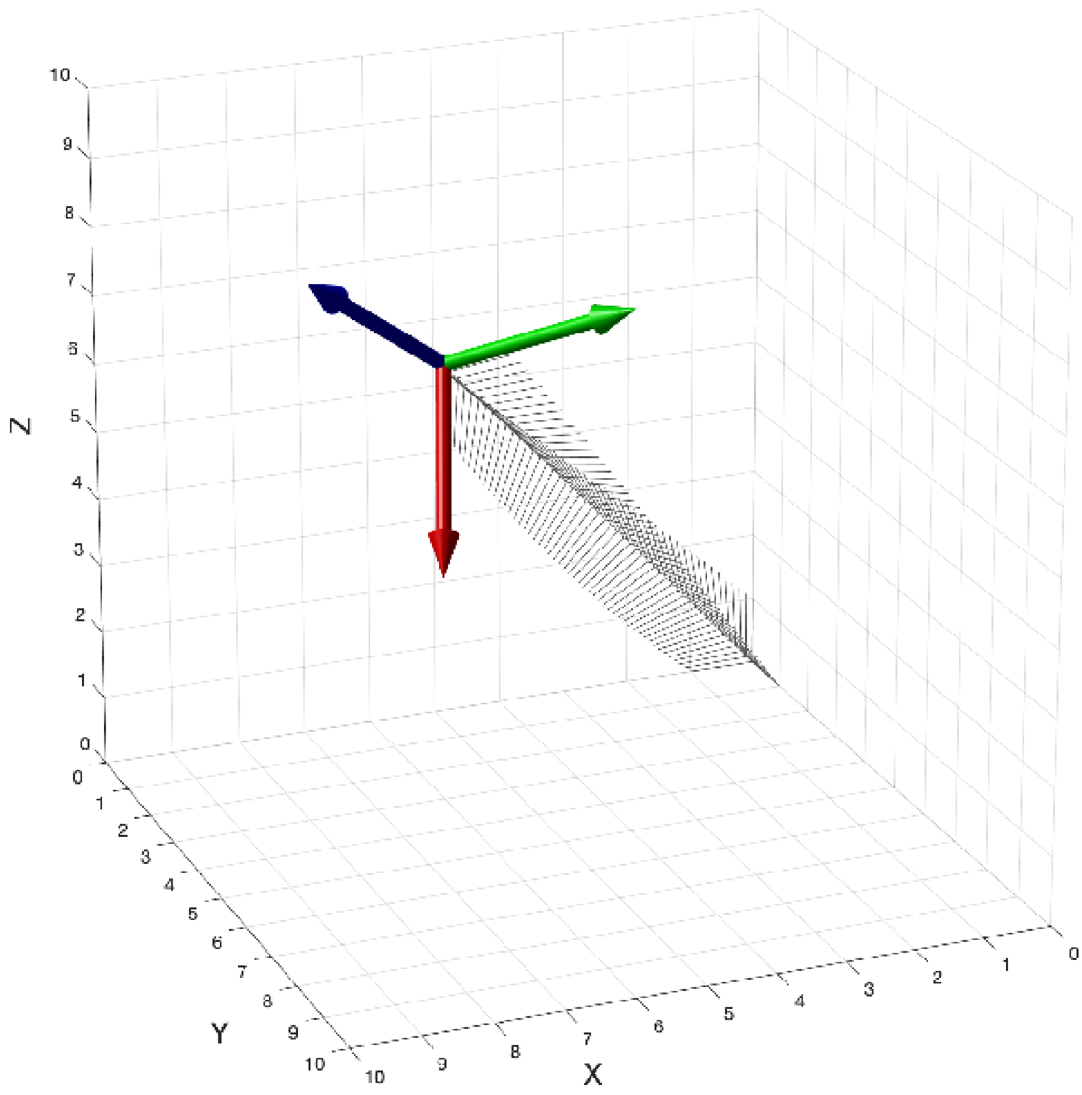
Spatial Math

Toolbox for MATLAB®

Release 1.0

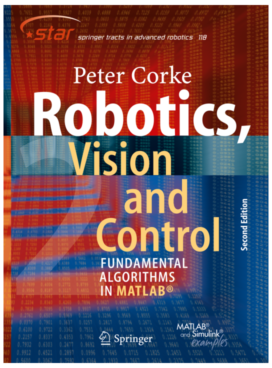


Peter Corke

Release	1.0
Release date	February 2021

Licence	MIT
Toolbox home page	https://github.com/petercorke/spatial-math
Discussion group	https://tiny.cc/rvcforum

Preface


star springer tracts in advanced robotics 118
Peter Corke
Robotics, Vision and Control
FUNDAMENTAL ALGORITHMS IN MATLAB®
Second Edition
Springer
MATLAB® and Simulink® examples

This is the first release of the Spatial Math Toolbox which has been refactored from the Robotics Toolbox for MATLAB. The latter represents over twenty five years of continuous development and a substantial level of maturity – a significant part of that code base was concerned with representing position, orientation and pose in 2D and 3D as well as lines in 3D using Plücker coordinates.

This MATLAB® Toolbox has a rich collection of functions for manipulating and converting between datatypes such as vectors, rotation matrices, unit-quaternions, quaternions, homogeneous transformations and twists which are necessary to represent position and orientation in 2- and 3-dimensions. These are useful in the study of robotics and computer vision, but also for other fields of engineering and physics.

The Toolbox makes strong use of classes to represent many of the mathematical objects and also includes Simulink® blocks for some conversions. The code is written in a straightforward manner which allows for easy understanding, perhaps at the expense of computational efficiency. If you feel strongly about computational efficiency then you can always rewrite the function to be more efficient, compile the M-file using the MATLAB compiler, or create a MEX version.

The bulk of this manual is auto-generated from the comments in the MATLAB code itself. For elaboration on the underlying principles, extensive illustrations and worked examples please consult "*Robotics, Vision & Control*" which provides a detailed discussion (720 pages, nearly 500 figures and over 1000 code examples) of how to use the Toolbox functions to solve many types of problems in robotics. This version corresponds to the **second edition** of the book "*Robotics, Vision & Control*" published in June 2017 – aka RVC2.

Contents

Chapter 1

Introduction

As already mentioned this code has been refactored from the Robotics Toolbox for MATLAB. As that Toolbox evolved there has been increasing adoption of classes, even for objects like rotation matrices and homogeneous transformation matrices which can be represented easily using native MATLAB matrices. The motivations for this are:

1. Classes ensure type safety. For example a 3x3 matrix could be an SO(3) rotation matrix or an SE(2) homogeneous transformation, or the transpose of an SE(3) homogeneous transformation is invalid. Overloaded class operators ensure that only valid operations can be performed.

2. The classes support more descriptive constructors with names like `SO3.eul` which constructs an SO(3) object from Euler angles.

3. A sequence, or trajectory, using native matrices, has to represented by a 3-dimensional matrix, eg. $4 \times 4 \times N$. Using objects we can represent this instead using a 1-dimensional vector of objects.

In RTB10 a set of classes have been introduced to represent orientation and pose in 2D and 3D: `SO2`, `SE2`, `SO3`, `SE3`, `Twist` and `UnitQuaternion`. These classes are fairly polymorphic, that is, they share many methods and operators[1]. All have a number of static methods that serve as constructors from particular representations. A trajectory is represented by a vector of these objects which makes code easier to read and understand. Overloaded operators are used so the classes behave in a similar way to native matrices[2]. The relationship between the classical Toolbox functions and the new classes are shown in Fig 1.1.

You can continue to use the classical functions. The new classes have methods with the names of classical functions to provide similar functionality. For instance

```
>> T = transl(1,2,3);  % create a 4x4 matrix
>> trprint(T)  % invoke the function trprint
>> T = SE3(1,2,3);  % create an SE3 object
>> trprint(T)  % invoke the method trprint
```

[1]For example, you could substitute objects of class `SO3` and `UnitQuaternion` with minimal code change.

[2]The capability is extended so that we can element-wise multiple two vectors of transforms, multiply one transform over a vector of transforms or a set of points.

```
>> T.T   % the equivalent 4x4 matrix
>> double(T) % the equivalent 4x4 matrix

>> T = SE3(1,2,3);  % create a pure translation SE3 object
>> T2 = T*T;  % the result is an SE3 object
>> T3 = trinterp(T, T2,, 5); % create a vector of five SE3 objects between T and T2
>> T3(1)  % the first element of the vector
>> T3*T  % each element of T3 multiplies T, giving a vector of five SE3 objects
```

Options to RTB functions can now be strings[3] or character arrays, ie. `rotx(45, 'deg')` or `rotx(45, "deg")`.

1.1 Installing the Toolbox

1.1.1 Automatically from GitHub

From MATLAB Desktop or Online use the AddOn Manager on the Home tab, and search for "spatial math" and click on the Spatial Math Toolbox. It will be installed into the folder `MATLAB/Add-Ons/Collections/Spatial Math Toolbox/petercorke-spatial-math-xxxx` in your default MATLAB documents folder[4].

This also works from MATLAB Online in which case it will be stored in `/MATLAB Add-Ons/Collections/Spatial Math Toolbox/petercorke-spatial-math-xxxx`.

The Toolbox will be automatically added to the end of your path. If you have the Phase Array Toolbox also installed then note that some of the Spatial Math functions will be shadowed. To check for this run

```
>> which rotx
```

If this indicates a path not as shown above then either:

1. use `pathtool` to move Phase Array Toolbox to the end of the path
2. remove the Phase Array Toolbox, if you don't need it, using the AddOn Manager.

1.1.2 Manually from GitHub

Clone the repository to your own computer

```
>> git clone https://github.com/petercorke/spatial-math
```

and ensure that the folder `spatial-math` is added to your MATLAB path.

1.1.3 Notes on implementation and versions

The Simulink blocks are implemented in Simulink itself with calls to MATLAB code, or as Level-1 S-functions (a proscribed coding format which MATLAB functions to interface with the Simulink simulation engine).

Simulink allows signals to have matrix values but not (yet) object values. Transformations must be represented as matrices, as per the classic functions, not classes. Very old versions of Simulink (prior to version 4) could only handle scalar signals which limited its usefulness for robotics.

1.1.4 Documentation

This document `spatialmath.pdf` is a comprehensive manual that describes all functions in the Toolbox. It is auto-generated from the comments in the MATLAB code and is fully hyperlinked: to external web sites, the table of content to functions, and the "See also" functions to each other.

[3]Introduced from MATLAB 2016b.

[4]`xxxx` is part of git's hash and represents the version number.

Figure 1.1: (top) new and classic methods for representing orientation and pose, (bottom) functions and methods to convert between representations. Reproduced from "*Robotics, Vision & Control, second edition, 2017*"

1.2 Compatible MATLAB versions

The Toolbox has been tested under R2018b and R2019aPRE. Compatibility problems are increasingly likely the older your version of MATLAB is.

1.3 Use in research

If the Toolbox helps you in your endeavours then I'd appreciate you citing the Toolbox when you publish. The details are:

```
@book{Corke17a,
    Author = {Peter I. Corke},
    Note = {ISBN 978-3-319-54413-7},
    Edition = {Second},
    Publisher = {Springer},
    Title = {Robotics, Vision \& Control: Fundamental Algorithms in {MATLAB}},
    Year = {2017}}
```

or

> P.I. Corke, Robotics, Vision & Control: Fundamental Algorithms in MATLAB. Second edition. Springer, 2017. ISBN 978-3-319-54413-7.

which is also given in electronic form in the CITATION file.

1.3.1 Octave

GNU Octave (www.octave.org) is an impressive piece of free software that implements a language that is close to, but not the same as, MATLAB. The Toolboxes currently do not work well with Octave, though as time goes by compatibility improves. Many Toolbox functions work just fine under Octave, but most classes do not.

For uptodate information about running the Toolbox with Octave check out the page `http://petercorke.com/wordpress/toolboxes/other-languages`.

1.4 Support

There is no support! This software is made freely available in the hope that you find it useful in solving whatever problems you have to hand. I am happy to correspond with people who have found genuine bugs or deficiencies but my response time can be long and I can't guarantee that I respond to your email.

I can guarantee that I will not respond to any requests for help with assignments or homework, no matter how urgent or important they might be to you. That's what your teachers, tutors, lecturers and professors are paid to do.

You might instead like to communicate with other users via the Google Group called "Robotics and Machine Vision Toolbox"

> `http://tiny.cc/rvcforum`

which is a forum for discussion. You need to signup in order to post, and the signup process is moderated by me so allow a few days for this to happen. I need you to write a few words about why you want to join the list so I can distinguish you from a spammer or a web-bot.

1.5 Contributing to the Toolboxes

I am very happy to accept contributions for inclusion in future versions of the toolbox. You will, of course, be suitably acknowledged.

Chapter 2

Functions and classes

about

Compact display of variable type

ABOUT(X) displays a compact line that describes the class and dimensions of X.

ABOUT X as above but this is the command rather than functional form.

Examples

```
>> a=1;
>> about a
a [double] : 1x1 (8 bytes)

>> a = rand(5,7);
>> about a
a [double] : 5x7 (280 bytes)
```

See also

whos

---

angdiff

Difference of two angles

ANGDIFF(TH1, TH2) is the difference between angles TH1 and TH2, ie. TH1-TH2 on the circle. The result is in the interval [-π π). Either or both arguments can be a vector:

- If TH1 is a vector, and TH2 a scalar then return a vector where TH2 is modulo subtracted from the corresponding elements of TH1.
- If TH1 is a scalar, and TH2 a vector then return a vector where the corresponding elements of TH2 are modulo subtracted from TH1.

- If `TH1` and `TH2` are vectors then return a vector whose elements are the modulo difference of the corresponding elements of `TH1` and `TH2`, which must be the
- same length.

`ANGDIFF(TH)` as above but `TH=[TH1 TH2]`.

`ANGDIFF(TH)` is the equivalent angle to the scalar `TH` in the interval [-π π).

Notes

- The MathWorks Robotics Systems Toolbox defines a function with the same name which computes `TH2-TH1` rather than `TH1-TH2`.
- If `TH1` and `TH2` are both vectors they should have the same orientation, which the output will assume.

---

angvec2r

Convert angle and vector orientation to a rotation matrix

`R = ANGVEC2R(THETA, V)` is an orthonormal rotation matrix (3×3) equivalent to a rotation of `THETA` about the vector `V`.

Notes

- Uses Rodrigues'formula
- If `THETA` == 0 then return identity matrix and ignore `V`.
- If `THETA` $\neq$ 0 then `V` must have a finite length.

See also

angvec2tr, eul2r, rpy2r, tr2angvec, trexp, SO3.angvec

---

angvec2tr

Convert angle and vector orientation to a homogeneous transform

`T = ANGVEC2TR(THETA, V)` is a homogeneous transform matrix (4×4) equivalent to a rotation of `THETA` about the vector `V`.

Note

- Uses Rodrigues'formula
- The translational part is zero.
- If `THETA` == 0 then return identity matrix and ignore `V`.
- If `THETA` $\neq$ 0 then `V` must have a finite length.

See also

angvec2r, eul2tr, rpy2tr, angvec2r, tr2angvec, trexp, SO3.angvec

Animate

Create an animation

Helper class for creating animations as MP4, animated GIF or a folder of images.

Example

```
anim = Animate('movie.mp4');
 for i=1:100
     plot(...);
     anim.add();
 end
 anim.close();
```

will save the frames in an MP4 movie file using VideoWriter.

Alternatively, to createa of images in PNG format frames named 0000.png, 0001.png and so on in a folder called 'frames'

```
anim = Animate('frames');
for i=1:100
    plot(...);
    anim.add();
end
anim.close();
```

To convert the image files to a movie you could use a tool like ffmpeg

```
ffmpeg -r 10 -i frames/%04d.png out.mp4
```

Notes

- MP4 movies cannot be generated under Linux, a limitation of MATLAB VideoWriter.

Animate.Animate

Create an animation class

`ANIM = ANIMATE(NAME, OPTIONS)` initializes an animation, and creates a movie file or a folder holding individual frames.

`ANIM` = ANIMATE({`NAME`, `OPTIONS`}) as above but arguments are passed as a cell array, which allows a single argument to a higher-level option like 'movie',M to express options as well as filename.

Options

'resolution',R	Set the resolution of the saved image to R pixels per inch.
'profile',P	See VideoWriter for details
'fps',F	Frame rate (default 30)
'bgcolor',C color name.	Set background color of axes, 3 vector or MATLAB

'inner'	inner frame of axes; no axes, labels, ticks.

