coordinate frames T0 and T1. The n points trajectory follows a trapezoidal velocity profile (LSPB) along the path. The resulting cartesian trajectory is a 3-dimensional homogeneous transform matrix with the 3rd dimension being the point index; i.e., `Ts(:,:,i)` is the i'th point along the path. Argument t can be given as number of time steps n or a time vector.

Examples

```
T0 = transl([0.4, 0.2, 0])*trotx(pi);
T1 = transl([-0.4, -0.2, 0.3])*troty(pi/2)*trotz(-pi/2);
Ts = ctraj(T0, T1, 50);
tranimate(Ts);
```

DeleteFrame

InsertFrame

ReplaceFrame

RTSX Category: Mathematics → Homogeneous Transformations

Insert,delete, or replace a frame in a frame structure data created by `Frame()`.

Syntax

- `F = InsertFrame(F, Fi, findex)`
- `F = DeleteFrame(F, findex)`
- `F = ReplaceFrame(F, Fi, findex)`

Input Arguments

- F – a frame structure created by `Frame()`
- Fi – a single frame created by `Frame()`
- findex – frame index; i.e., the frame number to insert, delete, or replace

Output Arguments

- F — modified frame data structure

Description

This group of functions is handy when a frame structure is already created by `Frame()` and later found that it needs some minor modification, such as adjusting only a single frame. Use `InsertFrame()`, `DeleteFrame()`, `ReplaceFrame()` to insert, remove, and replace a frame at a location specified by findex. In the case of insertion and replacement, a new frame must first be created using `Frame()`.
If you buit a chain with original frame structure before, you need to rebuild with `SerialFrame()` using the modified frame structure as input argument.

Examples

Try these commands to see what happens to the frame structure.

```
// create a sequence of frames for testing
F(1)=Frame(eye(4,4),'name','U');
F(2)=Frame(trotx(pi/2)*transl([0 1 1]),'name','V');
F(3)=Frame(transl([2 3 -1]),'abs','name','W');
 // create a single frame for insertion/replacement
Fi=Frame(trotz(pi/4)*trans([-1 1 -1]),'name','X');
F = InsertFrame(F,Fi,2);       // insert at location 2.
Now F has 4 frames
F=DeleteFrame(F,4);     // delete frame 4. F is reduced
to 3 frames
F=ReplaceFrame(F,Frame(trotz(pi/4)),3); // replace
frame 3 with another new frame
F=InsertFrame(F,Frame([],'name','?'),3); // insert a
blank frame at location 3
// after finish. Run SerialFrame( ) to create a chain
fc1 = SerialFrame(F,'name','Modified chain 1');
```

delta2tr

tr2delta

RTSX Category: Kinematics → Velocity Kinematics

Convert between differential motion and homogeneous transform

Syntax

- `T = delta2tr(d)`
- `d= tr2delta(T)`
- `d= tr2delta(T0, T1)`

Input/Output Arguments

- d — 6 x 1 differential motion
- T, T0, T1 — 4×4 homogeneous transformation matrix

Description

`tr2delta( )` converts homogeneous transform to differential motion. `d = tr2delta(T0, T1)` is the differential motion (6×1) corresponding to infinitessimal motion from pose T0 to T1 which are homogeneous transformations. `d=(dx, dy, dz, dRx, dRy, dRz)` and is an approximation to the average spatial velocity multiplied by time.

`d = tr2delta(T)` is the differential motion corresponding to the infinitessimal relative pose T expressed as a homogeneous transformation. d is only an approximation to the motion T, and assumes that T0 ~ T1 or T ~ eye(4,4).

`delta2tr( )` converts differential motion to a homogeneous transform. `T = delta2tr(d)` is a homogeneous transform representing differential translation and rotation.

DetachBase

DetachTool

RTSX Category: Kinematics → Robot Model Creation and Graphics

See `AttachBase`.

eul2jac

rpy2jac

tr2jac

RTSX Category: Kinematics → Velocity Kinematics

Compute Jacobian from homogeneous transform, euler, or RPY angles

Syntax

- `J = tr2jac(T)`
- `J = eul2jac(eul)`
- `J = rpy2jac(rpy)`

Input Arguments

- T — 4×4 homogeneous transformation matrix
- eul — 1 x 3 vector of euler angles (in radian)
- rpy — 1 x 3 vector of Roll-Pitch-Yaw angles (in radian)

Output Arguments

- J — 3 x 1 Jacobian matrix

Description

- `J = tr2jac(T)` is a Jacobian matrix (6×6) that maps spatial velocity or differential motion from the world frame to the frame represented by the homogeneous transform T.
- `J = eul2jac(eul)` is a Jacobian matrix (3×3) that maps Euler angle rates to angular velocity at the operating point eul =[phi, theta, psi].
- `J = rpy2jac(rpy)` is a Jacobian matrix (3×3) that maps roll-pitch-yaw angle rates to angular velocity at the operating point rpy=[R,P,Y].

Examples

```
-->T = transl([1 0 0])*troty(pi/2);
-->J = tr2jac(T)
 J  =
    0.    0.  - 1.    0.    1.    0.
    0.    1.    0.    0.    0.    1.
    1.    0.    0.    0.    0.    0.
    0.    0.    0.    0.    0.  - 1.
    0.    0.    0.    0.    1.    0.
    0.    0.    0.    1.    0.    0.

-->J = eul2jac([pi/3 -pi/4 pi/6])
 J  =
  - 0.6123724  - 0.5          0.
  - 0.3535534    0.8660254    0.
    0.7071068    0.           1.

 -->J = rpy2jac([0.1 0.2 0.3])
 J  =
    1.    0.           0.1986693
    0.    0.9950042  - 0.0978434
    0.    0.0998334    0.9751703
```

eul2r, eul2t

tr2eul

RTSX Category: Mathematics → 3D Position/Orientation →Rotations

Convert between euler angle (ZYZ sequence) and rotation/homogeneous transformation matrix.

Syntax

- `[R] = eul2r(eul)`
- `[T] = eul2t(eul)`
- `[eul] = tr2eul(R)`
- `[eul]= tr2eul(T)`

Input/Output Arguments

- eul – 1×3 vector representing euler angle (default in radian)
- R — 3×3 rotation matrix
- T — 4×4 homogeneous transformation matrix

Options

- Put `'deg'` as last argument to input angle value in degree. For example,
  - `T=eul2t([30 45 60], 'deg')`
  - `eul = tr2eul(T, 'deg')`

Description

`eul2r()`, `eul2t()` returns a rotation and homogeneous transformation matrix representing euler rotation of [phi theta psi] (ZYZ sequence is used.) `tr2eul()` just does the opposite. It receives a rotation or homogeneous transformation matrix and solves for a set of euler angle.

Examples

```
-->R=eul2r([30 45 90],'deg')
 R  =
  - 0.5         - 0.6123724    0.6123724
    0.8660254   - 0.3535534    0.3535534
    0.            0.7071068    0.7071068

-->tr2eul(R,'deg')
 ans  =
    30.     45.     90.
```

FKine

RTSX Category: Kinematics → Forward Kinematics

From a robot model and a vector of joint variable values, compute homogeneous transformation matrices from tool frame to base.

Syntax

- `T=FKine(robot,q)`

Input Arguments

- robot –a robot model
- q –a 1 x nq vector of joint variable values, where nq = number of joint variables

Output Arguments

- T — a 4×4 homogeneous transformation matrix representing tool frame w.r.t base

Description

A normal forward kinematics solution refers to a homogeneous transformation from the tool frame to base frame. `FKine()` simply calls `Link2AT()`, extracts the transformation from top to bottom link, and then adjust for the base and tool frames of the robot. The return matrix is just a 4 x 4 homogenous transformation matrix from tool to base. Nothing else matters.

Examples

Goal: find forward kinematics from tool to base of an RRR robot.

```
-->clear L;
 -->d1 = 1; a2 = 1; a3 = 1;
 -->L(1)= Link([0 d1 0 pi/2]);
 -->L(2)=Link([0 0 a2 0]);
 -->L(3)=Link([0 0 a3 0]);
 -->RRR_robot=SerialLink(L);
 -->q0 = [0 0 0];
 -->T=FKine(RRR_robot,q0)
 T  =
    1.     0.     0.     2.
    0.     0.   - 1.     0.
    0.     1.     0.     1.
    0.     0.     0.     1.
```

Frame

RTSX Category: Mathematics → Homogeneous Transformation

Create coordinate frame structure. Useful as a first step to build a chain of homogeneous transformations.

Syntax

- `F = Frame(T, options)`

Input Arguments

- T – a 4 x 4 homogeneous transformation matrix

Output Arguments

- F — frame data structure containing the following fields
  - Tabs — absolute homogeneous transformation matrix (w.r.t first frame in the chain)
  - Trel — relative homogeneous transformation matrix (w.r.t adjacent frame below)
  - name — a string specifying frame name
  - completed — a boolean value (0 or 1) to flag the completion of frame (used bySerialFrame( ) when forming a chain.)

Options

Note : options in [ ] must be passed to the function in pairs.

- 'rel' – input T is a relative homogeneous transformation (default)
- 'abs' — input T is an absolute homogeneous transformation
- ['name', '(frame name)'] — frame name. A single character such as 'A', 'B', '1', '2' is recommended. A long name would look messy during plotting.

Description

`Frame()` has little use by its own. one does not need it when working with only 1-2 frames. In contrast, if a chain of several frames needs to be constructed, `Frame()` combined with `SerialFrame()` and related functionsare a handy set of tools for such purpose. For those familiar with OOP, a frame data structure can be thought of roughly as an object (lacking method, of course). More information other than just homogenous transformation matrices could be attached to the structure. `Frame()` is used as the required first step to build a frame chain.

Examples

```
F=Frame(trotz(pi/4),'name','A');  // create just a single frame
// create a sequence of frames
F(1)=Frame(eye(4,4),'name','U');   // base frame
F(2)=Frame(trotx(pi/2)*transl([0 1 1]),'name','V');
// 2nd frame (default T input is relative)
F(3)=Frame(transl([2 3 -1]),'abs','name','W');
// 3rd frame is inputted as absolute T w.r.t base
F(4)=Frame([],'name','X');
// a blank frame can be inserted at only one place in chain
F(5)=Frame(troty(pi/3)*transl([3 2 -1]),'name','Y');
// last frame in structure
```

Notice in the example above that frame 4 is defined as a blank frame. This can be solved by function `ResolveFrame()` provided that another chain is available to form a closed chain, a typical problem in basic robot kinematics

gravload

RTSX Category: Dynamics

Compute gravity load in robot dynamics

Syntax

- `Tg = gravload(robot, q, grav)`

Input Arguments

- robot — n-link robot data structure created with `SerialLink()`
- q — 1 x n joint position
- grav — gravity acceleration vector (optional)

Output Arguments

- Tg — 1 x n joint torque from gravity loading

Description

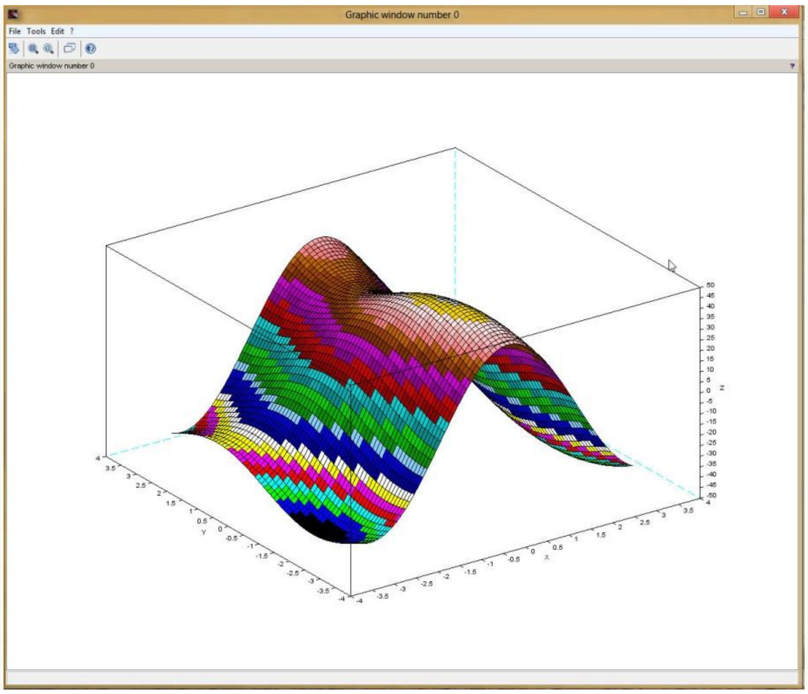
`Tg = gravload(robot, q, grav)` is the joint gravity loading for the robot in the joint configuration q. If gravity acceleration vector argument is omitted, the function uses the value given in robot data structure.