A profile can also be set by the file extension given:

none 0000.png, 0001.png and so on	Create a folder full of frames in PNG format frames named
.gif	Create animated GIF
.mp4	Create MP4 movie (not on Linux)
.avi	Create AVI movie
.mj2	Create motion jpeg file

Notes

- MP4 movies cannot be generated under Linux, a limitation of MATLAB VideoWriter.
- if no extension or profile is given a folder full of frames is created.
- if a profile is given a movie is created, see VideoWriter for allowable profiles.
- if the file has an extension it specifies the profile.
- if an extension of '.gif'is given an animated GIF is created
- if `NAME` is [] then an Animation object is created but the add() and close() methods do nothing.

See also

VideoWriter

Animate.add

Adds current plot to the animation

`A.ADD()` adds the current figure to the animation.

`A.ADD(FIG)` as above but captures the figure `FIG`.

Notes

- the frame is added to the output file or as a new sequentially numbered image in a folder.
- if the filename was given as [] in the constructor then no action is taken.

See also

print

Animate.close

Closes the animation

`A.CLOSE()` ends the animation process and closes any output file.

Notes

- if the filename was given as [] in the constructor then no action is taken.

circle

Compute points on a circle

`CIRCLE(C, R, OPTIONS)` plots a circle centred at `C` (1×2) with radius `R` on the current axes.

`X = CIRCLE(C, R, OPTIONS)` is a matrix $(2 \times N)$ whose columns define the coordinates [x,y] of points around the circumference of a circle centred at `C` (1×2) and of radius `R`.

`C` is normally 2×1 but if 3×1 then the circle is embedded in 3D, and `X` is $N \times 3$. The circle is always in the xy-plane with a z-coordinate of `C`(3).

Options

'n',N	Specify the number of points (default 50)

---

colnorm

Column-wise norm of a matrix

`CN = COLNORM(A)` is a vector $(1 \times M)$ comprising the Euclidean norm of each column of the matrix `A` $(N \times M)$.

See also

norm

---

delta2tr

Convert differential motion to SE(3) homogeneous transform

`T = DELTA2TR(D)` is a homogeneous transform (4×4) representing differential motion `D` (6×1).

The vector `D`=(dx, dy, dz, dRx, dRy, dRz) represents infinitessimal translation and rotation, and is an approximation to the instantaneous spatial velocity multiplied by time step.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p67.

See also

tr2delta, SE3.delta

---

e2h

Euclidean to homogeneous

`H = E2H(E)` is the homogeneous version ($K+1 \times N$) of the Euclidean points `E` ($K \times N$) where each column represents one point in $\mathbb{R}^K$.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p604.

See also

h2e

eul2jac

Euler angle rate Jacobian

`J = EUL2JAC(PHI, THETA, PSI)` is a Jacobian matrix (3×3) that maps ZYZ Euler angle rates to angular velocity at the operating point specified by the Euler angles `PHI`, `THETA`, `PSI`.

`J = EUL2JAC(EUL)` as above but the Euler angles are passed as a vector `EUL`=[`PHI`, `THETA`, `PSI`].

Notes

- Used in the creation of an analytical Jacobian.
- Angles in radians, rates in radians/sec.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p232-3.

See also

rpy2jac, eul2r, SerialLink.jacobe

eul2r

Convert Euler angles to rotation matrix

`R = EUL2R(PHI, THETA, PSI, OPTIONS)` is an SO(3) orthonornal rotation matrix (3×3) equivalent to the specified Euler angles. These correspond to rotations about the Z, Y, Z axes respectively. If `PHI`, `THETA`, `PSI` are column vectors ($N \times 1$) then they are assumed to represent a trajectory and `R` is a three-dimensional matrix ($3 \times 3 \times N$), where the last index corresponds to rows of `PHI`, `THETA`, `PSI`.

`R = EUL2R(EUL, OPTIONS)` as above but the Euler angles are taken from the vector (1×3) `EUL` = [`PHI THETA PSI`]. If `EUL` is a matrix ($N \times 3$) then `R` is a three-dimensional matrix ($3 \times 3 \times N$), where the last index corresponds to rows of RPY which are assumed to be [`PHI`,`THETA`,`PSI`].

Options

'deg'	Angles given in degrees (radians default)

Note

- The vectors `PHI`, `THETA`, `PSI` must be of the same length.

See also

eul2tr, rpy2tr, tr2eul, SO3.eul

---

eul2tr

Convert Euler angles to homogeneous transform

`T = EUL2TR(PHI, THETA, PSI, OPTIONS)` is an SE(3) homogeneous transformation matrix (4×4) with zero translation and rotation equivalent to the specified Euler angles. These correspond to rotations about the Z, Y, Z axes respectively. If `PHI`, `THETA`, `PSI` are column vectors ($N \times 1$) then they are assumed to represent a trajectory and R is a three-dimensional matrix ($4 \times 4 \times N$), where the last index corresponds to rows of `PHI`, `THETA`, `PSI`.

`R = EUL2R(EUL, OPTIONS)` as above but the Euler angles are taken from the vector (1×3) `EUL` = [`PHI THETA PSI`]. If `EUL` is a matrix ($N \times 3$) then `R` is a three-dimensional matrix ($4 \times 4 \times N$), where the last index corresponds to rows of RPY which are assumed to be [`PHI`,`THETA`,`PSI`].

Options

'deg'	Angles given in degrees (radians default)

Note

- The vectors `PHI`, `THETA`, `PSI` must be of the same length.
- The translational part is zero.

See also

eul2r, rpy2tr, tr2eul, SE3.eul

---

h2e

Homogeneous to Euclidean

`E = H2E(H)` is the Euclidean version ($K-1 \times N$) of the homogeneous points `H` ($K \times N$) where each column represents one point in $\mathbb{P}^K$.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p604.

See also

e2h

---

homline

Homogeneous line from two points

`L = HOMLINE(X1, Y1, X2, Y2)` is a vector (3×1) which describes a line in homogeneous form that contains the two Euclidean points (`X1`,`Y1`) and (`X2`,`Y2`).

Homogeneous points X (3×1) on the line must satisfy `L'`*X = 0.

See also

plot_homline

homtrans

Apply a homogeneous transformation

`P2 = HOMTRANS(T, P)` applies the homogeneous transformation `T` to the points stored columnwise in `P`.

- If `T` is in SE(2) (3×3) and
  - `P` is $2 \times N$ (2D points) they are considered Euclidean ($\mathbb{R}^2$)
  - `P` is $3 \times N$ (2D points) they are considered projective ($\mathbb{P}^2$)
- If `T` is in SE(3) (4×4) and
  - `P` is $3 \times N$ (3D points) they are considered Euclidean ($\mathbb{R}^3$)
  - `P` is $4 \times N$ (3D points) they are considered projective ($\mathbb{P}^3$)

`P2` and `P` have the same number of rows, ie. if Euclidean points are given then Euclidean points are returned, if projective points are given then projective points are returned.

`TP = HOMTRANS(T, T1)` applies homogeneous transformation `T` to the homogeneous transformation `T1`, that is `TP=T*T1`. If `T1` is a 3-dimensional transformation then `T` is applied to each plane as defined by the first two dimensions, ie. if `T` is $N \times N$ and `T1` is $N \times N \times M$ then the result is $N \times N \times M$.

Notes

- If `T` is a homogeneous transformation defining the pose of $\{B\}$ with respect to $\{A\}$, then the points are defined with respect to frame $\{B\}$ and are transformed to be
- with respect to frame $\{A\}$.

See also

e2h, h2e, RTBPose.mtimes

ishomog

Test if SE(3) homogeneous transformation matrix

`ISHOMOG(T)` is true (1) if the argument `T` is of dimension 4×4 or $4 \times 4 \times N$, else false (0).

`ISHOMOG(T, 'check')` as above, but also checks the validity of the rotation sub-matrix.

Notes

- A valid rotation sub-matrix has determinant of 1.
- The first form is a fast, but incomplete, test for a transform is SE(3).

See also

isrot, ishomog2, isvec

---

ishomog2

Test if SE(2) homogeneous transformation matrix

`ISHOMOG2(T)` is true (1) if the argument `T` is of dimension 3×3 or $3 \times 3 \times N$, else false (0).

`ISHOMOG2(T, 'check')` as above, but also checks the validity of the rotation sub-matrix.

Notes

- A valid rotation sub-matrix has determinant of 1.
- The first form is a fast, but incomplete, test for a transform in SE(3).

See also

ishomog, isrot2, isvec

---

isrot

Test if SO(3) rotation matrix

`ISROT(R)` is true (1) if the argument is of dimension 3×3 or $3 \times 3 \times N$, else false (0).

`ISROT(R, 'check')` as above, but also checks the validity of the rotation matrix.

Notes

- A valid rotation matrix has determinant of 1.

See also

ishomog, isrot2, isvec

---

isrot2

Test if SO(2) rotation matrix

`ISROT2(R)` is true (1) if the argument is of dimension 2×2 or $2 \times 2 \times N$, else false (0).

`ISROT2(R, 'check')` as above, but also checks the validity of the rotation matrix.

Notes

- A valid rotation matrix has determinant of 1.

See also

isrot, ishomog2, isvec

---

isunit

Test if vector has unit length

`ISUNIT(V)` is true if the vector has unit length.

Notes

- A tolerance of 100eps is used.

---

isvec

Test if vector

`ISVEC(V)` is true (1) if the argument `V` is a 3-vector, either a row- or column-vector. Otherwise false (0).

`ISVEC(V, L)` is true (1) if the argument `V` is a vector of length `L`, either a row- or column-vector. Otherwise false (0).

Notes

- Differs from MATLAB builtin function ISVECTOR which returns true for the case of a scalar, ISVEC does not.
- Gives same result for row- or column-vector, ie. 3×1 or 1×3 gives true.
- Works for a symbolic math symfun.

See also

ishomog, isrot

---

lift23

Lift SE(2) transform to SE(3)

`T3 = SE3(T2)` returns a homogeneous transform (4×4) that represents the same X,Y translation and Z rotation as does `T2` (3×3).

See also

SE2, SE2.SE3, transl, rotx

---

numcols

Number of columns in matrix

`NC = NUMCOLS(M)` is the number of columns in the matrix `M`.

Notes

- Readable shorthand for SIZE(`M`,2);

See also

numrows, size

---

numrows

Number of rows in matrix

`NR = NUMROWS(M)` is the number of rows in the matrix `M`.

Notes

- Readable shorthand for SIZE(`M`,1);

See also

numcols, size

---

oa2r

Convert orientation and approach vectors to rotation matrix

`R = OA2R(O, A)` is an SO(3) rotation matrix (3×3) for the specified orientation and approach vectors (3×1) formed from 3 vectors such that `R` = [N `O` `A`] and N = `O` x `A`.

Notes

- The matrix is guaranteed to be orthonormal so long as `O` and `A` are not parallel.
- The vectors `O` and `A` are parallel to the Y- and Z-axes of the coordinate frame respectively.

References

- Robot manipulators: mathematics, programming and control Richard Paul, MIT Press, 1981.

See also

rpy2r, eul2r, oa2tr, SO3.oa

oa2tr

Convert orientation and approach vectors to homogeneous transformation

`T = OA2TR(O, A)` is an SE(3) homogeneous tranformation (4×4) for the specified orientation and approach vectors (3×1) formed from 3 vectors such that R = [N `O` `A`] and N = `O` x `A`.

Notes

- The rotation submatrix is guaranteed to be orthonormal so long as `O` and `A` are not parallel.
- The vectors `O` and `A` are parallel to the Y- and Z-axes of the coordinate frame respectively.
- The translational part is zero.

References

- Robot manipulators: mathematics, programming and control Richard Paul, MIT Press, 1981.

See also

rpy2tr, eul2tr, oa2r, SE3.oa

PGraph

Graph class

g = PGraph()	create a 2D, planar embedded, directed graph
g = PGraph(n)	create an n-d, embedded, directed graph

Provides support for graphs that:

- are directed
- are embedded in a coordinate system (2D or 3D)
- have multiple unconnected components
- have symmetric cost edges (A to B is same cost as B to A)
- have no loops (edges from A to A)

Graph representation:

- vertices are represented by integer vertex ids (vid)
- edges are represented by integer edge ids (eid)
- each vertex can have arbitrary associated data
- each edge can have arbitrary associated data

Methods

Constructing the graph

g.add_node(coord)	add vertex
g.add_edge(v1, v2)	add edge fbetween vertices
g.setcost(e, c)	set cost for edge
g.setedata(e, u)	set user data for edge
g.setvdata(v, u)	set user data for vertex

Modifying the graph

g.clear()	remove all vertices and edges from the graph
g.delete_edge(e)	remove edge
g.delete_node(v)	remove vertex
g.setcoord(v)	set coordinate of vertex

Information from graph

g.about()	summary information about node
g.component(v)	component id for vertex
g.componentnodes(c)	vertices in component
g.connectivity()	number of edges for all vertices
g.connectivity_in()	number of incoming edges for all vertices
g.connectivity_out()	number of outgoing edges for all vertices
g.coord(v)	coordinate of vertex
g.cost(e)	cost of edge
g.distance_metric(v1,v2)	distance between nodes
g.edata(e)	get edge user data
g.edgedir(v1,v2)	direction of edge
g.edges(v)	list of edges for vertex
g.edges_in(v)	list of edges into vertex
g.edges_out(v)	list of edges from vertex
g.lookup(name)	vertex from name
g.name(v)	name of vertex
g.neighbours(v)	neighbours of vertex
g.neighbours_d(v)	neighbours of vertex and edge directions
g.neighbours_in(v)	neighbours with edges in
g.neighbours_out(v)	neighbours with edges out
g.samecomponent(v1,v2)	test if vertices in same component
g.vdata(v)	vertex user data
g.vertices(e)	vertices for edge

Display

g.char()	convert graph to string
g.display()	display summary of graph
g.highlight_node(v)	highlight vertex

g.highlight_edge(e)	highlight edge
g.highlight_component(c)	highlight all nodes in component
g.highlight_path(p)	highlight nodes and edge along path
g.pick(coord)	vertex closest to coord
g.plot()	plot graph

Matrix representations

g.adjacency()	adjacency matrix
g.degree()	degree matrix
g.incidence()	incidence matrix
g.laplacian()	Laplacian matrix

Planning paths through the graph

g.Astar(s, g)	shortest path from s to g
g.goal(v)	set goal vertex, and plan paths
g.path(v)	list of vertices from v to goal

Graph and world points

g.closest(coord)	vertex closest to coord
g.coord(v)	coordinate of vertex v
g.distance(v1, v2)	distance between v1 and v2
g.distances(coord)	return sorted distances from coord to all vertices

Object properties (read only)

g.n	number of vertices
g.ne	number of edges
g.nc	number of components

Example

```
g = PGraph();
g.add_node([1 2]');  % add node 1
g.add_node([3 4]');  % add node 1
g.add_node([1 3]');  % add node 1
g.add_edge(1, 2);    % add edge 1-2
g.add_edge(2, 3);    % add edge 2-3
g.add_edge(1, 3);    % add edge 1-3
g.plot()
```

Notes

- Support for edge direction is quite simple.
- The method distance_metric() could be redefined in a subclass.

PGraph.PGraph

Graph class constructor

`G=PGraph(D, OPTIONS)` is a graph object embedded in `D` dimensions.

Options

'distance',M 'Euclidean'(default) or 'SE2'.	Use the distance metric M for path planning which is either
'verbose'	Specify verbose operation

Notes

- Number of dimensions is not limited to 2 or 3.
- The distance metric 'SE2'is the sum of the squares of the difference in position and angle modulo 2π.
- To use a different distance metric create a subclass of PGraph and override the method distance_metric().

PGraph.add_edge

Add an edge

`E = G.add_edge(V1, V2)` adds a directed edge from vertex id `V1` to vertex id `V2`, and returns the edge id `E`. The edge cost is the distance between the vertices.

`E = G.add_edge(V1, V2, C)` as above but the edge cost is `C`.

Notes

- If `V2` is a vector add edges from `V1` to all elements of `V2`
- Distance is computed according to the metric specified in the constructor.

See also

PGraph.add_node, PGraph.edgedir

PGraph.add_node

Add a node

`V = G.add_node(X, OPTIONS)` adds a node/vertex with coordinate `X` ($D \times 1$) and returns the integer node id `V`.

Options:

'name',N	Assign a string name N to this vertex
'from',V	Create a directed edge from vertex `V` with cost equal to the distance between the vertices.
'cost',C	If an edge is created use cost C

Notes

- Distance is computed according to the metric specified in the constructor.

See also

PGraph.add_edge, PGraph.data, PGraph.getdata

PGraph.adjacency

Adjacency matrix of graph

`A = G.adjacency()` is a matrix ($N \times N$) where element `A`(i,j) is the cost of moving from vertex i to vertex j.

Notes

- Matrix is symmetric.

- Eigenvalues of `A` are real and are known as the spectrum of the graph.
- The element `A`(I,J) can be considered the number of walks of one edge from vertex I to vertex J (either zero or one). The element (I,J)
- of $\mathtt{A}^N$ are the number of walks of length N from vertex I to vertex J.

See also

PGraph.degree, PGraph.incidence, PGraph.laplacian

PGraph.Astar

path finding

`PATH = G.Astar(V1, V2)` is the lowest cost path from vertex `V1` to vertex `V2`. `PATH` is a list of vertices starting with `V1` and ending `V2`.

`[PATH,C] = G.Astar(V1, V2)` as above but also returns the total cost of traversing `PATH`.

Notes

- Uses the efficient A* search algorithm.
- The heuristic is the distance function selected in the constructor, it must be admissible, meaning that it never overestimates the actual
- cost to get to the nearest goal node.

References

- Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". Hart, P. E.; Nilsson, N. J.; Raphael, B.
- SIGART Newsletter 37: 28-29, 1972.

See also

PGraph.goal, PGraph.path

PGraph.char

Convert graph to string

`S = G.char()` is a compact human readable representation of the state of the graph including the number of vertices, edges and components.

PGraph.clear

Clear the graph

`G.clear()` removes all vertices, edges and components.

PGraph.closest

Find closest vertex

`V = G.closest(X)` is the vertex geometrically closest to coordinate `X`.

`[V,D] = G.closest(X)` as above but also returns the distance `D`.

See also

PGraph.distances

PGraph.component

Graph component

`C = G.component(V)` is the id of the graph component that contains vertex `V`.

PGraph.componentnodes

Graph component

`C = G.component(V)` are the ids of all vertices in the graph component `V`.

PGraph.connectivity

Node connectivity

`C = G.connectivity()` is a vector ($N \times 1$) with the number of edges per vertex.

The average vertex connectivity is

```
mean(g.connectivity())
```

and the minimum vertex connectivity is

```
min(g.connectivity())
```

PGraph.connectivity_in

Graph connectivity

`C = G.connectivity()` is a vector ($N \times 1$) with the number of incoming edges per vertex.

The average vertex connectivity is

```
mean(g.connectivity())
```

and the minimum vertex connectivity is

```
min(g.connectivity())
```

PGraph.connectivity_out

Graph connectivity

`C = G.connectivity()` is a vector ($N \times 1$) with the number of outgoing edges per vertex.

The average vertex connectivity is

```
mean(g.connectivity())
```

and the minimum vertex connectivity is

```
min(g.connectivity())
```

PGraph.coord

Coordinate of node

`X = G.coord(V)` is the coordinate vector ($D \times 1$) of vertex id `V`.

PGraph.cost

Cost of edge

`C = G.cost(E)` is the cost of edge id `E`.

PGraph.degree

Degree matrix of graph

`D = G.degree()` is a diagonal matrix ($N \times N$) where element `D`(i,i) is the number of edges connected to vertex id i.

See also

PGraph.adjacency, PGraph.incidence, PGraph.laplacian

PGraph.display

Display graph

`G.display()` displays a compact human readable representation of the state of the graph including the number of vertices, edges and components.

See also

PGraph.char

PGraph.distance

Distance between vertices

`D = G.distance(V1, V2)` is the geometric distance between the vertices `V1` and `V2`.

See also

PGraph.distances

PGraph.distances

Distances from point to vertices

`D = G.distances(X)` is a vector ($1 \times N$) of geometric distance from the point `X` (`D` $\times 1$) to every other vertex sorted into increasing order.

`[D,W] = G.distances(P)` as above but also returns `W` ($1 \times N$) with the corresponding vertex id.

Notes

- Distance is computed according to the metric specified in the constructor.

See also

PGraph.closest

PGraph.dotfile

Create a GraphViz dot file

`G.dotfile(filename, OPTIONS)` creates the specified file which contains the GraphViz code to represent the embedded graph.

`G.dotfile(OPTIONS)` as above but outputs the code to the console.

Options

'directed'	create a directed graph

Notes

- An undirected graph is default
- Use neato rather than dot to get the embedded layout

PGraph.edata

Get user data for node

`U = G.data(V)` gets the user data of vertex `V` which can be of any type such as a number, struct, object or cell array.

See also

PGraph.setdata

PGraph.edgedir

Find edge direction

`D = G.edgedir(V1, V2)` is the direction of the edge from vertex id `V1` to vertex id `V2`.

If we add an edge from vertex 3 to vertex 4

```
g.add_edge(3, 4)
```

then

```
g.edgedir(3, 4)
```

is positive, and

```
g.edgedir(4, 3)
```

is negative.

See also

PGraph.add_node, PGraph.add_edge

PGraph.edges

Find edges given vertex

`E = G.edges(V)` is a vector containing the id of all edges connected to vertex id `V`.

See also

PGraph.edgedir

PGraph.edges_in

Find edges given vertex

`E = G.edges(V)` is a vector containing the id of all edges connected to vertex id `V`.

See also

PGraph.edgedir

PGraph.edges_out

Find edges given vertex

`E = G.edges(V)` is a vector containing the id of all edges connected to vertex id `V`.

See also

PGraph.edgedir

PGraph.get.n

Number of vertices

`G.n` is the number of vertices in the graph.

See also

PGraph.ne

PGraph.get.nc

Number of components

`G.nc` is the number of components in the graph.

See also

PGraph.component

PGraph.get.ne

Number of edges

`G.ne` is the number of edges in the graph.

See also

PGraph.n

PGraph.graphcolor

the graph

PGraph.highlight_component

Highlight a graph component

`G.highlight_component(C, OPTIONS)` highlights the vertices that belong to graph component `C`.

Options

'NodeSize',S	Size of vertex circle (default 12)
'NodeFaceColor',C	Node circle color (default yellow)
'NodeEdgeColor',C	Node circle edge color (default blue)

See also

PGraph.highlight_node, PGraph.highlight_edge, PGraph.highlight_component

---

PGraph.highlight_edge

Highlight a node

`G.highlight_edge(V1, V2)` highlights the edge between vertices `V1` and `V2`.

`G.highlight_edge(E)` highlights the edge with id `E`.

Options

'EdgeColor',C	Edge edge color (default black)
'EdgeThickness',T	Edge thickness (default 1.5)

See also

PGraph.highlight_node, PGraph.highlight_path, PGraph.highlight_component

PGraph.highlight_node

Highlight a node

`G.highlight_node(V, OPTIONS)` highlights the vertex `V` with a yellow marker. If `V` is a list of vertices then all are highlighted.

Options

'NodeSize',S	Size of vertex circle (default 12)
'NodeFaceColor',C	Node circle color (default yellow)
'NodeEdgeColor',C	Node circle edge color (default blue)

See also

PGraph.highlight_edge, PGraph.highlight_path, PGraph.highlight_component

PGraph.highlight_path

Highlight path

`G.highlight_path(P, OPTIONS)` highlights the path defined by vector `P` which is a list of vertex ids comprising the path.

Options

'NodeSize',S	Size of vertex circle (default 12)
'NodeFaceColor',C	Node circle color (default yellow)
'NodeEdgeColor',C	Node circle edge color (default blue)
'EdgeColor',C	Node circle edge color (default black)
'EdgeThickness',T	Edge thickness (default 1.5)

See also

PGraph.highlight_node, PGraph.highlight_edge, PGraph.highlight_component

PGraph.incidence

Incidence matrix of graph

`IN = G.incidence()` is a matrix ($N \times NE$) where element `IN`(i,j) is non-zero if vertex id i is connected to edge id j.

See also

PGraph.adjacency, PGraph.degree, PGraph.laplacian

PGraph.laplacian

Laplacian matrix of graph

`L = G.laplacian()` is the Laplacian matrix ($N \times N$) of the graph.

Notes

- `L` is always positive-semidefinite.
- `L` has at least one zero eigenvalue.
- The number of zero eigenvalues is the number of connected components in the graph.

See also

PGraph.adjacency, PGraph.incidence, PGraph.degree

PGraph.name

Name of node

`X = G.name(V)` is the name (string) of vertex id `V`.

PGraph.neighbours

Neighbours of a vertex

`N = G.neighbours(V)` is a vector of ids for all vertices which are directly connected neighbours of vertex `V`.

`[N,C] = G.neighbours(V)` as above but also returns a vector `C` whose elements are the edge costs of the paths corresponding to the vertex ids in `N`.

PGraph.neighbours_d

Directed neighbours of a vertex

`N = G.neighbours_d(V)` is a vector of ids for all vertices which are directly connected neighbours of vertex `V`. Elements are positive if there is a link from `V` to the node (outgoing), and negative if the link is from the node to `V` (incoming).

`[N,C] = G.neighbours_d(V)` as above but also returns a vector `C` whose elements are the edge costs of the paths corresponding to the vertex ids in `N`.

PGraph.neighbours_in

Incoming neighbours of a vertex

`N = G.neighbours(V)` is a vector of ids for all vertices which are directly connected neighbours of vertex `V`.

`[N,C] = G.neighbours(V)` as above but also returns a vector `C` whose elements are the edge costs of the paths corresponding to the vertex ids in `N`.

PGraph.neighbours_out

Outgoing neighbours of a vertex

`N = G.neighbours(V)` is a vector of ids for all vertices which are directly connected neighbours of vertex `V`.

`[N,C] = G.neighbours(V)` as above but also returns a vector `C` whose elements are the edge costs of the paths corresponding to the vertex ids in `N`.

PGraph.pick

Graphically select a vertex

`V = G.pick()` is the id of the vertex closest to the point clicked by the user on a plot of the graph.

See also

PGraph.plot

PGraph.plot

Plot the graph

`G.plot(OPT)` plots the graph in the current figure. Nodes are shown as colored circles.

Options

'labels'	Display vertex id (default false)
'edges'	Display edges (default true)
'edgelabels'	Display edge id (default false)
'NodeSize',S	Size of vertex circle (default 8)
'NodeFaceColor',C	Node circle color (default blue)
'NodeEdgeColor',C	Node circle edge color (default blue)
'NodeLabelSize',S	Node label text sizer (default 16)
'NodeLabelColor',C	Node label text color (default blue)
'EdgeColor',C	Edge color (default black)
'EdgeLabelSize',S	Edge label text size (default black)
'EdgeLabelColor',C	Edge label text color (default black)
'componentcolor'	Node color is a function of graph component
'only',N	Only show these nodes

PGraph.samecomponent

Graph component

`C = G.component(V)` is the id of the graph component that contains vertex `V`.

PGraph.setcoord

Coordinate of node

`X = G.coord(V)` is the coordinate vector ($D \times 1$) of vertex id `V`.

PGraph.setcost

Set cost of edge

`G.setcost(E, C)` set cost of edge id `E` to `C`.

PGraph.setedata

Set user data for node

`G.setdata(V, U)` sets the user data of vertex `V` to `U` which can be of any type such as a number, struct, object or cell array.

See also

PGraph.data

PGraph.setvdata

Set user data for node

`G.setdata(V, U)` sets the user data of vertex `V` to `U` which can be of any type such as a number, struct, object or cell array.

See also

PGraph.data

PGraph.vdata

Get user data for node

`U = G.data(V)` gets the user data of vertex `V` which can be of any type such as a number, struct, object or cell array.

See also

PGraph.setdata

PGraph.vertices

Find vertices given edge

`V = G.vertices(E)` return the id of the vertices that define edge `E`.

plot2

Plot trajectories

Convenience function for plotting 2D or 3D trajectory data stored in a matrix with one row per time step.

`PLOT2(P)` plots a line with coordinates taken from successive rows of `P`($N \times 2$ or $N \times 3$).

If `P` has three dimensions, ie. $N \times 2 \times M$ or $N \times 3 \times M$ then the M trajectories are overlaid in the one plot.

`PLOT2(P, LS)` as above but the line style arguments `LS` are passed to plot.

See also

mplot, plot

---

plot_arrow

Draw an arrow in 2D or 3D

`PLOT_ARROW(P1, P2, OPTIONS)` draws an arrow from `P1` to `P2` (2×1 or 3×1). For 3D case the arrow head is a cone.

`PLOT_ARROW(P, OPTIONS)` as above where the columns of `P` (2×2 or 3×2) define the start and end points, ie. `P`=[`P1 P2`].

`H = PLOT_ARROW(...)` as above but returns the graphics handle of the arrow.

Options

- All options are passed through to arrow3.
- MATLAB LineSpec such as 'r'or 'b–'

Example

```
plot_arrow([0 0 0]', [1 2 3]', 'r')  % a red arrow
plot_arrow([0 0 0], [1 2 3], 'r--3', 4) % dashed red arrow big head
```

Notes

- Requires `https://www.mathworks.com/matlabcentral/fileexchange/14056-arrow3`
- ARROW3 attempts to preserve the appearance of existing axes. In particular, ARROW3 will not change XYZLim, View, or CameraViewAngle.

See also

arrow3

---

plot_box

Draw a box

`PLOT_BOX(B, OPTIONS)` draws a box defined by `B`=[XL XR; YL YR] on the current plot with optional MATLAB linestyle options LS.

`PLOT_BOX(X1,Y1, X2,Y2, OPTIONS)` draws a box with corners at (`X1`,`Y1`) and (`X2`,`Y2`), and optional MATLAB linestyle options LS.

`PLOT_BOX('centre', P, 'size', W, OPTIONS)` draws a box with center at `P`=[X,Y] and with dimensions `W`=[WIDTH HEIGHT].

`PLOT_BOX('topleft', P, 'size', W, OPTIONS)` draws a box with top-left at `P`=[X,Y] and with dimensions `W`=[WIDTH HEIGHT].

`PLOT_BOX('matlab', BOX, LS)` draws box(es) as defined using the MATLAB convention of specifying a region in terms of top-left coordinate, width and height. One box is drawn for each row of `BOX` which is [xleft ytop width height].

`H = PLOT_ARROW(...)` as above but returns the graphics handle of the arrow.

Options

'edgecolor'	the color of the circle's edge, MATLAB ColorSpec
'fillcolor'	the color of the circle's interior, MATLAB ColorSpec
'alpha'	transparency of the filled circle: 0=transparent, 1=solid

- For an unfilled box:
  - any standard MATLAB LineSpec such as 'r'or 'b—'.
  - any MATLAB LineProperty options can be given such as 'LineWidth', 2.
- For a filled box any MATLAB PatchProperty options can be given.

Examples

```
plot_box([0 1; 0 2], 'r')   % draw a hollow red box
plot_box([0 1; 0 2], 'fillcolor', 'b', 'alpha', 0.5)   % translucent filled blue box
```

Notes

- The box is added to the current plot irrespective of hold status.

See also

plot_poly, plot_circle, plot_ellipse

plot_circle

Draw a circle

plot_circleC, R, OPTIONS) draws a circle on the current plot with centre C=[X,Y] and radius R. If C=[X,Y,Z] the circle is drawn in the XY-plane at height Z.

If C $(2 \times N)$ then N circles are drawn. If R (1×1) then all circles have the same radius or else R $(1 \times N)$ to specify the radius of each circle.

`H = plot_circle(...)` as above but return handles. For multiple circles `H` is a vector of handles, one per circle.

Options

'edgecolor'	the color of the circle's edge, Matlab color spec
'fillcolor'	the color of the circle's interior, Matlab color spec
'alpha'	transparency of the filled circle: 0=transparent, 1=solid
'alter',H	alter existing circles with handle `H`

- For an unfilled circle:
  - any standard MATLAB LineStyle such as 'r'or 'b—'.
  - any MATLAB LineProperty options can be given such as 'LineWidth', 2.
- For a filled circle any MATLAB PatchProperty options can be given.

Example

```
H = plot_circle([3 4]', 2, 'r');  % draw red circle
plot_circle([3 4]', 3, 'alter', H); % change the circle radius
plot_circle([3 4]', 3, 'alter', H, 'LineColor', 'k'); % change the color
```

Notes

- The 'alter'option can be used to create a smooth animation.
- The circle(s) is added to the current plot irrespective of hold status.

See also

plot_ellipse, plot_box, plot_poly

plot_ellipse

Draw an ellipse or ellipsoid

`plot_ellipse(E, OPTIONS)` draws an ellipse or ellipsoid defined by X'EX = 0 on the current plot, centred at the origin. `E` (2×2) for an ellipse and `E` (2×3) for an ellipsoid.

`plot_ellipse(E, C, OPTIONS)` as above but centred at `C`=[X,Y]. If `C`=[X,Y,Z] the ellipse is parallel to the XY plane but at height Z.

`H = plot_ellipse(...)` as above but return graphic handle.