Examples

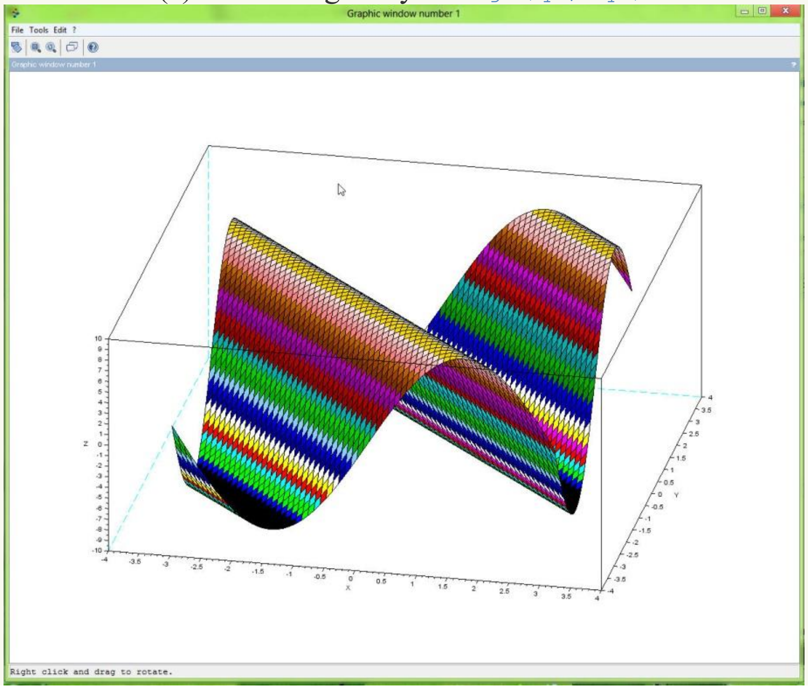
```
-->exec('./models/mdl_puma560.sce',-1);
 -->gravload(p560,q_n)
 ans  =
    0.    31.63988    6.035138    0.    0.0282528    0.
```

The following code plots the figures below. You may have to wait quite a while!

```
stepsize = 0.1;
[Q2,Q3] = meshgrid(-pi:stepsize:pi, -pi:stepsize:pi);
nloops = numcols(Q2)*numcols(Q3);
g2 = _zeros(size(Q2));
g3 = _zeros(size(Q3));
 for i=1:numcols(Q2),
     for j=1:numcols(Q3),
        g = gravload(p560,[0 Q2(i,j) Q3(i,j) 0 0 0]);
         g2(i,j) = g(2);
        g3(i,j) = g(3);
    end
 end
 figure; surf(Q2, Q3, g2);
 figure; surf(Q2, Q3, g3);
```



(a) Shoulder gravity load `g2(q2, q3)`



(b) Elbow gravity load `g3(q2, q3)`

Gravity load variation with manipulator pose

IBVS4

RTSX Category: Vision

Syntax

- `[pmat,pt,vmat,emat,Tcmat,jcond,zvec]=IBVS4(cam, options)`

Input Arguments

- cam –a camera model

Output Arguments

- pmat, pt — point trajectory data
- vmat — velocity data
- emat — point error data
- Tcmat — camera pose data
- jcond — Jacobian condition number data
- zvec — depth estimation data

Options

Note : options in [ ] must be passed to the function in pairs.

- ['niter', N] — Maximum number of iterations.
- ['eterm', E] — Terminate when norm of feature error less than E
- ['lambda', L] — Control gain, positve scalar
- ['T0', T] — Initial pose
- ['P', p] — Set of world points (3 x N)
- ['pstar', P] — The desired image plane coordinates
- ['depth', D] — set dept of points to D
- ‘depthest’ — run a simple depth estimator

Description

Use IBVS4( ) shows how to implement IBVS (Image-Based Visual Servoing) for 4 feature points. Results are plotted and given as outputs.

Examples

```
-->cam = CentralCamera('default');
-->P = mkgrid(2, 0.5,'T',transl(0,0,2));
-->pStar = [ 312   312   712   712;
   312   712   712   312];
-->Tc0 = transl(1,1,-3)*trotz(0.6);
-->ibvs4(cam,'T0',Tc0, 'P', P,'pstar',pStar,'depth',1);
-->ibvs4(cam,'T0',Tc0, 'P', P,'pstar',pStar,'niter',100,'depthest');
// next commands show how rotation about camera Z axis causes camera
retraction problem
-->pStar = camproject(cam,P,'Tcam',trotz(0.9*pi));
-->ibvs4(cam, 'P', P,'pstar',pStar,'depth',2);
```

IKine

RTSX Category: Kinematics →Inverse Kinematics

This function computes inverse kinemetics for a general robot model numerically

Syntax

- `q = Ikine(robot, T, options)`

Input Arguments

- robot –a robot model
- T — a 4×4 (or 4x4xns) homogeneous transformation matrix representing tool frame w.r.t base

Output Arguments

- q –a 1 x nq (or ns x nq) vector (matrix) of joint variable values, where nq = number of joint variables, ns = number of sequences.

Options

Note : options in [ ] must be passed to the function in pairs.

- 'pinv' — use pseudo-inverse instead of Jacobian transpose
- 'novarstep' — disable variable step size

- 'verbose' — show number of iterations for each point
- 'verbose=2' — show state at each iteration
- 'plot' — plot iteration states versus time (cannot be used when T is a 3D matrix.)
- ['q0', [1 x nq vector]] — initial estimate of joint variable values
- ['m', [1 x 6 mask vector]] –mask vector
- ['ilimit', value] –set the maximum iteration count (default 1000)
- ['tol', value] –set the tolerance on error norm (default 1e-6)
- ['alpha', value] –set step size gain (default 1)

Description

`IKine()` iterates an inverse kinematic solution for a general robot model that does not have a closed-form solution. The execution is quite slow and may diverge for bad choice of parameters and initial joint variable ettimates. Solution is sensitive to choice of initial gain. The variable step size logic (enabled by default) does its best to find a balance between speed of convergence and divergence. Some experimentation might be required to find the right values of tol, ilimit and alpha.The pinv option sometimes leads to much faster convergence. The tolerance is computed on the norm of the error between current and desired tool pose. This norm is computed from distances and angles without any kind of weighting. The inverse kinematic solution is generally not unique, and depends on the initial guess Q0 (defaults to 0). Joint offsets, if defined, are added to the inverse kinematics to generate Q.
For the case where the manipulator has fewer than 6 DOF the solution space has more dimensions than can be spanned by the manipulator joint coordinates. In this case the mask vector m specifies the Cartesian DOF (in the wrist coordinate frame) that will be ignored in reaching a solution. The mask vector has six elements that correspond to translation in X, Y and Z, and rotation about X, Y and Z respectively. The value should be 0 (for ignore) or 1. The number of non-zero elements should equal the number of manipulator DOF. For example when using a 5 DOF manipulator rotation about the wrist z-axis might be unimportant in which case M = [1 1 1 1 1 0].

Examples

```
-->exec('./models/mdl_puma560.sce',-1);   // create robot name p560
// with joint variable vectors q_n
-->q_n
 q_n  =
    0.     0.7853982    3.1415927    0.     0.7853982    0.
// Compute forward kinematics from q_n
-->T=fkine(p560, q_n)
 T  =
    0.     0.     1.     0.5963031
    0.     1.     0.   - 0.15005
  - 1.     0.     0.   - 0.0143543
    0.     0.     0.     1.
// Compute inverse kinematics
--> q_i = ikine(p560, T)
.............................................................................
.............................................................................
.............................................................................
.........................
 q_i  =
- 0.0000009  - 0.8335316    0.0939530    0.0000014  - 0.8312176  - 0.0000010
// forcing a solution with initial joint variable estimates
-->q_i = ikine(p560, T, 'q0', [0  0  3  0  0  0])
.............................................................................
.............................................................................
.............................................................................
...........
q_i  =
- 0.0000009    0.7853971    3.1415954  - 0.0000016    0.7853970    0.0000012
```

IKine6s

RTSX Category: Kinematics →Inverse Kinematics

This function computes inverse kinematics for a robot model with 6 revolute joints and having a spherical wrist, such as Puma 560.

Syntax

- q = Ikine6s(robot, T, options)

Input Arguments

- robot –a robot model
- T — a 4×4 (or 4x4xns) homogeneous transformation matrix representing tool frame w.r.t base

Output Arguments

- q –a 1 x nq (or ns x nq) vector (matrix) of joint variable values, where nq = number of joint variables, ns = number of sequences.

Options

Note : Configuration codes below can be put together in one string

- 'l' — arm to the left (default)
- 'r' — arm to the left
- 'u' — elbow up (default)
- 'd' — elbow down
- 'n' — wrist not flipped (default)
- 'f' — wrist flipped (rotated by 180 deg)

Description

`IKine6s()` computes joint coordinates corresponding to the robot end-effector pose T represented by the homogenenous transform. This is a analytic solution for a 6-axis robot with a spherical wrist (such as the Puma 560). The inverse kinematic solution is generally not unique, and depends on the configuration string. Joint offsets, if defined, are added to the inverse kinematics to generate q.