Options

'confidence',C	confidence interval, range 0 to 1
'alter',H	alter existing ellipses with handle `H`
'npoints',N	use N points to define the ellipse (default 40)
'edgecolor'	color of the ellipse boundary edge, MATLAB color spec
'fillcolor'	the color of the ellipses's interior, MATLAB color spec
'alpha'	transparency of the fillcolored ellipse: 0=transparent, 1=solid
'shadow'	show shadows on the 3 walls of the plot box

- For an unfilled ellipse:
  - any standard MATLAB LineStyle such as 'r'or 'b—'.

  - any MATLAB LineProperty options can be given such as 'LineWidth', 2.

- For a filled ellipse any MATLAB PatchProperty options can be given.

Example

```
H = plot_ellipse(diag([1 2]), [3 4]', 'r'); % draw red ellipse
plot_ellipse(diag([1 2]), [5 6]', 'alter', H); % move the ellipse
plot_ellipse(diag([1 2]), [5 6]', 'alter', H, 'LineColor', 'k'); % change color

plot_ellipse(COVAR, 'confidence', 0.95); % draw 95% confidence ellipse
```

Notes

- The 'alter'option can be used to create a smooth animation.
- If `E` (2×2) draw an ellipse, else if `E` (3×3) draw an ellipsoid.
- The ellipse is added to the current plot irrespective of hold status.
- Shadow option only valid for ellipsoids.
- If a confidence interval is given then `E` is interpretted as a covariance matrix and the ellipse size is computed using an inverse chi-squared function.
- This requires CHI2INV in the Statistics and Machine Learning Toolbox or
- CHI2INV_RTB from the Robotics Toolbox for MATLAB.

See also

plot_ellipse_inv, plot_circle, plot_box, plot_poly, ch2inv

---

plot_homline

Draw a line in homogeneous form

`PLOT_HOMLINE(L)` draws a 2D line in the current plot defined in homogenous form ax + by + c = 0 where `L` (3×1) is `L` = [a b c]. The current axis limits are used to determine the endpoints of the line. If `L` $(3 \times N)$ then N lines are drawn, one per column.

`PLOT_HOMLINE(L, LS)` as above but the MATLAB line specification `LS` is given.

`H = PLOT_HOMLINE(...)` as above but returns a vector of graphics handles for the lines.

Notes

- The line(s) is added to the current plot.
- The line(s) can be drawn in 3D axes but will always lie in the xy-plane.

Example

```
L = homline([1 2]', [3 1]'); % homog line from (1,2) to (3,1)
plot_homline(L, 'k--'); % plot dashed black line
```

See also

plot_box, plot_poly, homline

---

plot_point

Draw a point

`PLOT_POINT(P, OPTIONS)` adds point markers and optional annotation text to the current plot, where `P` ($2 \times N$) and each column is a point coordinate.

`H = PLOT_POINT(...)` as above but return handles to the points.

Options

'textcolor', colspec	Specify color of text
'textsize', size	Specify size of text
'bold'	Text in bold font.
'printf', fmt, data string and corresponding element of data	Label points according to printf format
'sequence'	Label points sequentially
'label',L	Label for point

Additional options to PLOT can be used:

- standard MATLAB LineStyle such as 'r'or 'b—'
- any MATLAB LineProperty options can be given such as 'LineWidth', 2.

Notes

- The point(s) and annotations are added to the current plot.
- Points can be drawn in 3D axes but will always lie in the xy-plane.
- Handles are to the points but not the annotations.

Examples

Simple point plot with two markers

```
P = rand(2,4);
plot_point(P);
```

Plot points with markers

```
plot_point(P, '*');
```

Plot points with solid blue circular markers

```
plot_point(P, 'bo', 'MarkerFaceColor', 'b');
```

Plot points with square markers and labelled 1 to 4

```
plot_point(P, 'sequence', 's');
```

Plot points with circles and labelled P1, P2, P4 and P8

```
data = [1 2 4 8];
plot_point(P, 'printf', {' P%d', data}, 'o');
```

plot_poly

Draw a polygon

`plot_poly(P, OPTIONS)` adds a closed polygon defined by vertices in the columns of `P` ($2 \times N$), in the current plot.

`H = plot_poly(...)` as above but returns a graphics handle.

plot_poly(`H`,)

OPTIONS

'fillcolor',F	the color of the circle's interior, MATLAB color spec
'alpha',A	transparency of the filled circle: 0=transparent, 1=solid.
'edgecolor',E	edge color
'animate'	the polygon can be animated
'tag',T	the polygon is created with a handle graphics tag
'axis',h	handle of axis or UIAxis to draw into (default is current axis)

- For an unfilled polygon:
  - any standard MATLAB LineStyle such as 'r'or 'b—'.
  - any MATLAB LineProperty options can be given such as 'LineWidth', 2.
- For a filled polygon any MATLAB PatchProperty options can be given.

Notes

- If `P` ($3 \times N$) the polygon is drawn in 3D
- If not filled the polygon is a line segment, otherwise it is a patch object.
- The 'animate'option creates an hgtransform object as a parent of the polygon, which can be animated by the last call signature above.
- The graphics are added to the current plot.

Example

```
POLY = [0 1 2; 0 1 0];
H = plot_poly(POLY, 'animate', 'r'); % draw a red polygon

H = plot_poly(POLY, 'animate', 'r'); % draw a red polygon that can be animated
plot_poly(H, transl(2,1,0) );  % transform its vertices by (2,1)
```

See also

plot_box, plot_circle, patch, Polygon

plot_ribbon

Draw a wide curved 3D arrow

`plot_ribbon()` adds a 3D curved arrow “ribbon” to the current plot. The ribbon by default is about the z-axis at the origin.

Options

'radius',R	radius of the ribbon (default 0.25)

'N',N	number of points along the ribbon (default 100)
'd',D	ratio of shaft length to total (default 0.9)
'w1',W	width of shaft (default 0.2)
'w2',W	width of head (default 0.4)
'phi',P	length of ribbon as fraction of circle (default 0.8)
'phase',P	rotate the arrow about its axis (radians, default 0)
'color',C	color as MATLAB ColorSpec (default 'r')
'specular',S	specularity of surface (default 0.2)
'diffuse',D	diffusivity of surface (default 0.8)
'nice'	adjust the phase for nicely phased arrow

The parameters of the ribbon are:

```
 ^
 |                             | \
 |  ^  +-----------------------|  \
 |  |  |                           .
 |  v  +-----------------------|  /
 |  w1                         | /
 v     <----------- d --------->
w2    <----------- phi ----------->
```

Examples

To draw the ribbon at distance A along the X, Y, Z axes is:

```
plot_ribbon2( SE3(A,0,0)*SE3.Ry(pi/2) )
plot_ribbon2( SE3(0, A,0)*SE3.Rx(pi/2) )
plot_ribbon2( SE3(0, 0, A) )
shading interp
camlight
```

See also

plot_arrow, plot

plot_sphere

Draw sphere

`PLOT_SPHERE(C, R, LS)` draws spheres in the current plot. `C` is the centre of the sphere (3×1), `R` is the radius and `LS` is an optional MATLAB ColorSpec, either a letter or a 3-vector.

`PLOT_SPHERE(C, R, COLOR, ALPHA)` as above but `ALPHA` specifies the opacity of the sphere where 0 is transparant and 1 is opaque. The default is 1.

If `C` ($3 \times N$) then N sphhere are drawn and H is $N \times 1$. If `R` (1×1) then all spheres have the same radius or else `R` ($1 \times N$) to specify the radius of each sphere.

`H = PLOT_SPHERE(...)` as above but returns the handle(s) for the spheres.

Notes

- The sphere is always added, irrespective of figure hold state.
- The number of vertices to draw the sphere is hardwired.

Example

```
plot_sphere( mkgrid(2, 1), .2, 'b'); % Create four spheres
lighting gouraud  % full lighting model
light
```

See also

: plot_point, plot_box, plot_circle, plot_ellipse, plot_poly

---

plotvol

Set the bounds for a 2D or 3D plot

2D plots::

`PLOTVOL([WX WY])` creates a new axis, and sets the bounds for a 2D plot, with X spanning [-WX, WX] and Y spanning [-WY,WY].

`PLOTVOL([XMIN XMAX YMIN YMAX])` as above but the X and Y axis limits are explicitly provided.

3D plots::

`PLOTVOL(W)` creates a new axis, and sets the bounds for a 3D plot with X, Y and Z spanning the interval -`W` to `W`.

`PLOTVOL([WX WY WZ])` as above with X spanning [-WX, WX], Y spanning [-WY, WY] and Z spanning [-WZ, WZ].

Notes

- The axes are labelled, grid is enabled, aspect ratio set to 1:1.
- Hold is enabled for subsequent plots.

See also

: axis

Plucker

Plucker coordinate class

Concrete class to represent a 3D line using Plucker coordinates.

Methods

Plucker	Contructor from points
Plucker.planes	Constructor from planes
Plucker.pointdir	Constructor from point and direction

Information and test methods

closest	closest point on line
commonperp	common perpendicular for two lines
contains	test if point is on line
distance	minimum distance between two lines
intersects	intersection point for two lines
intersect_plane	intersection points with a plane
intersect_volume	intersection points with a volume
pp	principal point
ppd	principal point distance from origin
point	generate point on line

Conversion methods

char	convert to human readable string
double	convert to 6-vector
skew	convert to 4×4 skew symmetric matrix

Display and print methods

display	display in human readable form
plot	plot line

Operators

$*$	multiply Plucker matrix by a general matrix
|	test if lines are parallel
^	test if lines intersect
==	test if two lines are equivalent
$\sim=$	test if lines are not equivalent

Notes

- This is reference (handle) class object
- Plucker objects can be used in vectors and arrays

References

- Ken Shoemake, "Ray Tracing News", Volume 11, Number 1 http://www.realtimerendering.com/resources/RTNews/html/rtnv11n1.html#art3
- Matt Mason lecture notes http://www.cs.cmu.edu/afs/cs/academic/class/16741-s07/www/lectures/lecture9.pdf
- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p596-7.

Implementation notes

- The internal representation is two 3-vectors: v (direction), w (moment).
- There is a huge variety of notation used across the literature, as well as the ordering of the direction and moment components in the 6-vector.

Plucker.Plucker

Create Plucker line object

`P = Plucker(P1, P2)` create a **Plucker** object that represents the line joining the 3D points `P1` (3×1) and `P2` (3×1). The direction is from `P2` to `P1`.

`P = Plucker(X)` creates a **Plucker** object from `X` (6×1) = [V,W] where V (3×1) is the moment and W (3×1) is the line direction.

`P = Plucker(L)` creates a copy of the **Plucker** object `L`.

Plucker.char

Convert to string

`s = P.char()` is a string showing **Plucker** parameters in a compact single line format.

See also

Plucker.display

Plucker.closest

Point on line closest to given point

`P = PL.closest(X)` is the coordinate of a point (3×1) on the line that is closest to the point `X` (3×1).

`[P,d] = PL.closest(X)` as above but also returns the minimum distance between the point and the line.

`[P,dist,lambda] = PL.closest(X)` as above but also returns the line parameter `lambda` corresponding to the point on the line, ie. `P` = PL.point(`lambda`)

See also

Plucker.point

Plucker.commonperp

Common perpendicular to two lines

`P = PL1.commonperp(PL2)` is a **Plucker** object representing the common perpendicular line between the lines represented by the Plucker objects PL1 and `PL2`.

See also

Plucker.intersect

Plucker.contains

Test if point is on the line

`PL.contains(X)` is true if the point `X` (3×1) lies on the line defined by the Plucker object PL.

Plucker.display

Display parameters

`P.display()` displays the **Plucker** parameters in compact single line format.

Notes

- This method is invoked implicitly at the command line when the result of an expression is a Plucker object and the command has no trailing
- semicolon.

See also

Plucker.char

Plucker.distance

Distance between lines

`d = P1.distance(P2)` is the minimum distance between two lines represented by Plucker objects P1 and `P2`.

Notes

- Works for parallel, skew and intersecting lines.

Plucker.double

Convert Plucker coordinates to real vector

`PL.double()` is a vector (6×1) comprising the **Plucker** moment and direction vectors.

Plucker.eq

Test if two lines are equivalent

PL1 == PL2 is true if the **Plucker** objects describe the same line in space. Note that because of the over parameterization, lines can be equivalent even if they have different parameters.

Plucker.get.uw

Line direction as a unit vector

`PL.UW` is a unit-vector parallel to the line

Plucker.intersect_plane

Line intersection with plane

`X = PL.intersect_plane(PI)` is the point where the **Plucker** line PL intersects the plane `PI`. `X`=[] if no intersection.

The plane `PI` can be either:

- a vector (1×4) = [a b c d] to describe the plane ax+by+cz+d=0.
- a structure with a normal `PI`.n (3×1) and an offset `PI`.p (1×1) such that `PI`.n `X` + `PI`.p = 0.

`[X,lambda] = PL.intersect_plane(P)` as above but also returns the line parameter at the intersection point, ie. `X` = PL.point(`lambda`).

See also

Plucker.point

Plucker.intersect_volume

Line intersection with volume

`P = PL.intersect_volume(BOUNDS)` is a matrix $(3 \times N)$ with columns that indicate where the Plcuker line PL intersects the faces of a volume specified by `BOUNDS` = [xmin xmax ymin ymax zmin zmax]. The number of columns N is either 0 (the line is outside the plot volume) or 2 (where the line pierces the bounding volume).

`[P,lambda] = PL.intersect_volume(bounds, line)` as above but also returns the `line` parameters $(1 \times N)$ at the intersection points, ie. X = PL.point(`lambda`).

See also

Plucker.plot, Plucker.point

Plucker.intersects

Find intersection of two lines

`P = P1.intersects(P2)` is the point of intersection (3×1) of the lines represented by Plucker objects P1 and `P2`. `P` = [] if the lines do not intersect, or the lines are equivalent.

Notes

- Can be used in operator form as $P1^{P2}$.
- Returns [] if the lines are equivalent (P1==`P2`) since they would intersect at an infinite number of points.

See also

Plucker.commonperp, Plucker.eq, Plucker.mpower

Plucker.isparallel

Test if lines are parallel

`P1.isparallel(P2)` is true if the lines represented by **Plucker** objects P1 and `P2` are parallel.

See also

Plucker.or, Plucker.intersects

Plucker.mpower

Test if lines intersect

`P1^P2` is true if lines represented by **Plucker** objects `P1` and `P2` intersect at a point.

Notes

- Is false if the lines are equivalent since they would intersect at an infinite number of points.

See also

Plucker.intersects, Plucker.parallel

Plucker.mtimes

Plucker multiplication

`PL1 * PL2` is the scalar reciprocal product.

`PL * M` is the product of the **Plucker** skew matrix and `M` $(4 \times N)$.

`M * PL` is the product of `M` $(N \times 4)$ and the **Plucker** skew matrix (4×4).

Notes

- The * operator is overloaded for convenience.
- Multiplication or composition of Plucker lines is not defined.
- Premultiplying by an SE3 will transform the line with respect to the world coordinate frame.

See also

Plucker.skew, SE3.mtimes

Plucker.ne

Test if two lines are not equivalent

PL1 $\neq$ PL2 is true if the **Plucker** objects describe different lines in space. Note that because of the over parameterization, lines can be equivalent even if they have different parameters.

Plucker.or

Test if lines are parallel

`P1|P2` is true if the lines represented by **Plucker** objects `P1` and `P2` are parallel.

Notes

- Can be used in operator form as `P1|P2`.

See also

Plucker.isparallel, Plucker.mpower

Plucker.planes

Create Plucker line from two planes

`P = Plucker.planes(PI1, PI2)` is a **Plucker** object that represents the line formed by the intersection of two planes `PI1`, `PI2` (each 4×1).

Notes

- Planes are given by the 4-vector [a b c d] to represent ax+by+cz+d=0.

Plucker.plot

Plot a line

`PL.plot(OPTIONS)` adds the **Plucker** line PL to the current plot volume.

`PL.plot(B, OPTIONS)` as above but plots within the plot bounds `B` = [XMIN XMAX YMIN YMAX ZMIN ZMAX].

Options

- Are passed directly to plot3, eg. 'k–', 'LineWidth', etc.

Notes

- If the line does not intersect the current plot volume nothing will be displayed.

See also

plot3, Plucker.intersect_volume

Plucker.point

Generate point on line

`P = PL.point(LAMBDA)` is a point on the line, where `LAMBDA` is the parametric distance along the line from the principal point of the line `P` = PP + PL.UW*`LAMBDA`.

See also

Plucker.pp, Plucker.closest

Plucker.pointdir

Construct Plucker line from point and direction

`P = Plucker.pointdir(P, W)` is a **Plucker** object that represents the line containing the point `P` (3×1) and parallel to the direction vector `W` (3×1).

See also

: Plucker

Plucker.pp

Principal point of the line

`P = PL.pp()` is the point on the line that is closest to the origin.

Notes

- Same as Plucker.point(0)

See also

Plucker.ppd, Plucker.point

Plucker.ppd

Distance from principal point to the origin

`P = PL.ppd()` is the distance from the principal point to the origin. This is the smallest distance of any point on the line to the origin.

See also

Plucker.pp

Plucker.skew

Skew matrix form of the line

`L = PL.skew()` is the **Plucker** matrix, a 4×4 skew-symmetric matrix representation of the line.

Notes

- For two homogeneous points P and Q on the line, PQ'-QP'is also skew symmetric.
- The projection of Plucker line by a perspective camera is a homogeneous line (3×1) given by vex(C*`L`*C') where C (3×4) is the camera matrix.

Quaternion

Quaternion class

A quaternion is 4-element mathematical object comprising a scalar s, and a vector v which can be considered as a pair (s,v). In the Toolbox it is denoted by q = s <<vx, vy, vz>>.

A quaternion of unit length can be used to represent 3D orientation and is implemented by the subclass UnitQuaternion.

Constructors

Quaternion	general constructor
Quaternion.pure	pure quaternion

Display and print methods

display	print in human readable form

Group operations

$*$	quaternion (Hamilton) product or elementwise multiplication by scalar
/	multiply by inverse or elementwise division by scalar
^	exponentiate (integer only)
+	elementwise sum of quaternion elements
-	elementwise difference of quaternion elements
conj	conjugate
exp	exponential
log	logarithm
inv	inverse
prod	product of elements
unit	unitized quaternion

Methods

inner	inner product
isequal	test for non-equality
norm	norm, or length

Conversion methods

char	convert to string
double	quaternion elements as 4-vector
matrix	quaternion as a 4×4 matrix

Overloaded operators

==	test for quaternion equality
$\sim=$	test for quaternion inequality

Properties (read only)

s	real part
v	vector part

Notes

- This is reference (handle) class object

- Quaternion objects can be used in vectors and arrays.

References

- Animating rotation with quaternion curves, K. Shoemake, in Proceedings of ACM SIGGRAPH, (San Fran cisco), pp. 245-254, 1985.
- On homogeneous transforms, quaternions, and computational efficiency, J. Funda, R. Taylor, and R. Paul,
- IEEE Transactions on Robotics and Automation, vol. 6, pp. 382-388, June 1990.
- Quaternions for Computer Graphics, J. Vince, Springer 2011.
- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p44-45.

See also

UnitQuaternion

Quaternion.Quaternion

Construct a quaternion object

`Q = Quaternion(S, V)` is a **Quaternion** formed from the scalar `S` and vector part `V` (1×3).

`Q = Quaternion([S V1 V2 V3])` is a **Quaternion** formed by specifying directly its 4 elements.

`Q = Quaternion()` is a zero **Quaternion**, all its elements are zero.

Notes

- The constructor is not vectorized, it cannot create a vector of Quaternions.

Quaternion.char

Convert to string

`S = Q.char()` is a compact string representation of the **Quaternion**'s value as a 4-tuple. If Q is a vector then `S` has one line per element.

Notes

- The vector part is delimited by double angle brackets, to differentiate from a UnitQuaternion which is delimited by single angle brackets.

See also

UnitQuaternion.char

Quaternion.conj

Conjugate of a quaternion

`Q.conj()` is a **Quaternion** object representing the conjugate of Q.

Notes

- Conjugatation is the negation of the vector component.

See also

Quaternion.inv

Quaternion.display

Display quaternion

`Q.display()` displays a compact string representation of the **Quaternion**'s value as a 4-tuple. If Q is a vector then S has one line per element.

Notes

- This method is invoked implicitly at the command line when the result of an expression is a Quaternion object and the command has no trailing
- semicolon.
- The vector part is displayed with double brackets $<< 1, 0, 0 >>$ to distinguish it from a UnitQuaternion which displays as $< 1, 0, 0 >$
- If Q is a vector of Quaternion objects the elements are displayed on consecutive lines.

See also

Quaternion.char

Quaternion.double

Convert a quaternion to a 4-element vector

`V = Q.double()` is a row vector (1×4) comprising the **Quaternion** elements, scalar then vector, ie. `V` = [s vx vy vz]. If Q is a vector $(1 \times N)$ of Quaternion objects then `V` is a matrix $(N \times 4)$ with rows corresponding to the quaternion elements.

Quaternion.eq

Test quaternion equality

Q1 == Q2 is true if the Quaternions Q1 and Q2 are equal.

Notes

- Overloaded operator '=='.
- Equality means elementwise equality of Quaternion elements.
- If either, or both, of Q1 or Q2 are vectors, then the result is a vector.
  - if Q1 is a vector $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1(i)==Q2.
  - if Q2 is a vector $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1==Q2(i).
  - if both Q1 and Q2 are vectors $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1(i)==Q2(i).

See also

Quaternion.ne

Quaternion.exp

Exponential of quaternion

`Q.log()` is the logarithm of the **Quaternion** Q.

See also

Quaternion.exp

Quaternion.inner

Quaternion inner product

`V = Q1.inner(Q2)` is the inner (dot) product of two vectors (1×4), comprising the elements of Q1 and `Q2` respectively.

Notes

- Q1.inner(Q1) is the same as Q1.norm().

See also

Quaternion.norm

Quaternion.inv

Invert a quaternion

`Q.inv()` is a **Quaternion** object representing the inverse of Q.

Notes

- If Q is a vector then an equal length vector of Quaternion objects is computed representing the elementwise inverse of Q.

See also

Quaternion.conj

Quaternion.isequal

Test quaternion element equality

`ISEQUAL(Q1,Q2)` is true if the Quaternions `Q1` and `Q2` are equal.

Notes

- Used by test suite verifyEqual() in addition to eq().
- Invokes eq() so respects double mapping for UnitQuaternion.

See also

Quaternion.eq

Quaternion.log

Logarithm of quaternion

`Q.log()` is the logarithm of the **Quaternion** Q.

See also
Quaternion.exp

Quaternion.matrix

Matrix representation of Quaternion
`Q.matrix()` is a matrix (4×4) representation of the **Quaternion** Q.

Quaternion, or Hamilton, multiplication can be implemented as a matrix-vector product, where the column-vector is the elements of a second quaternion:

```
matrix(Q1) * double(Q2)'
```

Notes
- This matrix is not unique, other matrices will serve the purpose for multiplication, see `https://en.wikipedia.org/wiki/Quaternion`#Matrix_representations
- The determinant of the matrix is the norm of the Quaternion to the fourth power.

See also
Quaternion.double, Quaternion.mtimes

Quaternion.minus

Subtract quaternions
`Q1-Q2` is a **Quaternion** formed from the element-wise difference of **Quaternion** elements.

`Q1-V` is a **Quaternion** formed from the element-wise difference of `Q1` and the vector `V` (1×4).

Notes
- Overloaded operator '-'.
- Effectively `V` is promoted to a Quaternion.

See also
Quaternion.plus

Quaternion.mpower

Raise quaternion to integer power
`Q^N` is the **Quaternion** `Q` raised to the integer power `N`.

Notes
- Overloaded operator '^textquotesingle .
- `N` must be an integer, computed by repeated multiplication.

See also
Quaternion.mtimes

Quaternion.mrdivide

Quaternion quotient.

R = Q1/Q2	is a Quaternion formed by Hamilton product of Q1 and inv(Q2).
R = Q/S	is the element-wise division of Quaternion elements by the scalar S.

Notes

- Overloaded operator '/'.
- If either, or both, of Q1 or Q2 are vectors, then the result is a vector.
  - if Q1 is a vector $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1(i)./Q2.
  - if Q2 is a vector $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1./Q2(i).
  - if both Q1 and Q2 are vectors $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1(i)./Q2(i).

See also

Quaternion.mtimes, Quaternion.mpower, Quaternion.plus, Quaternion.minus

---

Quaternion.mtimes

Multiply a quaternion object

$Q1*Q2$	is a Quaternion formed by the Hamilton product of two Quaternions.
$Q*S$	is the element-wise multiplication of Quaternion elements by the scalar S.
$S*Q$	is the element-wise multiplication of Quaternion elements by the scalar S.

Notes

- Overloaded operator '*'.
- If either, or both, of Q1 or Q2 are vectors, then the result is a vector.
  - if Q1 is a vector $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1(i)*Q2.
  - if Q2 is a vector $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1*Q2(i).
  - if both Q1 and Q2 are vectors $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1(i)*Q2(i).

See also

Quaternion.mrdivide, Quaternion.mpower

Quaternion.ne

Test quaternion inequality

Q1 $\neq$ Q2 is true if the Quaternions Q1 and Q2 are not equal.

Notes

- Overloaded operator '$\neq$'.
- If either, or both, of Q1 or Q2 are vectors, then the result is a vector.
  - if Q1 is a vector $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1(i)$\neq$Q2.

  - if Q2 is a vector $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1≠Q2(i).
  - if both Q1 and Q2 are vectors $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1(i)≠Q2(i).

See also

Quaternion.eq

Quaternion.new

Construct a new quaternion

`QN = Q.new()` constructs a new **Quaternion** object.

`QN = Q.new([S, V1, V2, V3])` as above but specified directly by its 4 elements.

`QN = Q.new(S, V)` as above but specified directly by the scalar `S` and vector part `V` (1×3)

Notes

- Polymorphic with UnitQuaternion and RTBPose derived classes.

Quaternion.norm

Quaternion magnitude

`Q.norm(Q)` is the scalar norm or magnitude of the **Quaternion** `Q`.

Notes

- This is the Euclidean norm of the Quaternion written as a 4-vector.
- A unit-quaternion has a norm of one and is represented by the UnitQuaternion class.

See also

Quaternion.inner, Quaternion.unit, UnitQuaternion

Quaternion.plus

Add quaternions

`Q1+Q2` is a **Quaternion** formed from the element-wise sum of **Quaternion** elements.

`Q1+V` is a **Quaternion** formed from the element-wise sum of `Q1` and the vector `V` (1×4).

Notes

- Overloaded operator '+'.
- Effectively `V` is promoted to a Quaternion.

See also

Quaternion.minus

Quaternion.prod

Product of quaternions

`prod(Q)` is the product of the elements of the vector of **Quaternion** objects `Q`.

See also

Quaternion.mtimes, RTBPose.prod

Quaternion.pure

Construct a pure quaternion

`Q = Quaternion.pure(V)` is a pure **Quaternion** formed from the vector `V` (1×3) and has a zero scalar part.

Quaternion.set.s

Set scalar component

`Q.s` = S sets the scalar part of the **Quaternion** object to S.

Quaternion.set.v

Set vector component

`Q.v` = V sets the vector part of the **Quaternion** object to V (1×3).

Quaternion.unit

Unitize a quaternion

`QU = Q.unit()` is a **Quaternion** with a norm of 1. If Q is a vector ($1 \times N$) then `QU` is also a vector ($1 \times N$).

Notes

- This is Quaternion of unit norm, not a UnitQuaternion object.

See also

Quaternion.norm, UnitQuaternion

r2t

Convert rotation matrix to a homogeneous transform

`T = R2T(R)` is an SE(2) or SE(3) homogeneous transform equivalent to an SO(2) or SO(3) orthonormal rotation matrix `R` with a zero translational component. Works for `T` in either SE(2) or SE(3):

- if `R` is 2×2 then `T` is 3×3, or
- if `R` is 3×3 then `T` is 4×4.

Notes

- Translational component is zero.
- For a rotation matrix sequence ($K\times K\times N$) returns a homogeneous transform sequence ($K+1\times K$+1 $\times N$).

See also

t2r

---

randinit

Reset random number generator

RANDINIT resets the defaul random number stream. For example:

```
>> rand
ans =
   0.8147
>> rand
ans =
    0.9058
>> rand
ans =
    0.1270
>> randinit
>> rand
ans =
    0.8147
```

---

rot2

SO(2) rotation matrix

`R = ROT2(THETA)` is an SO(2) rotation matrix (2×2) representing a rotation of `THETA` radians.

`R = ROT2(THETA, 'deg')` as above but `THETA` is in degrees.

See also

trot2, isrot2, trplot2, rotx, roty, rotz, SO2

---

rotx

SO(3) rotation about X axis

`R = ROTX(THETA)` is an SO(3) rotation matrix (3×3) representing a rotation of `THETA` radians about the x-axis.

`R = ROTX(THETA, 'deg')` as above but `THETA` is in degrees.

See also

trotx, roty, rotz, angvec2r, rot2, SO3.Rx

---

roty

SO(3) rotation about Y axis

`R = ROTY(THETA)` is an SO(3) rotation matrix (3×3) representing a rotation of `THETA` radians about the y-axis.

`R = ROTY(THETA, 'deg')` as above but `THETA` is in degrees.

See also

troty, rotx, rotz, angvec2r, rot2, SO3.Ry

---

rotz

SO(3) rotation about Z axis

`R = ROTZ(THETA)` is an SO(3) rotation matrix (3×3) representing a rotation of `THETA` radians about the z-axis.

`R = ROTZ(THETA, 'deg')` as above but `THETA` is in degrees.

See also

trotz, rotx, roty, angvec2r, rot2, SO3.Rx

---

rpy2jac

Jacobian from RPY angle rates to angular velocity

`J = RPY2JAC(RPY, OPTIONS)` is a Jacobian matrix (3×3) that maps ZYX roll-pitch-yaw angle rates to angular velocity at the operating point `RPY`=[R,P,Y].

`J = RPY2JAC(R, P, Y, OPTIONS)` as above but the roll-pitch-yaw angles are passed as separate arguments.

Options

'xyz'	Use XYZ roll-pitch-yaw angles
'yxz'	Use YXZ roll-pitch-yaw angles

Notes

- Used in the creation of an analytical Jacobian.
- Angles in radians, rates in radians/sec.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p232-3.

See also

eul2jac, rpy2r, SerialLink.jacobe

---

rpy2r

Roll-pitch-yaw angles to SO(3) rotation matrix

`R = RPY2R(ROLL, PITCH, YAW, OPTIONS)` is an SO(3) orthonornal rotation matrix (3×3) equivalent to the specified roll, pitch, yaw angles angles. These correspond to rotations about the Z, Y, X axes respectively. If `ROLL`, `PITCH`, `YAW` are column vectors ($N\times1$) then they are assumed to represent a trajectory and `R` is a three-dimensional matrix ($3\times3\times N$), where the last index corresponds to rows of `ROLL`, `PITCH`, `YAW`.

`R = RPY2R(RPY, OPTIONS)` as above but the roll, pitch, yaw angles are taken from the vector (1×3) `RPY`=[`ROLL`,`PITCH`,`YAW`]. If `RPY` is a matrix ($N\times3$) then `R` is a three-dimensional matrix ($3\times3\times N$), where the last index corresponds to rows of `RPY` which are assumed to be [`ROLL`,`PITCH`,`YAW`].

Options

'deg'	Compute angles in degrees (radians default)
'xyz'	Rotations about X, Y, Z axes (for a robot gripper)
'zyx'	Rotations about Z, Y, X axes (for a mobile robot, default)
'yxz'	Rotations about Y, X, Z axes (for a camera)
'arm'	Rotations about X, Y, Z axes (for a robot arm)

'vehicle'Rotations about Z, Y, X axes (for a mobile robot)

'camera'	Rotations about Y, X, Z axes (for a camera)

Note

- Toolbox rel 8-9 has XYZ angle sequence as default.
- ZYX order is appropriate for vehicles with direction of travel in the X direction. XYZ order is appropriate if direction of travel is in the Z
- direction.
- 'arm', 'vehicle', 'camera'are synonyms for 'xyz', 'zyx'and 'yxz'respectively.

See also

tr2rpy, eul2tr

rpy2tr

Roll-pitch-yaw angles to SE(3) homogeneous transform

`T = RPY2TR(ROLL, PITCH, YAW, OPTIONS)` is an SE(3) homogeneous transformation matrix (4×4) with zero translation and rotation equivalent to the specified roll, pitch, yaw angles angles. These correspond to rotations about the Z, Y, X axes respectively. If `ROLL`, `PITCH`, `YAW` are column vectors ($N \times 1$) then they are assumed to represent a trajectory and R is a three-dimensional matrix ($4 \times 4 \times N$), where the last index corresponds to rows of `ROLL`, `PITCH`, `YAW`.

`T = RPY2TR(RPY, OPTIONS)` as above but the roll, pitch, yaw angles are taken from the vector (1×3) `RPY`=[`ROLL`,`PITCH`,`YAW`]. If `RPY` is a matrix ($N \times 3$) then R is a three-dimensional matrix ($4 \times 4 \times N$), where the last index corresponds to rows of `RPY` which are assumed to be `ROLL`,`PITCH`,`YAW`].