Examples

```
-->exec('./models/mdl_puma560.sce',-1);  // create robot name
p560
// with joint variable vectors q_n
-->q_n
 q_n  =
    0.    0.7853982    3.1415927    0.    0.7853982    0.
// Compute forward kinematics from q_n
-->T=fkine(p560, q_n)
```

```
  T   =
     0.     0.     1.     0.5963031
     0.     1.     0.   - 0.15005
   - 1.     0.     0.   - 0.0143543
     0.     0.     0.     1.
// Compute inverse kinematics
-->q_i = ikine6s(p560,T)
 q_i   =
     2.6485612  - 3.9269908    0.0939558    2.5325594
0.9743496    0.3733996
 // even q_i not equal q_n, they render the same tool frame
-->fkine(p560, q_i)
 ans   =
     0.     0.     1.     0.5963031
     0.     1.     0.   - 0.15005
   - 1.     0.     0.   - 0.0143543
     0.     0.     0.     1.
 // force a solution with configuration code
-->q_i = ikine6s(p560,T,'ru')
 q_i   =
     0.     0.7853982    3.1415927    0.    0.7853982    0.
```

inertia

RTSX Category: Dynamics

Compute inertia matrix in robot dynamics

Syntax

- `M = inertia(robot, q)`

Input Arguments

- robot — n-link robot data structure created with `SerialLink()`
- q — 1 x n joint position

Output Arguments

- M — n x n joint inertia matrix

Description

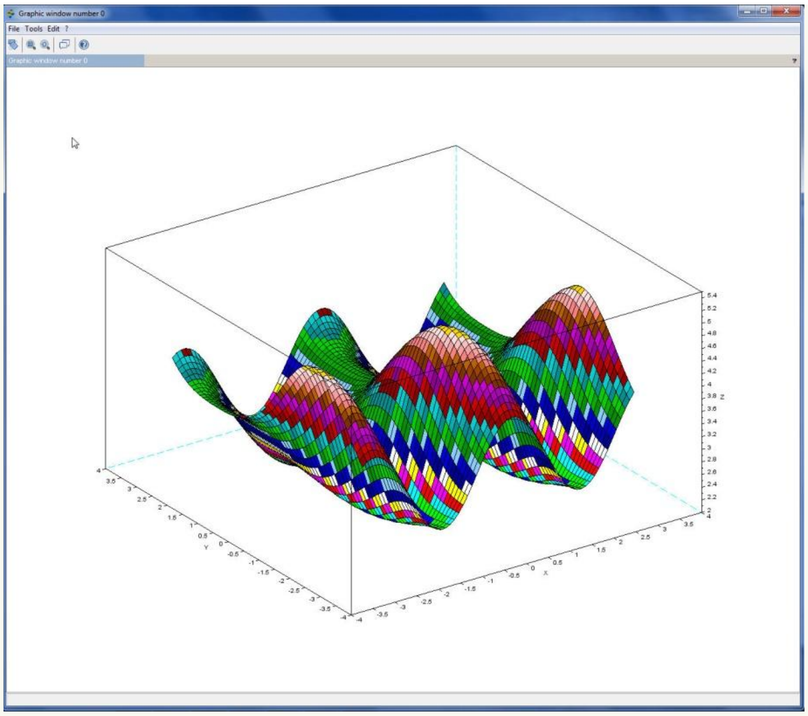
`M = inertia(robot, q)` computes an n x n symmatric joint inertia matrix relating joint torque to joint acceleration of the robot at joint configuration q. The diagonal elements `M(i,i)` are inertias seen by joint actuator i, including motor inertias reflected through the gear ratios. The off-diagonal element `M(i,j)` are coupling inertias relating acceleration on joint i to force/torque on joint j.

Examples

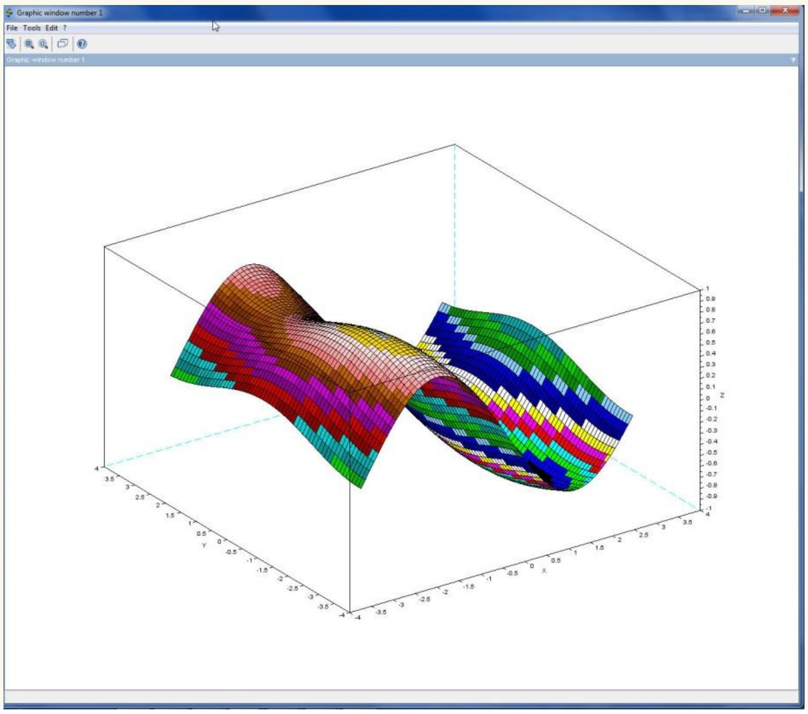
```
-->exec('./models/mdl_puma560.sce',-1);
-->M = inertia(p560, q_n)
 M  =
    3.6593754  - 0.4043612    0.1006136  - 0.0025170    0.
0.
  - 0.4043612    4.4137419    0.3508907    0.
0.0023595    0.
    0.1006136    0.3508907    0.9378416    0.
0.0014802    0.
  - 0.0025170    0.           0.           0.1925317    0.
0.0000283
    0.           0.0023595    0.0014802    0.
0.1713485    0.
    0.           0.           0.           0.0000283    0.
0.1941045
```

The following code plots the figures below. You may have to wait quite a while!

```
stepsize = 0.1;
[Q2,Q3] = meshgrid(-pi:stepsize:pi, -pi:stepsize:pi);
nloops = numcols(Q2)*numcols(Q3);
g2 = _zeros(size(Q2));
g3 = _zeros(size(Q3));
 for i=1:numcols(Q2),
     for j=1:numcols(Q3),
        M = inertia(p560,[0 Q2(i,j) Q3(i,j) 0 0 0]);
         M11(i,j) = M(1,1);
        M12(i,j) = M(1,2);
    end
 end
 figure; surf(Q2, Q3, M11);
 figure; surf(Q2, Q3, M12);
```



(a) Joint 1 inertia as a function of joint 2 and 3 angles `M11(q2, q3)`



(b) Product of inertia `M12(q2,q3)`

Variation of inertia matrix elements as a function of manipulator pose

InsertFrame

RTSX Category: Mathematics → Homogeneous Transformations

See `DeleteFrame`.

isquaternion

isqequal

RTSX Category: Mathematics → Quaternion

Test whether a variable is a quaternion, or whether two quaternions are equal.

Syntax

- `h = isquaternion(q)`
- `h = isqequal(q1,q2)`

Input Arguments

- q, q1, q2 — a quaternion s <v1,v2,v3>

Output Arguments

- h — boolean logic (true or false)

Description

`isquaternion(q)` returns true if q is a quaternion, false otherwise. `isqequal(q1, q2)` returns true if q1 and q2 are quaternions and q1 == q2, false otherwise. Both functions accept arrays of quaternions and return an array of the same size, whose elements are boolean logic corresponding to whether the test conditions are met.

Examples

```
-->T1 = troty(pi/4);
 -->q1 = quaternion(T1);
 -->isquaternion(q1)
 ans  =
  T
 -->R = roty(pi/4);
 -->q2 = quaternion(R);
 -->isqequal(q1,q2)
 ans  =
  T
```

jacob0

jacobn

RTSX Category: Kinematics → Velocity Kinematics

Compute Jacobian that maps joint velocity to base or end-effector spatial velocity

Syntax

- `J0 = jacob0(robot, q, options)`
- `Jn = jacobn(robot, q, options)`

Input Arguments

- robot –a robot model to modify
- q –a 1 x nq vector of joint variable values, where nq = number of joint variables

Output Arguments

- J0, Jn — 6 x nq Jacobian matrix

Options

- 'rpy' — compute analytical Jacobian with rotation rate in terms of roll-pitch-yaw angles

- 'eul' — compute analytical Jacobian with rotation rate in terms of Euler angles
- 'trans' — return translational submatrix of Jacobian
- 'rot' — return rotational submatrix of Jacobian

Description

`J0 = jacob0(robot, q, options)` is the Jacobian matrix (6 x nq) for the robot in pose q (1 x nq). The manipulator Jacobian matrix maps joint velocity to end-effector spatial velocity V = J0*QD expressed in the world-coordinate frame.

`Jn = jacobn(robot, q, options)` is the Jacobian matrix (6 x nq) for the robot in pose q (1 x nq). The manipulator Jacobian matrix maps joint velocity to end-effector spatial velocity V = JN*QD in the end-effector frame.

Examples

```
-->exec('./models/mdl_puma560.sce',-1);    // create p560
robot model
 -->J0 = jacob0(p560,q_n)
 J0  =
    0.15005       0.0143543    0.3196830    0.           0.
0.
    0.5963031    0.           0.           0.           0.
0.
    0.           0.5963031    0.2909744    0.           0.
0.
    0.           0.           0.           0.7071068    0.
1.
    0.         - 1.         - 1.           0.         - 1.
0.
    1.           0.           0.         - 0.7071068    0.
0.

-->Jn = jacobn(p560,q_n)
 Jn  =
    0.         - 0.5963031  - 0.2909744    0.           0.
0.
    0.5963031    0.           0.           0.           0.
0.
    0.15005       0.0143543    0.3196830    0.           0.
0.
  - 1.           0.           0.           0.7071068    0.
0.
    0.         - 1.         - 1.           0.         - 1.
0.
    0.           0.           0.           0.7071068    0.
1.
```

jtraj

RTSX Category: Path Generation

Compute a joint space trajectory between two points

Syntax

- `[q, qd, qdd] = jtraj(q0, qf, t, qd0, qdf)`

Input Arguments

- q0, qf — 1 x n row vectors of start and end positions, respectively.
- t — number of time steps, or time vector
- qd0, qdf — start and end velocity, respectively

Output Arguments

- q — generated trajectory matrix (m x n)
- qd — velocity matrix (m x n)
- qdd — acceleration matrix (m x n)

Description

`jtraj()` computes a joint space trajectory between two set of joint variable values q0 (1 x n) and qf (1 x n). A qunitic polynomial is used with default zero boundary conditions for velocity and acceleration. Argument t can be given as number of time steps or a time vector. Initial and final joint velocity can be specified via `qd0` and `qdf`, respectively.

Examples

```
q = jtraj([0 0],[pi/2 pi/4], 50);
plot(q);     // see the resulting trajectory
[q, qd] = jtraj([-1 0 0 ],[1 pi/4 pi],[1:49]/49, [1 1
1],[2 0 1]);
figure(1);
subplot(211), plot(q); title('joint trajectory')
subplot(212), plot(qd); title('joint velocity')
```

Link

RTSX Category: Kinematics → Robot Model Creation and Graphics

A Link data structure holds all information related to a robot link, such as DH parameters, rigid-body inertial parameters, motor and transmission parameters.

Syntax

- `L = Link(lparam, jtype,options)`

Input Arguments

- lparam – a 1 x 4 or 1 x 5 vector consisting of DH parmeters and offset in this order [theta d a alpha (offset)], where link offset is optional.
- jtype — joint type 'R' = revolute (default), 'P' = prismatic

Output Arguments

- L — link data structure containing the following fields
  - sigma, (RP) — joint type: 0 ('R') = revolute, 1 ('P') = prismatic
  - theta — kinematic: joint angle
  - d — kinematic: link offset
  - a — kinematic: link length
  - alpha — kinematic: link twist angle
  - offset — kinematic: joint variable offset
  - mdh — kinematic: 0 = standard DH, 1 = modified DH
  - qlim — kinematic: joint variable limits [min max]
  - m — dynamic: link mass
  - r — dynamic: link center of gravity w.r.t link coordinate frame ,1 x 3 vector
  - I — dynamic: link inertia matrix about link center of gravity, 1 x 6 vector
  - B — dynamic: link viscous friction
  - Tc — dynamic: link Coulomb friction
  - G — actuator: gear ratio
  - Jm — actuator: motor inertia

Options

Note : options in [ ] must be passed to the function in pairs.