Options

'deg'	Compute angles in degrees (radians default)
'xyz'	Rotations about X, Y, Z axes (for a robot gripper)
'zyx'	Rotations about Z, Y, X axes (for a mobile robot, default)
'yxz'	Rotations about Y, X, Z axes (for a camera)
'arm'	Rotations about X, Y, Z axes (for a robot arm)

'vehicle'Rotations about Z, Y, X axes (for a mobile robot)

'camera'	Rotations about Y, X, Z axes (for a camera)

Note

- Toolbox rel 8-9 has the reverse angle sequence as default.
- ZYX order is appropriate for vehicles with direction of travel in the X direction. XYZ order is appropriate if direction of travel is in the Z
- direction.
- 'arm', 'vehicle', 'camera'are synonyms for 'xyz', 'zyx'and 'yxz'respectively.

See also

tr2rpy, rpy2r, eul2tr

rt2tr

Convert rotation and translation to homogeneous transform

`TR = RT2TR(R, t)` is a homogeneous transformation matrix ($N+1 \times N+1$) formed from an orthonormal rotation matrix `R` ($N \times N$) and a translation vector `t` ($N \times 1$). Works for `R` in SO(2) or SO(3):

- If `R` is 2×2 and `t` is 2×1, then `TR` is 3×3

- If `R` is 3×3 and `t` is 3×1, then `TR` is 4×4

For a sequence `R` $(N \times N \times K)$ and `t` $(N \times K)$ results in a transform sequence $(N+1 \times N{+}1 \times K)$.

Notes

- The validity of `R` is not checked

See also

t2r, r2t, tr2rt

RTBPose

Superclass for SO2, SO3, SE2, SE3

This abstract class provides common methods for the 2D and 3D orientation and pose classes: SO2, SE2, SO3 and SE3.

Display and print methods

animate	graphically animate coordinate frame for pose
display	print the pose in human readable matrix form
plot	graphically display coordinate frame for pose
print	print the pose in single line format

Group operations

$*$	mtimes: multiplication within group, also transform vector
/	mrdivide: multiplication within group by inverse
prod	mower: product of elements

Methods

dim	dimension of the underlying matrix
isSE	true for SE2 and SE3
issym	true if value is symbolic
simplify	apply symbolic simplification to all elements
vpa	apply vpa to all elements

% Conversion methods::

char	convert to human readable matrix as a string
double	convert to real rotation or homogeneous transformation matrix

Operators

+	plus: elementwise addition, result is a matrix
-	minus: elementwise subtraction, result is a matrix
==	eq: test equality
$\sim=$	ne: test inequality

Compatibility methods

A number of compatibility methods give the same behaviour as the classic RTB functions:

tr2rt	convert to rotation matrix and translation vector
t2r	convert to rotation matrix
tranimate	animate coordinate frame
trprint	print single line representation
trprint2	print single line representation
trplot	plot coordinate frame
trplot2	plot coordinate frame

Notes

- This is a handle class.
- RTBPose subclasses can be used in vectors and arrays.
- Multiplication and division with normalization operations are performed in the subclasses.
- SO3 is polymorphic with UnitQuaternion making it easy to change rotational representations.

See also

SO2, SO3, SE2, SE3

RTBPose.animate

Animate a coordinate frame

`RTBPose.animate(P1, P2, OPTIONS)` animates a 3D coordinate frame moving from RTBPose `P1` to RTBPose `P2`.

`RTBPose.animate(P, OPTIONS)` animates a coordinate frame moving from the identity pose to the RTBPose `P`.

`RTBPose.animate(PV, OPTIONS)` animates a trajectory, where `PV` is a vector of RTBPose subclass objects.

%

Options

'fps', fps	Number of frames per second to display (default 10)
'nsteps', n	The number of steps along the path (default 50)
'axis',A	Axis bounds [xmin, xmax, ymin, ymax, zmin, zmax]
'movie',M	Save frames as files in the folder M
'cleanup'	Remove the frame at end of animation
'noxyz'	Don't label the axes
'rgb'	Color the axes in the order x=red, y=green, z=blue
'retain'	Retain frames, don't animate

Additional options are passed through to tranimate or tranimate2.

See also

tranimate, tranimate2

RTBPose.char

Convert to string

`s = P.char()` is a string showing **RTBPose** matrix elements as a matrix.

See also

RTBPose.display

RTBPose.dim

Dimension

`N = P.dim()` is the dimension of the matrix representing the **RTBPose** subclass instance P. It is 2 for SO2, 3 for SE2 and SO3, and 4 for SE3.

RTBPose.display

Display pose in matrix form

`P.display()` displays the matrix elements for the **RTBPose** instance P to the console. If P is a vector ($1 \times N$) then matrices are displayed sequentially.

Notes

- This method is invoked implicitly at the command line when the result of an expression is an RTBPose subclass object and the command has no trailing
- semicolon.
- If the function cprintf is found is used to colorise the matrix: rotational elements in red, translational in blue.
- See `https://www.mathworks.com/matlabcentral/fileexchange/24093-cprintf-display-formatted-colored-text-in-the-command-window`

See also

SO2, SO3, SE2, SE3

RTBPose.double

Convert to matrix

`T = P.double()` is a native matrix representation of the **RTBPose** subclass instance P, either a rotation matrix or a homogeneous transformation matrix.

If P is a vector ($1 \times N$) then `T` will be a 3-dimensional array ($M \times M \times N$).

Notes

- If the pose is symbolic the result will be a symbolic matrix.

RTBPose.ishomog

Test if SE3 class (compatibility)

`ISHOMOG(T)` is true (1) if `T` is of class SE3.

See also

ishomog

RTBPose.ishomog2

Test if SE2 class (compatibility)

`ISHOMOG2(T)` is true (1) if `T` is of class SE2.

See also

ishomog2

RTBPose.isrot

Test if SO3 class (compatibility)

`ISROT(R)` is true (1) if `R` is of class SO3.

See also

isrot

RTBPose.isrot2

Test if SO2 class (compatibility)

`ISROT2(R)` is true (1) if `R` is of class SO2.

See also

isrot2

RTBPose.isSE

Test if rigid-body motion

`P.isSE()` is true if P is an instance of the **RTBPose** sublass SE2 or SE3.

RTBPose.issym

Test if pose is symbolic

`P.issym()` is true if the **RTBPose** subclass instance P has symbolic rather than real values.

RTBPose.isvec

Test if vector (compatibility)

`ISVEC(T)` is always false.

See also

isvec

RTBPose.minus

Subtract poses

`P1-P2` is the elementwise difference of the matrix elements of the two poses. The result is a matrix not the input class type since the result of subtraction is not in the group.

RTBPose.mpower

Exponential of pose

`P^N` is an **RTBPose** subclass instance equal to **RTBPose** subclass instance `P` raised to the integer power `N`. It is equivalent of compounding `P` with itself `N`-1 times.

Notes

- `N` can be 0 in which case the result is the identity element.
- `N` can be negative which is equivalent to the inverse of ^-`N`).

See also

RTBPose.power, RTBPose.mtimes, RTBPose.times

RTBPose.mrdivide

Compound SO2 object with inverse

`R = P/Q` is an **RTBPose** subclass instance representing the composition of the RTB-Pose subclass instance `P` by the inverse of the RTBPose subclass instance `Q`.

If either, or both, of `P` or `Q` are vectors, then the result is a vector.

- if `P` is a vector ($1 \times N$) then `R` is a vector ($1 \times N$) such that `R`(i) = `P`(i)/`Q`.
- if `P` is a vector ($1 \times N$) then `R` is a vector ($1 \times N$) such that `R`(i) = `P`/`Q`(i).
- if both `P` and `Q` are vectors ($1 \times N$) then `R` is a vector ($1 \times N$) such that `R`(i) = `P`(i)/`Q`(i).

Notes

- Computed by matrix multiplication of their equivalent matrices with the second one inverted.

See also

RTBPose.mtimes

RTBPose.mtimes

Compound pose objects

`R = P*Q` is an **RTBPose** subclass instance representing the composition of the RTBPose subclass instance `P` by the RTBPose subclass instance `Q`.

If either, or both, of `P` or `Q` are vectors, then the result is a vector.

- if `P` is a vector ($1 \times N$) then `R` is a vector ($1 \times N$) such that `R`(i) = `P`(i)*`Q`.
- if `P` is a vector ($1 \times N$) then `R` is a vector ($1 \times N$) such that `R`(i) = `P`*`Q`(i).
- if both `P` and `Q` are vectors ($1 \times N$) then `R` is a vector ($1 \times N$) such that `R`(i) = `P`(i)*`Q`(i).

`W = P*V` is a column vector (2×1) which is the transformation of the column vector `V` (2×1) by the matrix representation of the RTBPose subclass instance `P`.

`P` can be a vector and/or `V` can be a matrix, a columnwise set of vectors:

- if `P` is a vector ($1 \times N$) then `W` is a matrix ($2 \times N$) such that `W`(:,i) = `P`(i)*`V`.
- if `V` is a matrix ($2 \times N$) `V` is a matrix ($2 \times N$) then `W` is a matrix ($2 \times N$) such that `W`(:,i) = `P`*`V`(:,i).
- if `P` is a vector ($1 \times N$) and `V` is a matrix ($2 \times N$) then `W` is a matrix ($2 \times N$) such that `W`(:,i) = `P`(i)*`V`(:,i).

Notes

- Computed by matrix multiplication of their equivalent matrices.

See also

RTBPose.mrdivide

RTBPose.plot

Draw a coordinate frame (compatibility)

`trplot(P, OPTIONS)` draws a 3D coordinate frame represented by `P` which is SO2, SO3, SE2 or SE3.

Compatible with matrix function trplot(T).

Options are passed through to trplot or trplot2 depending on the object type.

See also

trplot, trplot2

RTBPose.plus

Add poses

`P1+P2` is the elementwise summation of the matrix elements of the RTBPose subclass instances `P1` and `P2`. The result is a native matrix not the input class type since the result of addition is not in the group.

RTBPose.power

Exponential of pose

P.^N is the exponential of P where N is an integer, followed by normalization. It is equivalent of compounding the rigid-body motion of P with itself N-1 times.

Notes

- N can be 0 in which case the result is the identity matrix.
- N can be negative which is equivalent to the inverse of P.^{abs}(N).

See also

RTBPose.mpower, RTBPose.mtimes, RTBPose.times

RTBPose.print

Compact display of pose

P.print(OPTIONS) displays the **RTBPose** subclass instance P in a compact single-line format. If P is a vector then each element is printed on a separate line.

Example

```
T = SE3.rand()
T.print('rpy', 'xyz')  % display using XYZ RPY angles
```

Notes

- Options are passed through to trprint or trprint2 depending on the object type.

See also

trprint, trprint2

RTBPose.prod

Compound array of poses

P.prod() is an **RTBPose** subclass instance representing the product (composition) of the successive elements of P ($1 \times N$).

Note

- Composition is performed with the .* operator, ie. the product is renormalized at every step.

See also

RTBPose.times

RTBPose.simplify

Symbolic simplification

P2 = P.simplify() applies symbolic simplification to each element of internal matrix representation of the RTBPose subclass instance P.

See also

simplify

RTBPose.subs

Symbolic substitution

`T = subs(T, old, new)` replaces `old` with `new` in the symbolic transformation `T`.

See also: subs

RTBPose.t2r

Get rotation matrix (compatibility)

`t2r(P)` is a native matrix corresponding to the rotational component of the SE2 or SE3 instance `P`.

See also

t2r

RTBPose.tr2rt

Split rotational and translational components (compatibility)

`[R,t] = tr2rt(P)` is the rotation matrix and translation vector corresponding to the SE2 or SE3 instance `P`.

See also

tr2rt

RTBPose.tranimate

Animate a 3D coordinate frame (compatibility)

`TRANIMATE(P1, P2, OPTIONS)` animates a 3D coordinate frame moving between RTBPose subclass instances `P1` and pose `P2`.

`TRANIMATE(P, OPTIONS)` animates a 2D coordinate frame moving from the identity pose to the RTBPose subclass instance `P`.

`TRANIMATE(PV, OPTIONS)` animates a trajectory, where `PV` is a vector of RTBPose subclass instances.

Notes

- see tranimate for details of options.
- `P`, `P1`, `P2`, `PV` can be instances of SO3 or SE3.

See also

RTBPose.animate, tranimate

RTBPose.tranimate2

Animate a 2D coordinate frame (compatibility)

`TRANIMATE2(P1, P2, OPTIONS)` animates a 2D coordinate frame moving between RTBPose subclass instances `P1` and pose `P2`.

`TRANIMATE2(P, OPTIONS)` animates a 2D coordinate frame moving from the identity pose to the RTBPose subclass instance `P`.

`TRANIMATE2(PV, OPTIONS)` animates a trajectory, where `PV` is a vector of RTBPose subclass instances.

Notes

- see tranimate2 for details of options.
- `P`, `P1`, `P2`, `PV` can be instances of SO2 or SE2.

See also

RTBPose.animate, tranimate

RTBPose.trplot

Draw a 3D coordinate frame (compatibility)

`trplot(P, OPTIONS)` draws a 3D coordinate frame represented by **RTBPose** subclass instance `P`.

Notes

- see trplot for details of options.
- `P` can be instances of SO3 or SE3.

See also

RTBPose.plot, trplot

RTBPose.trplot2

Draw a 2D coordinate frame (compatibility)

`trplot2(P, OPTIONS)` draws a 2D coordinate frame represented by **RTBPose** subclass instance `P`.

Notes

- see trplot for details of options.
- `P` can be instances of SO2 or SE2.

See also

RTBPose.plot, trplot2

RTBPose.trprint

Compact display of 3D rotation or transform (compatibility)

`trprint(P, OPTIONS)` displays the **RTBPose** subclass instance `P` in a compact single-line format. If `P` is a vector then each element is printed on a separate line.

Notes

- see trprint for details of options.
- `P` can be instances of SO3 or SE3.

See also

RTBPose.print, trprint

RTBPose.trprint2

Compact display of 2D rotation or transform (compatibility)

`trprint2(P, OPTIONS)` displays the **RTBPose** subclass instance `P` in a compact single-line format. If `P` is a vector then each element is printed on a separate line.

Notes

- see trprint for details of options.
- `P` can be instances of SO2 or SE2.

See also

RTBPose.print, trprint2

RTBPose.vpa

Variable precision arithmetic

`P2 = P.vpa()` numerically evaluates each element of internal matrix representation of the RTBPose subclass instance P.

`P2 = P.vpa(D)` as above but with `D` decimal digit accuracy.

Notes

- Values of symbolic variables are taken from the workspace.

See also

vpa, simplify

SE2

Representation of 2D rigid-body motion

This subclasss of RTBPose is an object that represents rigid-body motion in 2D. Internally this is a 3×3 homogeneous transformation matrix (3×3) belonging to the group SE(2).

Constructor methods

SE2	general constructor
SE2.exp	exponentiate an se(2) matrix
SE2.rand	random transformation
new	new SE2 object

Display and print methods

animate	^graphically animate coordinate frame for pose
display	^print the pose in human readable matrix form
plot	^graphically display coordinate frame for pose
print	^print the pose in single line format

Group operations

$*$	^mtimes: multiplication (group operator, transform point)
/	^mrdivide: multiply by inverse
^	^mpower: exponentiate (integer only):
inv	inverse
prod	^product of elements

Methods

det	determinant of matrix component
eig	eigenvalues of matrix component
log	logarithm of rotation matrix
inv	inverse
simplify$*$	apply symbolic simplication to all elements
interp	interpolate between poses
theta	rotation angle

Information and test methods

dim	^returns 2
isSE	^returns true
issym	^test if rotation matrix has symbolic elements
SE2.isa	test if matrix is SE(2)

Conversion methods

char$*$	convert to human readable matrix as a string
SE2.convert	convert SE2 object or SE(2) matrix to SE2 object
double	convert to rotation matrix
R	convert to rotation matrix
SE3	convert to SE3 object with zero translation
SO2	convert rotational part to SO2 object
T	convert to homogeneous transformation matrix
Twist	convert to Twist object
t	get.t: convert to translation column vector

Compatibility methods

isrot2	^returns false
ishomog2	^returns true
tr2rt	^convert to rotation matrix and translation vector
t2r	^convert to rotation matrix

transl2	^translation as a row vector
trprint2	^print single line representation
trplot2	^plot coordinate frame
tranimate2	^animate coordinate frame

^inherited from RTBPose class.

See also

SO2, SE3, RTBPose

SE2.SE2

Construct an SE(2) object

Constructs an SE(2) pose object that contains a 3×3 homogeneous transformation matrix.

`T = SE2()` is the identity element, a null motion.

`T = SE2(X, Y)` is an object representing pure translation defined by `X` and `Y`.

`T = SE2(XY)` is an object representing pure translation defined by `XY` (2×1). If `XY` $(N \times 2)$ returns an array of SE2 objects, corresponding to the rows of `XY`.

`T = SE2(X, Y, THETA)` is an object representing translation, `X` and `Y`, and rotation, angle `THETA`.

`T = SE2(XY, THETA)` is an object representing translation, `XY` (2×1), and rotation, angle `THETA`.

`T = SE2(XYT)` is an object representing translation, `XYT`(1) and `XYT`(2), and rotation angle `XYT`(3). If `XYT` $(N \times 3)$ returns an array of SE2 objects, corresponding to the rows of `XYT`.

`T = SE2(T)` is an object representing translation and rotation defined by the SE(2) homogeneous transformation matrix `T` (3×3). If `T` $(3 \times 3 \times N)$ returns an array $(1 \times N)$ of SE2 objects, corresponding to the third index of `T`.

`T = SE2(R)` is an object representing pure rotation defined by the SO(2) rotation matrix `R` (2×2)

`T = SE2(R, XY)` is an object representing rotation defined by the orthonormal rotation matrix `R` (2×2) and position given by `XY` (2×1)

`T = SE2(T)` is a copy of the **SE2** object `T`. If `T` $(N \times 1)$ returns an array of **SE2** objects, corresponding to the index of `T`.

Options

'deg'	Angle is specified in degrees

Notes

- Arguments can be symbolic
- The form SE2(`XY`) is ambiguous with SE2(`R`) if `XY` has 2 rows, the second form is assumed.
- The form SE2(`XYT`) is ambiguous with SE2(`T`) if `XYT` has 3 rows, the second

form is assumed.

- R and T are checked to be valid SO(2) or SE(2) matrices.

SE2.convert

Convert to SE2

Q = SE2.convert(X) is an **SE2** object equivalent to X where X is either an SE2 object, or an SE(2) homogeneous transformation matrix (3×3).

SE2.exp

Construct SE2 from Lie algebra

SE2.exp(SIGMA) is the **SE2** rigid-body motion corresponding to the se(2) Lie algebra element SIGMA (3×3).

SE3.exp(TW) as above but the Lie algebra is represented as a twist vector TW (1×1).

Notes

- TW is the non-zero elements of X.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p25-31.

See also

trexp2, skewa

SE2.get.t

Get translational component

P.t is a column vector (2×1) representing the translational component of the rigid-body motion described by the SE2 object P.

Notes

- If P is a vector the result is a MATLAB comma separated list, in this case use P.transl().

See also

SE2.transl

SE2.interp

Interpolate between SO2 objects

P1.interp(P2, s) is an **SE2** object which is an interpolation between poses represented by SE2 objects P1 and P2. s varies from 0 (P1) to 1 (P2). If s is a vector $(1 \times N)$ then the result will be a vector of SE2 objects.

Notes

- It is an error if S is outside the interval 0 to 1.

See also

SO2.angle

SE2.inv

Inverse of SE2 object

`Q = inv(P)` is the inverse of the **SE2** object `P`.

Notes

- This is formed explicitly, no matrix inverse required.
- This is a group operator: input and output in the $SE(2)$ group.
- `P*Q` will be the identity group element (zero motion, identity matrix).

SE2.isa

Test if matrix is SE(2)

`SE2.isa(T)` is true (1) if the argument `T` is of dimension 3×3 or $3\times 3\times N$, else false (0).

`SE2.isa(T, true)` as above, but also checks the validity of the rotation submatrix.

Notes

- This is a class method.
- The first form is a fast, but incomplete, test for a transform in $SE(3)$.
- There is ambiguity in the dimensions of SE2 and SO3 in matrix form.

See also

SO3.ISA, SE2.ISA, SO2.ISA, ishomog2

SE2.log

Lie algebra

`se2 = P.log()` is the Lie algebra corresponding to the **SE2** object P. It is an augmented skew-symmetric matrix (3×3).

See also

SE2.Twist, logm, skewa, vexa

SE2.new

Construct a new object of the same type

`P2 = P.new(X)` creates a new object of the same type as P, by invoking the **SE2** constructor on the matrix `X` (3×3).

`P2 = P.new()` as above but defines a null motion.

Notes

- Serves as a dynamic constructor.
- This method is polymorphic across all RTBPose derived classes, and allows easy creation of a new object of the same class as an existing
- one without needing to explicitly determine its type.

See also

SE3.new, SO3.new, SO2.new

SE2.rand

Construct a random SE(2) object

`SE2.rand()` is an **SE2** object with a uniform random translation and a uniform random orientation. Random numbers are in the interval [-1 1] and rotations in the interval [-π π].

See also

rand

SE2.SE3

Lift to 3D

`Q = P.SE3()` is an SE3 object formed by lifting the rigid-body motion described by the SE2 object P from 2D to 3D. The rotation is about the z-axis, and the translation is within the xy-plane.

See also

SE3

SE2.set.t

Set translational component

`P.t` = TV sets the translational component of the rigid-body motion described by the SE2 object `P` to TV (2×1).

Notes

- TV can be a row or column vector.
- If TV contains a symbolic value then the entire matrix becomes symbolic.

SE2.SO2

Extract SO(2) rotation

`Q = SO2(P)` is an SO2 object that represents the rotational component of the SE2 rigid-body motion.

See also

SE2.R

SE2.T

Get homogeneous transformation matrix

`T = P.T()` is the homogeneous transformation matrix (3×3) associated with the SE2 object P, and has zero translational component. If P is a vector ($1 \times N$) then `T` ($3 \times 3 \times N$) is a stack of homogeneous transformation matrices, with the third dimension corresponding to the index of P.

See also

SO2.T

SE2.transl

Get translational component

`TV = P.transl()` is a row vector (1×2) representing the translational component of the rigid-body motion described by the SE2 object P. If P is a vector of objects ($1 \times N$) then `TV` ($N \times 2$) will have one row per object element.

SE2.Twist

Convert to Twist object

`TW = P.Twist()` is the equivalent Twist object. The elements of the twist are the unique elements of the Lie algebra of the SE2 object P.

See also

SE2.log, Twist

SE2.xyt

Extract configuration

`XYT = P.xyt()` is a column vector (3×1) comprising the minimum three configuration parameters of this rigid-body motion: translation (x,y) and rotation theta.

SE3

Representation of 3D rigid-body motion

This subclasss of RTBPose is an object that represents rigid-body motion in 2D. Internally this is a 3×3 homogeneous transformation matrix (4×4) belonging to the group SE(3).

Constructor methods

SE3	general constructor
SE3.angvec	rotation about vector
SE3.eul	rotation defined by Euler angles

SE3.exp	exponentiate an se(3) matrix
SE3.oa	rotation defined by o- and a-vectors
SE3.Rx	rotation about x-axis
SE3.Ry	rotation about y-axis
SE3.Rz	rotation about z-axis
SE3.rand	random transformation
SE3.rpy	rotation defined by roll-pitch-yaw angles
new	new SE3 object

Display and print methods

animate	^graphically animate coordinate frame for pose
display	^print the pose in human readable matrix form
plot	^graphically display coordinate frame for pose
print	^print the pose in single line format

Group operations

$*$	^mtimes: multiplication (group operator, transform point)
$.*$	^^times: multiplication (group operator) followed by normalization
/	^mrdivide: multiply by inverse
./	^^rdivide: multiply by inverse followed by normalization
^	^mpower: xponentiate (integer only)
.^	^power: exponentiate followed by normalization
inv	inverse
prod	^product of elements

Methods

det	determinant of matrix component
eig	eigenvalues of matrix component
log	logarithm of rotation matrixr>=0 && r<=1ub
simplify	^apply symbolic simplication to all elements
Ad	adjoint matrix (6×6)
increment	update pose based on incremental motion
interp	interpolate poses
velxform	compute velocity transformation
interp	interpolate between poses
ctraj	Cartesian motion
norm	normalize the rotation submatrix

Information and test methods

dim$*$	returns 4
isSE$*$	returns true
issym$*$	test if rotation matrix has symbolic elements
isidentity	test for null motion
SE3.isa	check if matrix is SE(3)

Conversion methods

char	convert to human readable matrix as a string
SE3.convert	convert SE3 object or SE(3) matrix to SE3 object
double	convert to SE(3) matrix

R	convert rotation part to SO(3) matrix
SO3	convert rotation part to SO3 object
T	convert to SE(3) matrix
t	translation column vector
toangvec	convert to rotation about vector form
todelta	convert to differential motion vector
toeul	convert to Euler angles
torpy	convert to roll-pitch-yaw angles
tv	translation column vector for vector of SE3
UnitQuaternion	convert to UnitQuaternion object

Compatibility methods

homtrans	apply to vector
isrot	^returns false
ishomog	^returns true
t2r	^convert to rotation matrix
tr2rt	^convert to rotation matrix and translation vector
tr2eul	^^convert to Euler angles
tr2rpy	^^convert to roll-pitch-yaw angles
tranimate	^animate coordinate frame
transl	translation as a row vector
trnorm	^^normalize the rotation matrix
trplot	^plot coordinate frame
trprint	^print single line representation

Other operators

+	^plus: elementwise addition, result is a matrix
-	^minus: elementwise subtraction, result is a matrix
==	^eq: test equality
∼=	^ne: test inequality

- ^inherited from RTBPose
- ^^inherited from SO3

Properties

n	get.n: normal (x) vector
o	get.o: orientation (y) vector
a	get.a: approach (z) vector
t	get.t: translation vector

For single SE3 objects only, for a vector of SE3 objects use the equivalent methods

t	translation as a 3×1 vector (read/write)
R	rotation as a 3×3 matrix (read)

Notes

- The properies R, t are implemented as MATLAB dependent properties. When applied to a vector of SE3 object the result is a comma-separated
- list which can be converted to a matrix by enclosing it in square
- brackets, eg [T.t] or more conveniently using the method T.transl

See also
SO3, SE2, RTBPose

SE3.SE3

Create an SE(3) object
Constructs an SE(3) pose object that contains a 4×4 homogeneous transformation matrix.

`T = SE3()` is the identity element, a null motion.

`T = SE3(X, Y, Z)` is an object representing pure translation defined by `X`, `Y` and `Z`.

`T = SE3(XYZ)` is an object representing pure translation defined by `XYZ` (3×1). If `XYZ` $(N \times 3)$ returns an array of SE3 objects, corresponding to the rows of `XYZ`.

`T = SE3(T)` is an object representing translation and rotation defined by the homogeneous transformation matrix `T` (3×3). If `T` $(3 \times 3 \times N)$ returns an array of SE3 objects, corresponding to the third index of `T`.

`T = SE3(R, XYZ)` is an object representing rotation defined by the orthonormal rotation matrix `R` (3×3) and position given by `XYZ` (3×1).

`T = SE3(T)` is a copy of the **SE3** object `T`. If `T` $(N \times 1)$ returns an array of **SE3** objects, corresponding to the index of `T`.

Options

'deg'	Angle is specified in degrees

Notes
- Arguments can be symbolic.
- `R` and `T` are checked to be valid SO(2) or SE(2) matrices.

SE3.Ad

Adjoint matrix
`A = P.Ad()` is the adjoint matrix (6×6) corresponding to the pose P.

See also
Twist.ad

SE3.angvec

Construct SE3 from angle and axis vector
`SE3.angvec(THETA, V)` is an **SE3** object equivalent to a rotation of `THETA` about the vector `V` and with zero translation.

Notes
- If `THETA` == 0 then return identity matrix.

- If `THETA` $\neq 0$ then `V` must have a finite length.

See also

SO3.angvec, eul2r, rpy2r, tr2angvec

SE3.convert

Convert to SE3

`Q = SE3.convert(X)` is an **SE3** object equivalent to `X` where `X` is either an SE3 object, or an SE(3) homogeneous transformation matrix (4×4).

SE3.ctraj

Cartesian trajectory between two poses

`TC = T0.ctraj(T1, N)` is a Cartesian trajectory defined by a vector of **SE3** objects ($1 \times$ `N`) from pose T0 to `T1`, both described by SE3 objects. There are `N` points on the trajectory that follow a trapezoidal velocity profile along the trajectory.

`TC = CTRAJ(T0, T1, S)` as above but the elements of `S` (`N` $\times 1$) specify the fractional distance along the path, and these values are in the range [0 1]. The i'th point corresponds to a distance `S`(i) along the path.

Notes

- In the second case `S` could be generated by a scalar trajectory generator such as TPOLY or LSPB (default).
- Orientation interpolation is performed using quaternion interpolation.

Reference

Robotics, Vision & Control, Sec 3.1.5, Peter Corke, Springer 2011

See also

lspb, mstraj, trinterp, ctraj, UnitQuaternion.interp

SE3.delta

Construct SE3 object from differential motion vector

`T = SE3.delta(D)` is an **SE3** pose object representing differential motion `D` (6×1).

The vector `D`=(dx, dy, dz, dRx, dRy, dRz) represents infinitessimal translation and rotation, and is an approximation to the instantaneous spatial velocity multiplied by time step.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p67.

See also

SE3.todelta, SE3.increment, tr2delta

SE3.eul

Construct SE3 from Euler angles

`P = SO3.eul(PHI, THETA, PSI, OPTIONS)` is an **SE3** object equivalent to the specified Euler angles. These correspond to rotations about the Z, Y, Z axes respectively. If `PHI`, `THETA`, `PSI` are column vectors ($N \times 1$) then they are assumed to represent a trajectory then `P` is a vector ($1 \times N$) of SE3 objects.

`P = SO3.eul(EUL, OPTIONS)` as above but the Euler angles are taken from consecutive columns of the passed matrix `EUL` = [`PHI THETA PSI`]. If `EUL` is a matrix ($N \times 3$) then they are assumed to represent a trajectory then `P` is a vector ($1 \times N$) of SE3 objects.

Options

'deg'	Angles are specified in degrees (default radians)

Note

- Translation is zero.
- The vectors `PHI`, `THETA`, `PSI` must be of the same length.

Reference

- Robotics, Vision & Control: Second Edition, `P`. Corke, Springer 2016; p36-37.

See also

SO3.eul, SE3.rpy, eul2tr, rpy2tr, tr2eul

SE3.exp

Construct SE3 from Lie algebra

`SE3.exp(SIGMA)` is the **SE3** rigid-body motion corresponding to the se(3) Lie algebra element `SIGMA` (4×4).

`SE3.exp(TW)` as above but the Lie algebra is represented as a twist vector `TW` (6×1).

`SE3.exp(SIGMA, THETA)` as above, but the motion is given by `SIGMA*THETA` where `SIGMA` is an se(3) element (4×4) whose rotation part has a unit norm.

Notes

- `TW` is the non-zero elements of X.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p42-43.

See also

trexp, skewa, Twist

SE3.homtrans

Apply transformation to points (compatibility)

`homtrans(P, V)` applies **SE3** pose object `P` to the points stored columnwise in `V` ($3 \times N$) and returns transformed points ($3 \times N$).

Notes

- `P` is an SE3 object defining the pose of {A} with respect to {B}.
- The points are defined with respect to frame {A} and are transformed to be with respect to frame {B}.
- Equivalent to `P*V` using overloaded SE3 operators.

See also

RTBPose.mtimes, homtrans

SE3.increment

Apply incremental motion to an SE3 pose

`P1 = P.increment(D)` is an **SE3** pose object formed by compounding the SE3 pose with the incremental motion described by `D` (6×1).

The vector `D`=(dx, dy, dz, dRx, dRy, dRz) represents infinitessimal translation and rotation, and is an approximation to the instantaneous spatial velocity multiplied by time step.

See also

SE3.todelta, SE3.delta, delta2tr, tr2delta

SE3.interp

Interpolate SE3 poses

`P1.interp(P2, s)` is an **SE3** object representing an interpolation between poses represented by SE3 objects P1 and `P2`. `s` varies from 0 (P1) to 1 (`P2`). If `s` is a vector ($1 \times N$) then the result will be a vector of SO3 objects.

`P1.interp(P2, N)` as above but returns a vector ($1 \times$ `N`) of **SE3** objects interpolated between P1 and `P2` in `N` steps.

Notes

- The rotational interpolation (slerp) can be interpretted

as interpolation along a great circle arc on a sphere.

- It is an error if any element of S is outside the interval 0 to 1.

See also

trinterp, ctraj, UnitQuaternion

SE3.inv

Inverse of SE3 object

`Q = inv(P)` is the inverse of the **SE3** object `P`.

Notes

- This is formed explicitly, no matrix inverse required.
- This is a group operator: input and output in the SE(3)) group.

- `P*Q` will be the identity group element (zero motion, identity matrix).

SE3.isa

Test if matrix is SE(3)

`SE3.ISA(T)` is true (1) if the argument `T` is of dimension 4×4 or $4 \times 4 \times N$, else false (0).