- ['dhtype', (choice)] — type of DH parameters. Pass 0 (or ‘sdh’) for standard DH, and 1 (or ‘mdh’) for modified DH.
- ['qlim', [qmin qmax]] — joint variable limits. Pass minimum and maximum values in a 1×2 vector as shown.
- ['dynparm', [1 x 15 dynamic parameters]] — complete dynamic parameter assignment with a 1 x 15 vector in this order

  [r(1x3), I(1x6), m, Jm, G, B, Tc(1x2)] .
- ['r', [values]] — link center of gravity, 1 x 3 vector.
- ['i', [values]] — link inertia matrix, 1 x 6 vector.
- ['m', value] — link mass
- ['jm', value] — motor inertia
- ['g', value] — motor gear ratio
- ['b', [values]] — link coulomb friction, 1 x 2 vector

Description

A physical robot can be thought of as a mechanical structure built from links and joints connected in series. This concept can be brought into software model by creating a data structure containing link and joint information. Link( ) is the first step in such modeling process. We use the function to construct a link data structure with kinematic, dynamic, and some other essential parameters, then pass the link structure to SerialLink( ) to form a robot model.
The only required arguments for Link( ) are the 4 DH parameters. A link with revolute joint is returned by default, unless prismatic joint type ‘P’ is specified. Other parameters can be passed at creation or filled in later by UpdateRobotLink( ).

Examples

```
L = Link([0 1 1 pi/2]);  // create a single revolute link, d=1, a=1,
// alpha=pi/2  and theta is joint variable

// create a structure of links
L(1) = Link([0 0 1 0]);
L(2) = Link([pi/2 1 0 0],'P');   // link 2 is prismatic d=variable
```

```
with initial value 1
// Note: if one wants to assign other options such as dynamic
parameters, joint type
// must be specified for both R and P types. Cannot leave as default.
L(3) = Link([0 0 2 pi],'R','m',0.43,'r',[0, 0.018, 0],'qlim',[-pi/2
pi/2]);
```

Link2AT

RTSX Category: Kinematics → Forward Kinematics

From link data structure, compute homogeneous transformation matrices from each frame to base.

Syntax

- `[A,T,Ti] = Link2AT(L,q)`

Input Arguments

- L –a link data strucure created by Link( )consisting of link and joint parameters ordered from bottom to top.
- q –a 1 x nq vector of joint variable values, where nq = number of joint variables

Output Arguments

- A — a 4 x 4 x nq homogeneous transformation matrix representing each frame w.r.t. frame below, where nq = number of joints
- T — a 4×4 homogeneous transformation matrix representing uppermost frame w.r.t base
- Ti — a 4x4xnq homogeneous transformation matrix representing each frame w.r.t base

Description

`Link2AT()` computes homogeneous transformation matrices that describe each joint coordinate frame completely; that is, each frame w.r.t adjacent frame below and each frame w.r.t bottom frame. The resulting transformations are packed to two set of 3-dimensional matrices A and Ti. T is the forward kinematics solution. i.e., homogeneous transformation of frame {n} w.r.t {0}. This notation is quite common in robotics textbooks. For example, the transformation from frame (n) w.r.t. (n-1) is labeled An, and T {n} w.r.t {0} = A1*A2* … *An. See, for example, [SHV06].

Examples

The following example shows how to compute A and T matrices from an RRR robot.

```
clear L;
d1 = 1; a2 = 1; a3 = 1;
L(1)= Link([0 d1 0 pi/2]);
L(2)=Link([0 0 a2 0]);
L(3)=Link([0 0 a3 0]);
q0 = [0 0 0];
[A,T,Ti]=Link2AT(L,q);
```

LSPB

RTSX Category: Path Generation

See `cpoly`.

maniplty

RTSX Category: Kinematics → Velocity Kinematics

Compute manipulability for a robot.

Syntax

- `M = maniplty(robot, q, options)`

Input Arguments

- robot –a robot model to modify
- q –a 1 x nq vector of joint variable values, where nq = number of joint variables

Output Arguments

- M — the manipulability index measure

Options

- 'T' — manipulability for translational motion only
- 'R' — manipulability for rotational motion only
- 'yoshikawa' — use Yoshikawa algorithm (default)
- 'asada' — use Asada algorithm

Description

`M = maniplty(robot, q, options)` is the manipulability index measure for the robot at the joint configuration Q. It indicates dexterity, that is, how isotropic the robot's motion is with respect to the 6 degrees of Cartesian motion. The measure is high when the manipulator is capable of equal motion in all directions and low when the manipulator is close to a singularity.

Examples

```
See chapter 5 and 6.
```

mtraj

mstraj

RTSX Category: Path Generation

Multi-axis (mtraj) and multi-segment/multi-axis (mstraj) trajectory generation

Syntax

- `[s, sd, sdd] = mtraj(q0, qf, M, options)`
- `s = mstraj(segments, qdmax, tsegment, q0, dt, Tacc, options)`

Input Arguments

- q0, qf — row vectors of start and end positions, respectively. Number of elements in each vector equal number of motion axes.
- M — number of points
- segments — m x n matrix of via points, one row per point, one column per axis. The last via point is the destination.
- qdmax — 1 x n vector of axis velocity limits that cannot be excedded
- tsegment — m x 1 vector containing the durations for each of the m segments
- q0 — 1 x n vector of initial axis coordinates
- dt — time step
- tacc — 1 x m vector of acceleration time for each segment

Output Arguments

- s — generated trajectory matrix
- sd — velocity matrix
- sdd — acceleration matrix

Options

Note : options in [ ] must be passed to the function in pairs.

`mtraj`

- 'cpoly' — use cubic polynomial trajectory
- 'qpoly' — use quintic polynomial trajectory
- 'lspb' — use linear segment with parabolic blend trajectory
- 'plot' — plot trajectory

`mstraj`

- 'verbose' — print information in command window
- 'plot' — plot trajectory
- ['qd0', value] — specify 1 x n vector of initial axis velocity
- ['qdf', value] — specify 1 x n vector of final axis velocity

Description

`mtraj()` generates multi-axis trajectory from q0 to qf using cubic, quintic polynomial, or linear segment with parabolic blend (default), specified as an option. `mstraj()` generates multi-segment/multi-axis trajectory based on via points and axis velocity limits. Linear segment with parabolic blend is used. Initial and final axis velocities can be specified as options.

Examples

```
s = mtraj([0 2], [1 -1], 50, 'qpoly');
s = mtraj([-1 1], [2 3], 100, 'plot');
via = [4, 1; 4, 4; 5, 2; 2, 5];
q = mstraj(via, [2 1],[], [4, 1], 0.05, 1, 'plot');
```

payload

RTSX Category: Dynamics

Add payload mass to end-effector of robot

Syntax

- `robot = payload(robot, m, p)`

Input/output Arguments

- robot — n-link robot data structure created with SerialLink( )
- m — point mass
- p — position to add payload

Description

robot = payload(robot, m, p) adds a payload with point mass m at position p in the end-effector coordinate frame.

Examples

```
-->exec('./models/mdl_puma560.sce',-1);
-->p560 = payload(p560, 2.5, [0 0 0.1]);
```

PlotFrame

RTSX Category: Mathematics → Homogeneous Transformations

`PlotFrame()` provides a more systematic way to visualize a sequence of homogeneous transformations than `trplot()`. A tranformation chain must be passed to the function instead of a homogeneous transformation matrix.

Syntax

- `PlotFrame(fc, options)`

Input Arguments

- fc – a chain of completed coordinate frames created by SerialFrame( )

Output Arguments

- none

Options

Note : options in [ ] must be passed to the function in pairs.

- 'grid' — show grid
- 'oinfo' — show origin location (could look messy when several frames are condensed together)
- ['figure', fnum] — command the plot to a particular window indicated by fnum

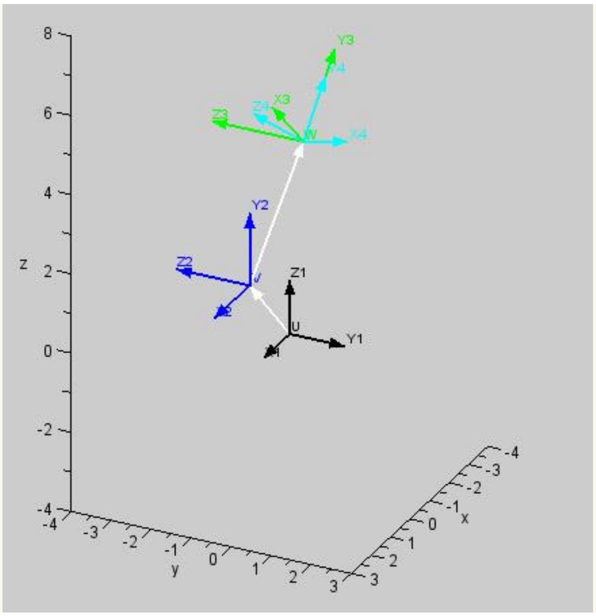
Description

To study robot kinematics, one must understand clearly the basic of kinematic chain connecting coordinate frames together. `PlotFrame()` serves as a convenient graphical tool to visualize such a chain. The function accepts a completed chain created by `SerialFrame()` . An incomplete chain must be resolved first by `ResolveFrame()`, if possible. Unlike `trplot()` where a user has control over a frame's axis color and line style, `PlotFrame()` assigns colors to frames automatically depending on their positions in the chain.

Examples

The example below shows the necessary steps from frame creation to plotting. The result is shown below.

```
// first create a sequence of frames
clear F;
F(1)=Frame(eye(4,4),'name','U');
F(2)=Frame(trotx(pi/2)*transl([0 1 1]),'name','V');
F(3)=Frame(trotz(pi/4)*transl([2 3 -1]),'name','W');
F(4)=Frame(troty(pi/3),'name','Y');
fc=SerialFrame(F,'name','Chain 1');
PlotFrame(fc);
```



transformation chain plot example with `PlotFrame()`

PlotResolveFrame

ResolveFrame

RTSX Category: Mathematics → Homogeneous Transformations

For a chain with missing frame, `ResolveFrame()` connects the chain with another completed chain to form a closed loop and then computes the missing data.`PlotResolveFrame()` adds plotting feature to visualize the closed chain.

Syntax

- `[T_rel, T_abs, fc1]= ResolveFrame(fc1,fc2)`
- `[T_rel, T_abs, fc1]= PlotResolveFrame(fc1,fc2, options)`

Input Arguments

- fc1 – an incomplete chain of coordinate frames created by `SerialFrame()` that has exactly one frame missing somewhere along the chain.
- fc2 – a completed chain of coordinate frames created by `SerialFrame()`, or simply a homogeneous transformation matrix describing the end frame of fc1 w.r.t its base.

Output Arguments

- T_rel – 4 x 4 relative homogeneous transformation matrix of the missing frame w.r.t the frame below it.
- T_abs – 4 x 4 absolute homogeneous transformation matrix of the missing frame w.r.t frame 1 of the chain.
- fc1 – chain passed to the first input argument whose missing information is resolved and is now completed.

Options

Note : options in [ ] must be passed to the function in pairs.

- 'grid' — show grid
- 'oinfo' — show origin location (may look messy when several frames are condensed together)
- ['figure', fnum] — command the plot to a particular window indicated by fnum

Description

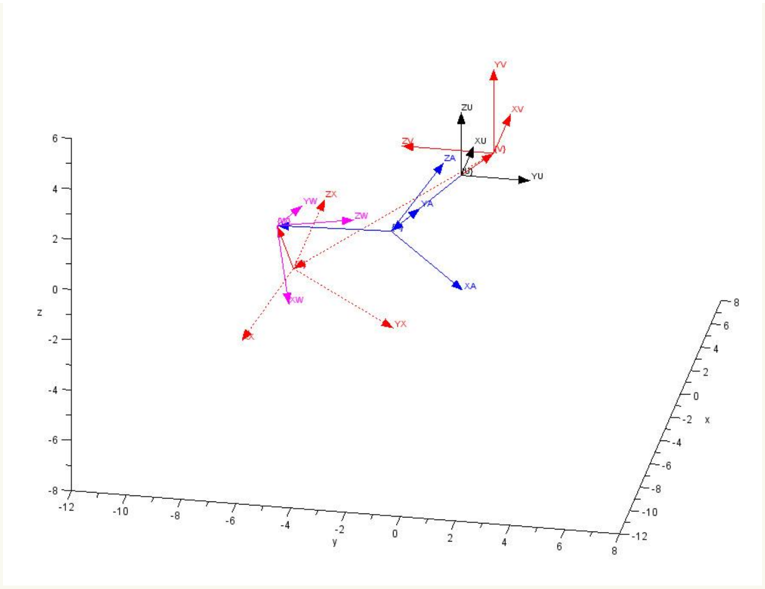
It is a common problem in robotic study when homogeneous transformations are connected in a closed chain configuration, with all information known except one particular link. The goal is to solve for that unknown link in terms of the rest, the known ones. One common algebraic method is to form two transform equations of the two paths from base to end, combine them, and rearrange so that the unknown frame on the left side is described as a product of the known frames and their inversions. Alternatively, one can write a computer program to solve this problem numerically. `ResolveFrame()` is an algorithm designed to perform this job. `PlotResolveFrame()` is just a graphical shell that internally calls `ResolveFrame()`

To use the functions, two loops of the closed chain must be constructed as two separate transformation chain using `Frame()` and `SerialFrame()` . The chain fc1 contains the missing link, while fc2 must be a completed chain with the same base and end frame as fc1 . Alternatively, fc2 can be passed as a homogeneous transformation matrix describing the end frame of fc1 w.r.t its base. If the problem is solvable, `ResolveFrame()` returns the transformations of missing frame together with the chain fc1 that is now completed. `PlotResolveFrame()` also creates a graphical window to display the frames of closed chain formed by fc1 and fc2 . Chain fc1 is drawn in red, while fc2 is in blue. The missing link is shown as a dotted red arrow. The base and end frame are in black and purple, respectively.

Examples

The example below shows the necessary steps from frame creation to plotting. The result is shown below.

```
-->F1(1)=Frame(eye(4,4),'name','U');
-->F1(2)=Frame(transl([1 1
0.5])*trotx(pi/2),'name','V');
-->F1(3)=Frame([],'name','X');  // missing frame at
location 3
-->F1(4)=Frame(transl([1 1 1])*angvec2t(pi/2, [1 1
1]),'name','W');
-->fc1=SerialFrame(F1,'name','Chain 1'); // incomplete
chain to be solved

Reading frame data and computing missing information
Processing Upwards ...
1 -- {U} : Found T_rel. Fill in T_abs with T_rel
2 -- {V}: Found T_rel. Computing T_abs :{V} w.r.t {U}--
Finished
3 -- {X}: *** Missing both T_abs: {X} w.r.t {U} and
T_rel: {X} w.r.t {V} ***
4 -- {W}: *** Missing T_abs: {W} w.r.t {U} ***
List of missing frames in Chain 1
================================
3 -- {X}: T_abs : {X} w.r.t {U} , T_rel : {X} w.r.t {V}
,
4 -- {W}: T_abs : {W} w.r.t {U} ,

-->F2(1)=Frame(eye(4,4),'name','U');
-->F2(2)=Frame(transl([-3 -2 -1])*eul2t([pi/6 pi/3
pi/4]),'name','A');
-->F2(3)=Frame(transl([-3 -4 -
1])*troty(pi/4),'name','W');
-->fc2=SerialFrame(F2,'name','Chain 2');  // chain 2 is
completed

Reading frame data and computing missing information
Processing Upwards ...
1 -- {U} : Found T_rel. Fill in T_abs with T_rel
2 -- {A}: Found T_rel. Computing T_abs :{A} w.r.t {U}--
Finished
3 -- {W}: Found T_rel. Computing T_abs :{W} w.r.t {U}--
Finished
Chain 2: All missing information are computed and
filled in

-->[Trel,Tabs,fc1r]=PlotResolveFrame(fc1,fc2);

Running ResolveFrame ...
4 -- {W}: Fill in T_abs : {W} w.r.t {U}-- Finished
3 -- {X}: Computing T_abs : {X} w.r.t {U} -- Finished
3 -- {X}: Computing T_rel : {X} w.r.t {V} -- Finished
Rechecking if there are still missing frame data in
Chain 1-- none found.
--- Chain 1 is now completed. ---

 Missing data T: {X} w.r.t {V} in Chain 1 is computed
as

    0.2333285    0.0273474    0.9720133  - 7.7659853
  - 0.8371037  - 0.5029826    0.2150952  - 1.4899574
    0.4947881  - 0.8638638  - 0.0944675    5.8682128
    0.           0.           0.           1.
```



Close chain plot using `PlotResolveFrame()`

qpoly

RTSX Category: Path Generation

See `cpoly`.

q2str, q2vec, q2tr, tr2q

RTSX Category: Mathematics →Quaternion

Conversion between quaternion and other types of data.

Syntax

- `qstr = q2str(q)`
- `v = q2vec(q)`
- `[T,R] = q2tr(q)`
- `q = tr2q(R)`
- `q = tr2q(T)`

Input Arguments

- q — a quaternion s <v1,v2,v3>
- R — 3 x 3 rotation matrix
- T — 4×4 homogeneous transformation matrix

Output Arguments

- qstr — a string that represents quaternion value s <v1,v2,v3>
- v — a vector representation [s v1 v2 v3] of quaternion s <v1,v2,v3>
- R — 3 x 3 rotation matrix
- T — 4×4 homogeneous transformation matrix
- q — a quaternion s <v1,v2,v3>

Description

A quaternion is a compact method of representing a 3D rotation that has computational advantages including speed and numerical robustness. A quaternion has 2 parts, a scalar s, and a vector v and is typically written: q = s <vx, vy, vz> This group of functions convert between quaternion and other types of data: string, vector, rotation/homogeneous transform matrix.

- `qstr = q2str(q )` converts a quaternion s <v1, v2, v3> to a string
- `v = q2vec(q)` converts a quaternion s <v1, v2, v3> to a vector v = [s v1 v2 v3].
- `[T,R] = q2tr(q)` returns homogenous transform and rotation matrices corresponding to the unit quaternion q.
- `q = tr2q(R), q = tr2q(T)` returns a unit quaternion corresponding to a rotation matrix and homogeneous transform matrix, respectively.

Examples

```
-->q = quaternion([1 -2 0 4]);
 -->q2str(q)
 ans  =
 1.000000, < -2.000000, 0.000000, 4.000000>
 -->q2vec(q)
 ans  =
    1.  - 2.    0.    4.
```

```
-->T = q2tr(qunit(q))
T  =
 - 0.5238095  - 0.3809524  - 0.7619048    0.
   0.3809524  - 0.9047619    0.1904762    0.
 - 0.7619048  - 0.1904762    0.6190476    0.
   0.           0.           0.           1.
-->tr2q(T)
ans  =
  s: 0.2182179
  v: [-0.4364358,0,0.8728716]
```

qadd, qsubtract, qmult, qdivide, qinv, qpower, qnorm, qunit, qscale

RTSX Category: Mathematics →Quaternion

Various math operations on quaternion.

Syntax

- `q = qadd(q1, q2 )`
- `q = qsubtract(q1, q2)`
- `q = qmult(q1, q2)`
- `q = qdivide(q1, q2)`
- `qi = qinv(q)`
- `qp = qpower(q, p)`
- `qmag = qnorm(q)`
- `qu = qunit(q)`
- `qs = qscale(q, r)`

Input Arguments

- q, q1, q2 — a quaternion s <v1,v2,v3>
- p — integer
- r — real number between 0 – 1

Output Arguments

- q, qi, qp, qu, qs — a quaternion s <v1,v2,v3>
- qmag — a scalar representing norm of quaternion

Description

A quaternion is a compact method of representing a 3D rotation that has computational advantages including speed and numerical robustness. A quaternion has 2 parts, a scalar s, and a vector v and is typically written: q = s <vx, vy, vz> The following math operations on quaternions are implemented.

- `q = qadd(q1, q2 )` and `q = qsubtract(q1, q2 )` add/subtract two quaternions (or arrays of quaternions).
- `q = qmult(q1, q2)` multiplies two quaternions. `q = qmult(q1, v)` rotates vector v by unit-quaternion q1. `q = qmult(q1, s)` mulitplies quaternion q1 by a scalar s.
- `q = qdivide(q1, q2)` returns a quaternion formed by Hamilton product of q1 and inv(q2). `q = qdivide(q1, s)` performs element-wise division of quaternion elements by scalar s.
- `qi = qinv(q)` inverts a unit quaternion.
- `qp = qpower(q, p)` raises quaternion to integer power.
- `qmag = qnorm(q)` computes scalar norm or magnitude of quaternion q. This is simply an Euclidean norm of vector representation of q. A unit quaternion has norm of 1.
- `qu = qunit(q)` unitizes a quaternion.
- `qs = qscale(q, r)` interpolates rotations expressed by quaternion. qs is a unit quaternion interpolated between identity for r = 0 to q for s = 1. This is a spherical linear interpolation (slerp) that can be interpreted as interpolation along a great circle arc on a sphere.

qinterp

RTSX Category: Mathematics →Quaternion

Interpolates a rotation between two quaternions.

Syntax

- `q = qinterp(q1, q2, r)`

Input Arguments

- q1, q2 — quaternion s <v1,v2,v3>
- r — a vector of points between 0 and 1

Output Arguments

- q — a vector of quaternions corresponding to sequential elements of r

Description

`q = qinterp(q1, q2, r)` is an interpolation between two unit quaternions that produces a rotation around a fixed axis in space. Quaternion interpolation is achieved using spherical linear interpolation (slerp) in which the unit quaternions follow a great circle path on a 4-dimensional hypersphere.

Examples

The example below demonstrates how to apply qinterp( ) to coordinate frame rotation.

```
-->R0 = rotz(-pi/3)*roty(-pi/3);       // start
frame
 -->R1 = rotz(pi/3)*roty(pi/3);         // end frame
 -->q0 = Quaternion(R0);     // creates start and
end quaternions
 -->q1 = Quaternion(R1);
 -->q = qinterp(q0, q1, [0:49]'/49);     // perform
interpolation
 -->tranimate(q)               // rotation animation
```

Quaternion

Create a quaternion s <v1,v2,v3>, which is useful for representing 3D rotation.

Syntax

- `q = Quaternion( )`
- `q = Quaternion(Q1)`
- `q = Quaternion([s v1 v2 v3])`
- `q = Quaternion(s)`
- `q = Quaternion(V)`
- `q = Quaternion(theta, K)`
- `q = Quaternion(R)`
- `q = Quaternion(T)`

Input Arguments

- Q1 — a quaternion s <v1,v2,v3>
- [s v1 v2 v3] — a 1 x 4 vector representaion of quaternion
- s — a scalar part of quaternion
- V — a 1 x 3 vector part of quaternion
- theta — rotation angle (radian)
- K — a 1 x 3 vector representing axis of rotation
- R — 3 x 3 rotation matrix
- T — 4×4 homogeneous transformation matrix

Output Arguments

- q — a quaternion s <v1,v2,v3>

Description

A quaternion is a compact method of representing a 3D rotation that has computational advantages including speed and numerical robustness. A quaternion has 2 parts, a scalar s, and a vector v and is typically written: q = s <vx, vy, vz>

- `q = Quaternion()` creates an identity quaternion 1 <0, 0, 0>
- `q = Quaternion(Q1)` creates a copy of quaternion Q1
- `q = Quaternion([s v1 v2 v3])` creates a quaternion from 1 x 4 vector representation
- `q = Quaternion(s)` creates a quaternion with scalar part s and zero vector part
- `q = Quaternion(V)` creates a quaternion 0 < V >
- `q = Quaternion(theta, K)` creates a unit quaternion representing a rotation of theta about vector K
- `q = Quaternion(R)` creates a unit quaternion corresponding to a rotation matrix R. If R (3x3xN) is a sequence then q (Nx1) is a vector of quaternions corresponding to the elements of R.
- `q = Quaternion(T)` creates a unit quaternion corresponding to a homogeneous transform T. If T (4x4xN) is a sequence then q (Nx1) is a vector of quaternions corresponding to the elements of T.

Examples

```
-->q = Quaternion([2 1 -1 4])
 q  =
   s: 2
   v: [1,-1,4]

-->q = Quaternion(pi/4,[1 -1 1])
 q  =
   s: 0.9238795
   v: [0.2209424,-0.2209424,0.2209424]

-->T = trotx(pi/2);
-->q = Quaternion(T)
 q  =
   s: 0.7071068
   v: [0.7071068,0,0]
```

r2t, rd2t, t2d, t2r

RTSX Category: Mathematics → Homogeneous Transformations

Data conversion between rotation matrix R, origin vector d, and homogeneous transformation matrix T

Syntax

- `d = t2d(T)`
- `R = t2r(T)`
- `T = r2t(R)`
- `T = rd2t(R, d)`

Input/Output Arguments

- R — 3×3 rotation matrix
- T — 4×4 homogeneous transformation matrix
- d – a 3×1 vector representing location

Description

This group of functions extract or combine data in different forms. `t2d()` and `t2r()` extracts the location vector d and rotation matrix R from a homogeneous transformation matrix T. `r2t()` forms T by combining input argument R with d = [0, 0, 0] '. `rd2t()` forms T by combining input arguments R and d.

rne

RTSX Category: Dynamics

Compute inverse dynamics of a robot using recursive Newton-Euler algorithm

Syntax

- `Tau = rne(robot, q, qd, qdd, grav, fext)`

Input Arguments

- robot — n-link robot data structure created with `SerialLink()`

- q — 1 x n joint position
- qd — 1 x n joint velocity
- qdd — n x 1 joint acceleration
- grav* — gravitional acceleration vector
- fext * — a 1 x 6 wrench vector [Fx Fy Fz Mx My Mz] acting on the tool end of the robot.

* Optional inputs

Output Arguments

- Tau — 1 x n joint torque

Description

`rne()` computes inverse dynamics using Newton-Euler algorithm. The computed torque contains contribution from armature inertia and joint friction. The homogeneous transform of robot base is ignored.

Examples

```
-->exec('./models/mdl_puma560.sce',-1);
 -->Tq = rne(p560, q_n, q_z, q_z)
 Tq  =
    0.     31.63988    6.035138    0.     0.0282528    0.
-->Tq = rne(p560, q_n, [1 0 0 0 0 0], q_z) // joint 1
moving at 1 rad/s
 Tq  =
  - 30.533206    32.267903    5.6743908  - 0.0003056
0.0282528    0.
```

Robot2AT

RTSX Category: Kinematics → Forward Kinematics

From a robot model, compute homogeneous transformation matrices from each frame to base.

Syntax

- `[A,T,Tb,Tt] = Robot2AT(robot,q,options)`

Input Arguments

- robot –a robot model
- q –a 1 x nq vector of joint variable values, where nq = number of joint variables

Output Arguments

- A — a 4 x 4 x nq homogeneous transformation matrix representing each frame w.r.t. frame below, where nq = number of joints
- T — a 4×4 homogeneous transformation matrix representing tool frame w.r.t base
- Tb — a 4×4 base transformation matrix
- Tt — a 4×4 tool transformation matrix

Options

Note: options only affect homogeneous matrix T. A remains unchanged.

- 'none' — T =A1*A2* …*An
- 'base' — T = robot.base*A1*A2* … *An
- 'tool' — T = A1*A2* …*An*robot.tool
- 'all' — T=robot.base*A1*A2*…*An*robot.tool

Description

`Robot2AT()` computes homogeneous transformation matrices that describe each joint coordinate frame A and T, where A(:,:,i) describes frame {i} w.r.t {i-1} and T describes {n} w.r.t {0}

Examples

```
exec('./models/mdl_puma560.sce',-1);
p560 = AttachBase(p560,transl([1,1,2])); // add  base
transform
p560 = AttachTool(p560,trotx(pi/2)); // add tool
```

```
transform
[A,T,Tb,Tt]=Robot2AT(p560,q_n); // base/tool are
excluded from T
[A,T]=Robot2AT(p560,q_n,'base'); // T = Tb*T
[A,T]=Robot2AT(p560,q_n,'tool'); // T = T*Tt
[A,T]=Robot2AT(p560,q_n,'all');  // T=Tb*T*Tt
[A,T]=Robot2AT(p560,q_n,'base','tool'); // this is
equivalent to 'all'
```

Robot2hAT

RTSX Category: Kinematics → Forward Kinematics

From a robot model and a sequence of joint variable values, compute homogeneous transformation matrices from each frame to base.

Syntax

- `[A,T] = Robot2hAT(robot,qs)`

Input Arguments

- robot –a robot model
- q –a ns x nq matrix of joint variable values, where ns = number of setpoints (move steps) and nq = number of joint variables

Output Arguments

- A — a 4 x 4 x ns x (nq+2) homogeneous transformation hyper-matrix representing each frame w.r.t. frame below, where ns = number of setpoints (move steps) and nq = number of joint variables
- T — a 4×4 x ns x (nq+2) homogeneous transformation matrix representing each frame w.r.t base

Description

Robot2hAT() can be thought of as an extension of Link2AT() to handle a sequence of joint variable values. The output matrices are now 4-dimensional. Another difference is Robot2hAT() accepts a robot model as input argument and takes into account the base and tool frame of the robot.

Frankly, Robot2hAT() is designed to support animation functions and is somewhat too complicated for direct use. To solve a standard forward kinematics problem with a single set of joint variables, FKine() or Link2AT() suffices.

Examples

The following example shows how to compute hyper-matrices A and T from a two-link manipulator with a sequence of joint variables.

```
clear L;
a1 = 1.2; a2 = 1;
L(1)=Link([0 0 a1 0]);
L(2)=Link([0 0 a2 0]);
twolink=SerialLink(L);  // a 2-link manipulator
// generate simple setpoints
// both joints move full circle
t = [0:0.01:1]';        // "time" data
qs = [2*pi*t 2*pi*t];
[A,T]=Robot2hAT(twolink, qs);
```

RobotInfo

RTSX Category: Kinematics → Robot Model Creation and Graphics

RobotInfo() is a function to display information of a robot model created by SerialLink().

Syntax

- RobotInfo(robot)

Input Arguments

- robot –a robot model created by `SerialLink()` .

Output Arguments

- none.

Description

`RobotInfo()` prints essential information of a robot model, such as name, manufacturer, DH parameters, etc., in Scilab console window.

Examples

```
-->exec('./models/mdl_puma560.sce',-1);
-->RobotInfo(p560)

================= Robot Information ===============
Robot name: Puma 560
Manufacturer: Unimation
Number of joints: 6
Configuration: RRRRRR
Method: Standard DH
+---+------------+------------+------------+------------+
| j |   theta    |      d     |      a     |   alpha    |
+---+------------+------------+------------+------------+
| 1 |     q1     |    0.00    |    0.00    |   1.57     |
| 2 |     q2     |    0.00    |    0.43    |   0.00     |
| 3 |     q3     |    0.15    |    0.02    |   -1.57     |
| 4 |     q4     |    0.43    |    0.00    |   1.57     |
| 5 |     q5     |    0.00    |    0.00    |   -1.57     |
| 6 |     q6     |    0.00    |    0.00    |   0.00     |
+---+------------+------------+------------+------------+
Gravity =
    0.
    0.
    9.81
Base =
    1.     0.     0.     0.
    0.     1.     0.     0.
    0.     0.     1.     0.
    0.     0.     0.     1.
Tool =
    1.     0.     0.     0.
    0.     1.     0.     0.
    0.     0.     1.     0.
    0.     0.     0.     1.
```

rotx, roty, rotz

trotx, troty, trotz

RTSX Category: Mathematics → Rotations

Basic rotation about coordinate axis X,Y, or Z

Syntax

- `[R] = rotx(theta)`
- `[R] = roty(theta)`
- `[R] = rotz(theta)`
- `[T] = trotx(theta)`
- `[T] = troty(theta)`
- `[T] = trotz(theta)`

Input Arguments

- theta – rotation angle (default in radian. See options below)

Output Arguments

- R — 3×3 rotation matrix
- T — 4×4 homogeneous transformation matrix

Options

- Put 'deg ' as second argument to input angle value in degree. Ex. `R=rotx(45, 'deg ')`

Description

Compute basic rotation around main coordinate axis X, Y, Z. The argument input value must is defaulted to radian. `rotx(),roty(),rotz()` returns a 3×3 basic rotation matrix R, while `trotx(),troty(), trotz()` returns a 4×4 homogeneous translation matrix `T = [R d;0 0 0 1]` with `d = [0,0,0]'`.

Examples

```
-->R=rotx(pi/2)
 R  =

    1.     0.     0.
    0.     0.   - 1.
    0.     1.     0.

-->T=trotz(pi/4)
 T  =

    0.7071068  - 0.7071068    0.     0.
    0.7071068    0.7071068    0.     0.
    0.           0.           1.     0.
    0.           0.           0.     1.
```

rpy2r, rpy2t

tr2rpy

RTSX Category: Mathematics → Rotations

Convert beween Row-Pitch-Yaw angles and rotation/homogeneous transformation matrix.

Syntax

- `[R] = rpy2r(rpy)`
- `[T] = rpy2t(rpy)`
- `[rpy] = tr2eul(R)`
- `[rpy]= tr2eul(T)`

Input/Output Arguments

- rpy – 1×3 vector representing Row-Pitch-Yaw angle (default in radian)
- R — 3×3 rotation matrix
- T — 4×4 homogeneous transformation matrix

Options

- 'zyx ' — return solution for sequential rotations about Z,Y,X axes
- 'deg ' — input and output argument values in degrees instead of radians

Description

rpy2r(), rpy2t() returns a rotation and homogeneous transformation matrix representing row,pitch, yaw angles corresponding to rotations about X,Y,Z axes respectively (or Z,Y,X if 'zyx ' option is selected) . tr2rpy() receives a rotation or homogeneous transformation matrix and solves for row-pitch-yaw angles.

Examples

```
-->R=rpy2r([0.1 0.2 0.3])
 R  =
    0.9362934  - 0.2896295    0.1986693
    0.3129918    0.9447025  - 0.0978434
  - 0.1593451    0.153792     0.9751703

-->rpy=tr2rpy(R)
 rpy  =
    0.1    0.2    0.3
```

SerialFrame

RTSX Category: Mathematics → Homogeneous Transformation

Form a transformation chain from a frame structure created by Frame()

Syntax

- fc = SerialFrame(F, options)

Input Arguments

- F – a frame structure created by Frame()

Output Arguments

- fc — frame chain structure containing the following fields
  - Frame(i) — indexed frames in the chain with the same sequential order as in F
  - nf — number of frames in the chain
  - name — a string specifying chain name
  - completed — a boolean value (0 or 1) to flag the completion of the chain
  - missingframe — frame number that has missing link (no Tabs and Trel information)
  - viewangle — a view angle data used by `PlotFrame()`

Options

Note : options in [ ] must be passed to the function in pairs.

- ['name', '(chain name)'] — chain name. It is printed in output messages in Scilab window. A simple identification like 'chain 1′ is often helpful, especially when two chains are passed to `ResolveFrame()`
- ['viewangle', [alpha, theta]] — Used in `PlotFrame()` and `PlotResolveFrame()`. These two values give the spherical coordinates of the observation points (in degrees).

Description

After a frame data structure is created by `Frame()` , use `SerialFrame()` to form a homogeneous tranformation chain with lowest number frame (1) as base and highest number frame as top. One blank frame can be included in the chain, to be solved later by `ResolveFrame()` when another chain is connected to form a closed chain. When `SerialFrame()` is invoked, the function will try to fill in missing information if it has enough information to solve. If there is no missing frame present in the frame structure, all missing information (either Trel or Tabs of each frame) can be computed and the chain is marked as completed by setting the completed field to 1. If `SerialFrame()` cannot solve for missing information at this time but knows it can be done when closed chain is formed, it will reset completed field to 0. For the worst case when a frame structure cannot be completed by any function, `SerialFrame()` stops and shows error message. These cases happen, for example, when more than one missing frame are present in the chain, or Tabs instead of Trel is given to any frame above the missing frame.

Examples

This example shows how to construct a frame chain using `Frame()` and `SerialFrame()`. From the output messages, we see that the algorithm could complete information upwards the chain until it reaches location 4, where a black frame is inserted. The missing data above that location has to be left incomplete. The missing frames can be computed by `ResolveFrame()`, providing another (completed) chain forming a closed loop .

```
-->F(1)=Frame(eye(4,4),'name','U');   // base frame

-->F(2)=Frame(trotx(pi/2)*transl([0 1 1]),'name','V'); // 2nd
frame (default T input is relative)

-->F(3)=Frame(transl([2 3 -1]),'abs','name','W');  // 3rd
frame is inputted as absolute T w.r.t base

-->F(4)=Frame([],'name','X');  // a blank frame can be
inserted at only one place in chain

-->F(5)=Frame(troty(pi/3)*transl([3 2 -1]),'name','Y');  //
last frame in structure

-->fc1=SerialFrame(F,'name','Chain 1');

Reading frame data and computing missing information
Processing Upwards ...

1 -- {U} : Found T_rel. Fill in T_abs with T_rel
2 -- {V}: Found T_rel. Computing T_abs :{V} w.r.t {U}--
Finished
3 -- {W}: Found T_abs. Computing T_rel :{W} w.r.t {V}--
Finished
4 -- {X}: *** Missing both T_abs: {X} w.r.t {U} and T_rel:
{X} w.r.t {W} ***
5 -- {Y}: *** Missing T_abs: {Y} w.r.t {U} ***

List of missing frames in Chain 1
===============================
4 -- {X}: T_abs : {X} w.r.t {U} , T_rel : {X} w.r.t {W} ,
5 -- {Y}: T_abs : {Y} w.r.t {U} ,
```

SerialLink

RTSX Category: Kinematics → Robot Model Creation and Graphics

`SerialLink()` is a function that "assembles" a robot model from a link structure created by `Link()` .

Syntax

- `robot = SerialLink(L,options)`

Input Arguments

- L –a link data strucure created by `Link()` consisting of link and joint parameters ordered from bottom to top.

Output Arguments

- robot — a robot data structure containing the following fields
  - name — robot name
  - manuf — robot manufacturer
  - nj — number of joints (or links). This field is created automatically by counting links in L.
  - Link(1:nj) — link data with the same parameters inherited from link structure L
  - gravity — direction of gravity [gx gy gz]
  - base — base frame of robot (4 x 4 homogeneous transformation matrix)
  - tool — tool frame of robot (4 x 4 homogeneous transformation matrix)
  - conf — robot configuration string corresponding to joint types from bottom to top. Ex. 'RRR', 'RPR'. This field is created automatically from link data structure L.
  - viewangle — suggested view angle in spherical coordinate [alpha, theta], in degrees
  - comment — general comment string for this robot

Options

Note : options in [ ] must be passed to the function in pairs.

- ['name', 'robot name'] — give the robot model some name for reference.
- ['manuf', 'robot manufacturer'] — robot manufacturer company name.
- ['comment', 'comment string'] — a comment for this robot.
- ['viewangle', [alpha,theta]] — suggested view angle (in degrees) in spherical coordinate. This is used to set up rotation_angle property of a graphic window when the robot is plotted or animated.
- ['base', Tb] — specify a base frame for the robot, Tb must be a 4 x 4 homogeneous transformation matrix
- ['tool', Tt] — specify a tool frame for the robot, Tt must be a 4 x 4 homogeneous transformation matrix
- ['gravity', value] — direction of gravity [gx gy gz]

Description

After a link data structure L is created with `Link()` , a robot model can be formed by passing L to `SerialLink()`. In addition to link data already in L, `SerialLink()` adds some global parameters such as robot name and manufacturer, base and tool frames, gravity information (if provided). It also creates and fills in useful information such as number of joints and joint configuration.
While assembling a robot model, `SerialLink()` can perform some sanity check for link data. For example, if one inputs mixed DH types into L that will result in an error when executing the function. Validity of data types are also checked.
Whenever you modify a link data structure, you must pass it to `SerialLink()` again so that the robot model is updated properly. Never write to a field manually to avoid data inconsistency. Instead, use a function designed for that purpose.

Examples

This example demonstrates how to make an RRR robot.

```
d1 = 1; a2 = 1; a3 = 1;
L(1)= Link([0 d1 0 pi/2]);
L(2)=Link([0 0 a2 0]);
L(3)=Link([0 0 a3 0]);
```

```
RRR_robot=SerialLink(L,'name','An RRR robot arm');
```

t2d, t2r

RTSX Category: Mathematics →Homogeneous Transformation

See `r2t`

tr2angvec, tr2eul, tr2rpy

RTSX Category: Mathematics →Rotations

See `angvec2r, eul2r, rpy2r`

tr2jac

RTSX Category: Kinematics →Velocity Kinematics

See `eul2jac`

tr2q

RTSX Category: Mathematics →Quaternion

See `q2tr`

tranimate

RTSX Category: Mathematics →Homogeneous Transformation

Animate a 3D motion from a start frame to end frame.

Syntax

- `tranimate(T1, T2, options)`
- `tranimate(T,options)`
- `tranimate(Tseq,options)`

Input Arguments

- T, T1, T2 – a 4 x 4 homogeneous transformation matrix , or 3 x 3 rotation matrix, or quaternion data structure
- Tseq – a 4 x 4 x nq homogeneous transformation matrix sequence, or 3 x 3 x nq rotation matrix sequence

Output Arguments

- none.

Options

Note : options in [ ] must be passed to the function in pairs.

- 'world ' – plot world coordinate for comparison
- 'grid ' — show grid
- 'hold ' — hold frames from previous plots. Must use with 'figure' option to plot on the same figure
- 'holdstart ' — hold start frame
- 'noreplay ' — bypass the replay loop at end of function
- ['figure', fnum] — command the plot to a particular window indicated by fnum
- ['speed', s] — animation speed 1 = slowest, 10 = fastest
- ['nsteps', n] — number of steps along the path (default 50)

- ['color', '(your choice)'] — plot in a particular color. Choices are‘black’, ‘blue’, ‘green’, ‘cyan’, ‘red’, ‘purple’, ‘yellow’, ‘white’
- ['linestyle','(your choice)'] — plot in a particular line style. Choices are‘solid’ (‘-’), ‘dash’ (‘–’), ‘dash dot’ (‘-.’), ‘longdash dot’ (‘–.’), ‘bigdash dot’ (‘=.’), ‘bigdash longdash’ (‘=–’), ‘dot’ (‘.’), ‘double dot’ (‘..’)

Description

tranimate(T1, T2, options) animates a 3D coordinate frame moving from T1 to T2, where T1 and T2 can be represented by homogeneous transformation matrices (4 x 4), rotation matrices (3 x 3), or quaternions
tranimate(T, options) animates a coordinate frame moving from world frame to T
tranimate(Tseq, options) animates a trajectory Tseq

Examples

Try the following commands to see the response.

```
R = rotx(pi/2);
tranimate(R);
tranimate(R,'world');
R0 = rotz(-1)*roty(-1);
R1 = rotz(1)*roty(1);
q0 = Quaternion(R0);
q1 = Quaternion(R1);
q = qinterp(q0, q1, [0:49]'/49);
tranimate(q);
```

transl

RTSX Category: Mathematics →Translation

Translation in 3-D

Syntax

- `T= transl(p)`

Input Arguments

- p – a 3 x 1 (1 x 3 is acceptable) vector representing new location

Output Arguments

- T — 4×4 homogeneous transformation matrix

Description

transl( ) returns a homogeneous transformation matrix representing a translation to a new location specified by vector p. The rotation part is just a 3×3 identity matrix.

Examples

```
-->T=transl([-1 1 2])
 T  =
    1.     0.     0.   - 1.
    0.     1.     0.     1.
    0.     0.     1.     2.
    0.     0.     0.     1.
```

trinterp

RTSX Category: Mathematics →Homogeneous Transformation

Homogeneous transform interpolation between two frames

Syntax

- `T = trinterp(T0, T1, S)`
- `T = trinterp(T, S)`

Input/Output Arguments

- T0, T1, T — 4×4 (or 4x4xn) homogeneous transformation matrices
- S – a point of vector of points between 0 – 1

Description

T = trinterp(T0, T1, S) is a homogeneous transform interpolation between T0 when S = 0 to T1 when S=1. Rotation is interpolated using quaternion spherical linear interpolation. If S has size n x 1, then T (4x4xn) is a sequence of homogeneous transforms corresponding to the interpolation values in S.
T = trinterp(T,S) is a transform that varies from the world frame when S = 0 to T when S = 1

Examples

```
T0 = transl([0.4, 0.2 0])*trotx(pi);
T1 = transl([-0.4, -0.2, 0.3])*troty(pi/2)*trotz(-
pi/2);
Ts = trinterp(T0, T1, [0:49]/49);
tranimate(Ts);
```

trotw

RTSX Category: Mathematics →Rotation

Rotation of a frame about an axis X,Y, or Z of world, or fixed coordinate frame (refering to a frame with origin at [0, 0, 0]‘ and never moves or rotates)

Syntax

- `[R] = trotw(R, theta, axis)`
- `[T] = trotw(T,theta, axis, options)`

Input Arguments

- R – a 3 x 3 rotation matrix
- T – a 4 x 4 homogeneous transformation matrix with origin at [0,0,0]‘
- theta – rotation angle (default in radian. See options below)

- axis — a character indicating the rotation axis 'x','y',or 'z' (either lower or upper case, but must be put in quotes)

Output Arguments

- R — a 3×3 rotation matrix is returned when first input is a rotation matrix
- T — a 4×4 homogeneous transformation matrix is returned when first input is a homogenous transformation matrix

Options

- 'deg' : input angle value is in degree. Ex. `Tr=trotw(T,45, 'x', 'deg')`

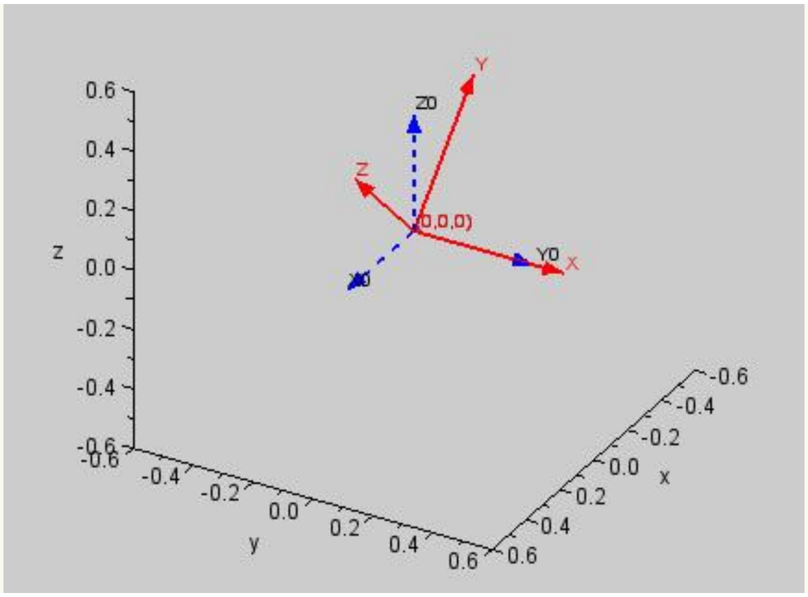
Description

Compute a rotation/homogeneous transformation matrix representing a rotation around one of the world coordinate axes X, Y, Z, depending on the choice of third input argument. The origin of world and rotated frame must be at the same point [0,0,0]'.

Examples

This example demonstrates a combination of two rotations. First we rotate 45 degree about X axis, then rotate 90 degree about the *world* Z axis (not the current Z axis). Use the function `trplot()` to see the resulting rotated frame below.

```
   -->T=trotw(trotx(pi/4),pi/2,'z')
    T  =
        0.   - 0.7071068    0.7071068    0.
       1.    0.            0.            0.
       0.    0.7071068     0.7071068     0.
       0.    0.            0.            1.
    -->trplot(T,'world')
```



rotation about a world coordinate axis

trotx, troty, trotz

RTSX Category: Mathematics →Rotations

See `rotx, roty, rotz`

trplot

RTSX Category: Mathematics →Homogeneous Transformation

Plot a coordinate frame compared to world frame. Can hold previous frames for comparison.

Syntax

- `trplot(T,options)`

Input Arguments

- T – a 4 x 4 homogeneous transformation matrix

Output Arguments

- none.

Options

Note : options in [ ] must be passed to the function in pairs.

- 'world ' – plot world coordinate for comparison
- 'grid ' — show grid
- 'hold ' — hold frames from previous plots. Must use with ‘figure’ option to plot on the same figure
- ['figure', fnum] — command the plot to a particular window indicated by fnum
- ['color', '(your choice)'] — plot in a particular color. Choices are‘black’, ‘blue’, ‘green’, ‘cyan’, ‘red’, ‘purple’, ‘yellow’, ‘white’
- ['linestyle','(your choice)'] — plot in a particular line style. Choices are‘solid’ (‘-’), ‘dash’ (‘–’), ‘dash dot’ (‘-.’), ‘longdash dot’ (‘–.’), ‘bigdash dot’ (‘=.’), ‘bigdash longdash’ (‘=–’), ‘dot’ (‘.’), ‘double dot’ (‘..’)

Description

`trplot()` is a simple graphic function that can be used as quick visual check for a transformed coordinate frame compared with world frame, or a couple of transformations compared together. It is best used to compare rotated frame at the origin. When it is used with a frame translated far away from the orign, the function may have difficulty scaling the axes, especially when previous plots are held by ‘hold’ option. Using `PlotFrame()` is recommended for a chain of frames involving both rotation and translation.

Examples

Try the following commands to see the response.

```
T=trotx(pi/2);  // rotate pi/2 about X axis
```

```
trplot(T);      // plot without world frame
trplot(T,'world','grid');      // plot together with world frame and grid
trplot(T,'figure',1,'color','blue','linestyle','-.');
// plot a dash-dot, blue frame in graphic window no 1
T2=trotz(pi/4);    // make another frame rotated pi/4 about Z axis
trplot(T2,'figure', 1, 'hold', 'color','yellow', 'linestyle', '.');
// compare a new dotted, yellow frame with previous
```

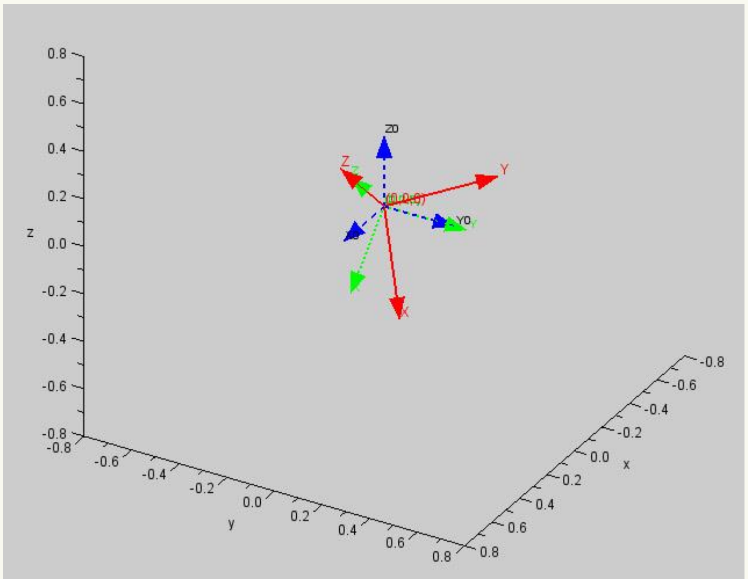
Suppose you want to see the resulting frames from the following steps:

1. rotate 45 degree about Y axis
2. rotate 30 degree about current Z axis

Issuing these commands

```
T1= troty(45,'deg');    // first rotation
trplot(T1,'world','figure',1,'color','green','linestyle','.');
// plot first frame in dotted green
T2 = T1*trotz(30,'deg');       // combined rotation of step 1 and 2
trplot(T2,'figure',1,'color','red','hold');
// hold previous frame and plot second frame in red
```

yields the plot shown in Figure 1 below. The first frame is shown in dotted green and second frame in solid red. Verify that they correspond correctly to the combined rotation steps above.



combined rotation example with `trplot()`

UpdateRobot

UpdateRobotLink

RTSX Category: Kinematics →Robot Model Creation and Graphics

These two functions are used to add/change data in a robot model or one of its links.

Syntax

- `robot = UpdateRobot(robot, options)`
- `robot = UpdateRobotLink(robot, i, options)`

Input Arguments

- robot –a robot model to modify
- i –index of robot link to modify

Output Arguments

- robot — updated robot model

Options

An option specific to a function is given that function name in ( ). Otherwise, it can be used for both functions.

Note : options in [ ] must be passed to the function in pairs.

- ['name', 'robot name'] — (UpdateRobot) add/replace robot name.
- ['manuf', 'robot manufacturer'] — (UpdateRobot) add/replace robot manufacturer.
- ['comment', 'comment string'] — (UpdateRobot) add/replace comment
- ['viewangle', [alpha,theta]] — (UpdateRobot) add/replace view angle (in degrees) in spherical coordinate. This is used to set up rotation_angle property of a graphic window when the robot is plotted or animated.
- ['base', Tb] — (UpdateRobot) add/replace a base frame for the robot, Tb must be a 4 x 4 homogeneous transformation matrix

- ['tool', Tt] — (UpdateRobot) add/replace a tool frame for the robot, Tt must be a 4 x 4 homogeneous transformation matrix
- ['gravity', value] — (UpdateRobot) add/replace direction of gravity [gx gy gz]
- ['link', value] — (UpdateRobot) select a particular link for update
- ['RP', (0,'R' /1,'P')] — add/change joint type. Pur 0 or 'R' for revolute, 1 or 'P' for prismatic.
- ['qlim', [qmin qmax]] — add/change joint variable limits. Pass minimum and maximum values in a 1×2 vector as shown.
- ['linkparm', [theta d a alpha (offset)]] — update whole DH parameters with a 1×4 or 1×5 parameter vector
- ['theta', value] — update theta (joint angle)
- ['d', value] — update d (link offset)
- ['a', value] — update a (link length)
- ['alpha', value] — update alpha (link twist angle)
- ['offset', value] — update joint variable offset
- ['dynparm', [r I m Jm G B Tc]] — update whole dynamic parameters with a 1 x 15 parameter vector
- ['r, [1 x 3]] — update link center of gravity, 1 x 3 vector
- ['I', [1 x 6]] — update link inertia matrix, 1 x 6 vector
- ['m', value] — update link mass
- ['Jm', value] — update motor inertia
- ['G', value] — update gear ratio
- ['B', value] — update link viscous friction
- ['Tc', [1 x 2]] — update link Coulomb friction, 1 x 2 vector

Description

Use `UpdateRobot()` or `UpdateRobotLink()` to add/change data in a robot model. While it is possible to access any data in the model directly, we advise against that error-prone practice. These two functions provide some data type and consistency check. Fields such as number of joints and joint configuration are also automatically updated.

Examples

```
L(1)=Link([0 0 1.2 0]);
L(2)=Link([0 0 1 0]);
twolink=SerialLink(L); // a 2-link manipulator
twolink=UpdateRobot(twolink,'name','Robo2'); // add name
twolink=UpdateRobot(twolink,'base',transl([1 1 0]),'tool',troty(pi/2));
// add base & tool frame
twolink=UpdateRobot(twolink,'link',1,'a',1.5);
// change link 1 length to 1.5
twolink=UpdateRobotLink(twolink,1, 'a',1.5);
// same operation as above
```

Model script files for standard robots in subdirectory /models

File name	Name in Scilab workspace	Robot type
mdl_cylindrical.sce	cylind_robot	cylindrical
mdl_puma560.sce	p560	PUMA 560
mdl_scara.sce	scara_robot	SCARA
mdl_spherical.sce	sph_robot	spherical
mdl_sphwrist.sce	sph_wrist	spherical wrist
mdl_stanford.sce	stanf	Stanford arm
mdl_twolink.sce	twolink	two-link planar

List of Xcos model files used in Chapter 5, in subdirectory /xcos

- /pid_examples
  - pid_1joint.zcos -- simple PID feedback (Ex 5.2)
  - pid_1joint_dist.zcos -- PID control with disturbance (Ex 5.3)
  - pid_1joint_ilim_compare.zcos -- demonstrate effect from input saturation in PID control (Ex 5.4)
  - pid_1joint_track.zcos -- tracking with PID and feedforward control (Ex 5.5)
  - pid_cascade_1joint_track.zcos -- cascade PID control (Ex 5.6)
  - servojoint_vpid.zcos -- PID velocity control for advanced servomotor (Ex 5.7)
  - pid_servojoint.zcos -- cascade PID control for advanced servomotor (Ex 5.7)
- /linear_examples
  - ppol_1joint.zcos -- state feedback control (Ex 5.8)
  - ppol_int_1joint.zcos -- state feedback with integrator (Ex 5.9)
  - ofb_servojoint.zcos -- single-loop output feedback H_∞ control (Ex 5.10)
  - vofb_servojoint.zcos -- velocity loop H_∞ control (Ex 5.11)
  - cofb_servojoint.zcos -- cascade H_∞ control (Ex 5.11)
- /nonlinear_examples
  - mdl_2link.zcos -- two-link manipulator model (Ex 5.12)
  - pd_2link_step.zcos -- two-link manipulator with PD control, gravity compensation, and step input (Ex 5.13)
  - pd_2link_track.zcos -- two-link manipulator tracking with PD control and gravity compensation (Ex 5.14)

- invdyn_2link.zcos -- two-link manipulator with inverse dynamics control (Ex 5.15). Must run script file setup2link.sce first to initialize variables.
- adaptive_2link_track.zcos -- two-link manipulator with adaptive control (Ex 5.16). Must run script file setup_adaptive_2link.sce first to initialize variables.