`SE3.ISA(T, 'valid')` as above, but also checks the validity of the rotation sub-matrix.

Notes

- Is a class method.
- The first form is a fast, but incomplete, test for a transform in SE(3).

See also

SO3.isa, SE2.isa, SO2.isa

SE3.isidentity

Test if identity element

`P.isidentity()` is true if the **SE3** object P corresponds to null motion, that is, its homogeneous transformation matrix is identity.

SE3.log

Lie algebra

`P.log()` is the Lie algebra corresponding to the **SE3** object P. It is an augmented skew-symmetric matrix (4×4).

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p42-43.

See also

SE3.logs, SE3.Twist, trlog, logm, skewa, vexa

SE3.logs

Lie algebra in vector form

`P.logs()` is the Lie algebra expressed as a vector (1×6) corresponding to the SE2 object P. The vector comprises the translational elements followed by the unique elements of the skew-symmetric upper-left 3×3 submatrix.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p42-43.

See also

SE3.log, SE3.Twist, trlog, logm

SE3.new

Construct a new object of the same type

`P2 = P.new(X)` creates a new object of the same type as P, by invoking the **SE3** constructor on the matrix `X` (4×4).

`P2 = P.new()` as above but defines a null motion.

Notes

- Serves as a dynamic constructor.
- This method is polymorphic across all RTBPose derived classes, and allows easy creation of a new object of the same class as an existing
- one without needing to explicitly determine its type.

See also

SO3.new, SO2.new, SE2.new

SE3.norm

Normalize rotation submatrix (compatibility)

`P.norm()` is an **SE3** pose equivalent to P but the rotation matrix is normalized (guaranteed to be orthogonal).

Notes

- Overrides the classic RTB function trnorm for an SE3 object.

See also

trnorm

SE3.oa

Construct SE3 from orientation and approach vectors

`P = SE3.oa(O, A)` is an **SE3** object for the specified orientation and approach vectors (3×1) formed from 3 vectors such that R = [N `O` `A`] and N = `O` x `A`, with zero translation.

Notes

- The rotation submatrix is guaranteed to be orthonormal so long as `O` and `A` are not parallel.
- The vectors `O` and `A` are parallel to the Y- and Z-axes of the coordinate frame.

References

- Robot manipulators: mathematics, programming and control Richard Paul, MIT Press, 1981.

See also

rpy2r, eul2r, oa2tr, SO3.oa

SE3.rand

Construct random SE3

`SE3.rand()` is an **SE3** object with a uniform random translation and a uniform random RPY/ZYX orientation. Random numbers are in the interval -1 to 1.

See also

rand

SE3.rpy

Construct SE3 from roll-pitch-yaw angles

`P = SE3.rpy(ROLL, PITCH, YAW, OPTIONS)` is an **SE3** object equivalent to the specified roll, pitch, yaw angles angles with zero translation. These correspond to rotations about the Z, Y, X axes respectively. If `ROLL`, `PITCH`, `YAW` are column vectors ($N \times 1$) then they are assumed to represent a trajectory then `P` is a vector ($1 \times N$) of SE3 objects.

`P = SE3.rpy(RPY, OPTIONS)` as above but the roll, pitch, yaw angles angles angles are taken from consecutive columns of the passed matrix `RPY` = [`ROLL`, `PITCH`, `YAW`]. If `RPY` is a matrix ($N \times 3$) then they are assumed to represent a trajectory and `P` is a vector ($1 \times N$) of SE3 objects.

Options

'deg'	Compute angles in degrees (radians default)
'xyz'	Rotations about X, Y, Z axes (for a robot gripper)
'yxz'	Rotations about Y, X, Z axes (for a camera)

Reference

- Robotics, Vision & Control: Second Edition, `P`. Corke, Springer 2016; p37-38.

See also

SO3.rpy, SE3.eul, tr2rpy, eul2tr

SE3.Rx

Construct SE3 from rotation about X axis

`P = SE3.Rx(THETA)` is an **SE3** object representing a rotation of `THETA` radians about the x-axis. If the `THETA` is a vector ($1 \times N$) then `P` will be a vector ($1 \times N$) of corresponding SE3 objects.

`P = SE3.Rx(THETA, 'deg')` as above but `THETA` is in degrees.

See also

SE3.Ry, SE3.Rz, rotx

SE3.Ry

Construct SE3 from rotation about Y axis

`P = SE3.Ry(THETA)` is an **SE3** object representing a rotation of `THETA` radians about the y-axis. If the `THETA` is a vector ($1 \times N$) then `P` will be a vector ($1 \times N$) of corresponding SE3 objects.

`P = SE3.Ry(THETA, 'deg')` as above but `THETA` is in degrees.

See also

SE3.Ry, SE3.Rz, rotx

SE3.Rz

Construct SE3 from rotation about Z axis

`P = SE3.Rz(THETA)` is an **SE3** object representing a rotation of `THETA` radians about the z-axis. If the `THETA` is a vector ($1 \times N$) then `P` will be a vector ($1 \times N$) of corresponding SE3 objects.

`P = SE3.Rz(THETA, 'deg')` as above but `THETA` is in degrees.

See also

SE3.Ry, SE3.Rz, rotx

SE3.set.t

Get translation vector

`T = P.t` is the translational part of **SE3** object as a 3-element column vector.

Notes

- If applied to a vector will return a comma-separated list, use .tv() instead.

See also

SE3.tv, transl

SE3.SO3

Convert rotational component to SO3 object

`P.SO3` is an `SO3` object representing the rotational component of the **SE3** pose `P`. If `P` is a vector ($N \times 1$) then the result is a vector ($N \times 1$).

SE3.T

Get homogeneous transformation matrix

`T = P.T()` is the homogeneous transformation matrix (3×3) associated with the SO2 object P, and has zero translational component. If P is a vector ($1 \times N$) then `T` ($3 \times 3 \times N$) is a stack of rotation matrices, with the third dimension corresponding to the index of P.

See also
SO2.T

SE3.toangvec

Convert to angle-vector form
`[THETA,V] = P.toangvec(OPTIONS)` is rotation expressed in terms of an angle `THETA` (1×1) about the axis `V` (1×3) equivalent to the rotational part of the SE3 object P.

If P is a vector ($1 \times N$) then `THETA` ($K \times 1$) is a vector of angles for corresponding elements of the vector and `V` ($K \times 3$) are the corresponding axes, one per row.

Options

'deg'	Return angle in degrees

Notes
- If no output arguments are specified the result is displayed.

See also
angvec2r, angvec2tr, trlog

SE3.todelta

Convert SE3 object to differential motion vector
`D = P0.todelta(P1)` is the differential motion (6×1) corresponding to infinitessimal motion (in the P0 frame) from SE3 pose P0 to `P1`.

The vector `D`=(dx, dy, dz, dRx, dRy, dRz) represents infinitessimal translation and rotation, and is an approximation to the instantaneous spatial velocity multiplied by time step.

`D = P.todelta()` as above but the motion is from the world frame to the **SE3** pose P.

Notes
- `D` is only an approximation to the motion, and assumes that P0≈`P1` or P≈eye(4,4).
- can be considered as an approximation to the effect of spatial velocity over a a time interval, average spatial velocity multiplied by time.

See also
SE3.increment, tr2delta, delta2tr

SE3.toeul

Convert to Euler angles
`EUL = P.toeul(OPTIONS)` are the ZYZ Euler angles (1×3) corresponding to the rotational part of the SE3 object P. The 3 angles `EUL`=[PHI,THETA,PSI] correspond to sequential rotations about the Z, Y and Z axes respectively.

If P is a vector ($1 \times N$) then each row of `EUL` corresponds to an element of the vector.

Options

'deg'	Compute angles in degrees (radians default)
'flip'	Choose first Euler angle to be in quadrant 2 or 3.

Notes

- There is a singularity for the case where THETA=0 in which case PHI is arbitrarily set to zero and PSI is the sum (PHI+PSI).

See also

SO3.toeul, SE3.torpy, eul2tr, tr2rpy

SE3.torpy

Convert to roll-pitch-yaw angles

`RPY = P.torpy(options)` are the roll-pitch-yaw angles (1×3) corresponding to the rotational part of the SE3 object P. The 3 angles `RPY`=[R,P,Y] correspond to sequential rotations about the Z, Y and X axes respectively.

If P is a vector ($1 \times N$) then each row of `RPY` corresponds to an element of the vector.

Options

'deg'	Compute angles in degrees (radians default)
'xyz'	Return solution for sequential rotations about X, Y, Z axes
'yxz'	Return solution for sequential rotations about Y, X, Z axes

Notes

- There is a singularity for the case where P=$\pi/2$ in which case R is arbitrarily set to zero and Y is the sum (R+Y).

See also

SE3.torpy, SE3.toeul, rpy2tr, tr2eul

SE3.transl

Get translation vector

`T = P.transl()` is the translational part of **SE3** object as a 3-element row vector. If P is a vector ($1 \times N$) then

the rows of `T` ($M \times 3$) are the translational component of the

corresponding pose in the sequence.

`[X,Y,Z] = P.transl()` as above but the translational part is returned as three components. If P is a vector ($1 \times N$) then `X`,`Y`,`Z` ($1 \times N$) are the translational components of the corresponding pose in the sequence.

Notes

- The .t method only works for a single pose object, on a vector it returns a comma-separated list.

See also

SE3.t, transl

SE3.trnorm

Normalize rotation submatrix (compatibility)

`T = trnorm(P)` is an **SE3** object equivalent to `P` but normalized (rotation matrix guaranteed to be orthogonal).

Notes

- Overrides the classic RTB function trnorm for an SE3 object.

See also

trnorm

SE3.tv

Return translation for a vector of SE3 objects

`P.tv` is a column vector (3×1) representing the translational part of the SE3 pose object `P`. If `P` is a vector of SE3 objects ($N \times 1$) then the result is a matrix ($3 \times N$) with columns corresponding to the elements of `P`.

See also

SE3.t

SE3.Twist

Convert to Twist object

`TW = P.Twist()` is the equivalent Twist object. The elements of the twist are the unique elements of the Lie algebra of the SE3 object P.

See also

SE3.logs, Twist

SE3.velxform

Velocity transformation

Transform velocity between frames. A is the world frame, B is the body frame and C is another frame attached to the body. PAB is the pose of the body frame with respect to the world frame, PCB is the pose of the body frame with respect to frame C.

`J = PAB.velxform()` is a 6×6 Jacobian matrix that maps velocity from frame B to frame A.

`J = PCB.velxform('samebody')` is a 6×6 Jacobian matrix that maps velocity from frame C to frame B. This is also the adjoint of PCB.

skew

Create skew-symmetric matrix

`S = SKEW(V)` is a skew-symmetric matrix formed from `V`.

If `V` (1×1) then `S` =

```
| 0   -v |
| v    0 |
```

and if `V` (1×3) then `S` =

```
|  0   -vz   vy |
| vz     0  -vx |
|-vy    vx    0 |
```

Notes

- This is the inverse of the function VEX().
- These are the generator matrices for the Lie algebras so(2) and so(3).

References

- Robotics, Vision & Control: Second Edition, Chap 2, P. Corke, Springer 2016.

See also

skewa, vex

skewa

Create augmented skew-symmetric matrix

`S = SKEWA(V)` is an augmented skew-symmetric matrix formed from `V`.

If `V` (1×3) then `S` =

```
|  0   -v3  v1 |
| v3     0  v2 |
|  0     0   0 |
```

and if `V` (1×6) then `S` =

```
|  0   -v6   v5  v1 |
| v6     0  -v4  v2 |
|-v5    v4    0  v3 |
|  0     0    0   0 |
```

Notes

- This is the inverse of the function VEXA().
- These are the generator matrices for the Lie algebras se(2) and se(3).
- Map twist vectors in 2D and 3D space to se(2) and se(3).

References

- Robotics, Vision & Control: Second Edition, Chap 2, P. Corke, Springer 2016.

See also
skew, vex, Twist

SO2

Representation of 2D rotation
This subclasss of RTBPose is an object that represents rotation in 2D. Internally this is a 2×2 orthonormal matrix belonging to the group $\mathrm{SO}(2)$.

Constructor methods

SO2	general constructor
SO2.exp	exponentiate an so(2) matrix
SO2.rand	random orientation
new	new SO2 object from instance

Display and print methods

animate	^graphically animate coordinate frame for pose
display	^print the pose in human readable matrix form
plot	^graphically display coordinate frame for pose
print	^print the pose in single line format

Group operations

$*$	^mtimes: multiplication (group operator, transform point)
/	^mrdivide: multiply by inverse
^	^mpower: exponentiate (integer only)
inv	^inverse rotation
prod	^product of elements

Methods

det	determinant of matrix value (is 1)
eig	^eigenvalues of matrix value
interp	interpolate between rotations
log	logarithm of rotation matrix
simplify	^apply symbolic simplication to all elements
subs	^symbolic substitution
vpa	^symbolic variable precision arithmetic

Information and test methods

dim	^returns 2
isSE	^returns false
issym	^test if rotation matrix has symbolic elements
SO2.isa	test if matrix is $\mathrm{SO}(2)$

Conversion methods

char	^convert to human readable matrix as a string

SO2.convert	convert SO2 object or SO(2) matrix to SO2 object
double	^convert to rotation matrix
theta	rotation angle
R	convert to rotation matrix
SE2	convert to SE2 object with zero translation
T	convert to homogeneous transformation matrix with zero translation

Compatibility methods

ishomog2	^returns false
isrot2	^returns true
tranimate2	^animate coordinate frame
trplot2	^plot coordinate frame
trprint2	^print single line representation

Operators

+	^plus: elementwise addition, result is a matrix
-	^minus: elementwise subtraction, result is a matrix
==	^eq: test equality
∼=	^ne: test inequality

^inherited from RTBPose class.

See also

SE2, SO3, SE3, RTBPose

SO2.SO2

Construct SO2 object

`P = SO2()` is the identity element, a null rotation.

`P = SO2(THETA)` is an **SO2** object representing rotation of `THETA` radians. If `THETA` is a vector (N) then `P` is a vector of objects, corresponding to the elements of `THETA`.

`P = SO2(THETA, 'deg')` as above but with `THETA` degrees.

`P = SO2(R)` is an **SO2** object formed from the rotation matrix `R` (2×2).

`P = SO2(T)` is an **SO2** object formed from the rotational part of the homogeneous transformation matrix `T` (3×3).

`P = SO2(Q)` is an **SO2** object that is a copy of the **SO2** object `Q`.

Notes

- For matrix arguments `R` or `T` the rotation submatrix is checked for validity.

See also

rot2, SE2, SO3

SO2.angle

Rotation angle

`P.angle()` is the rotation angle, in radians $[-\pi, \pi)$, associated with the SO2 object P.

See also

atan2

SO2.char

Convert to string

`P.char()` is a string containing rotation matrix elements.

See also

RTB.display

SO2.convert

Convert value to SO2

`Q = SO2.convert(X)` is an **SO2** object equivalent to `X` where `X` is either an SO2 object, an SO(2) rotation matrix (2×2), an SE2 object, or an SE(2) homogeneous transformation matrix (3×3).

SO2.det

Determinant

`det(P)` is the determinant of the **SO2** object `P` and should always be +1.

SO2.eig

Eigenvalues and eigenvectors

`E = eig(P)` is a column vector containing the eigenvalues of the underlying rotation matrix.

`[V,D] = eig(P)` produces a diagonal matrix `D` of eigenvalues and a full matrix `V` whose columns are the corresponding eigenvectors such that A*`V` = `V`*`D`.

See also

eig

SO2.exp

Construct SO2 from Lie algebra

`R = SO3.exp(X)` is the **SO2** rotation corresponding to the so(2) Lie algebra element SIGMA (2×2).

`R = SO3.exp(TW)` as above but the Lie algebra is represented as a twist vector `TW` (1×1).

Notes

- `TW` is the non-zero elements of `X`.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p25-31.

See also

trexp2, skewa

SO2.interp

Interpolate between rotations

`P1.interp(P2, s)` is an **SO2** object representing interpolation between rotations represented by SO2 objects P1 and `P2`. `s` varies from 0 (P1) to 1 (`P2`). If `s` is a vector ($1 \times N$) then the result will be a vector of SO2 objects.

`P1.interp(P2,N)` as above but returns a vector ($1 \times$ `N`) of **SO2** objects interpolated between P1 and `P2` in `N` steps.

Notes

- It is an error if any element of S is outside the interval 0 to 1.

See also

SO2.angle

SO2.inv

Inverse

`Q = inv(P)` is an **SO2** object representing the inverse of the **SO2** object `P`.

Notes

- This is a group operator: input and output in the SO(2) group.
- This is simply the transpose of the underlying matrix.
- `P*Q` will be the identity group element (zero rotation, identity matrix).

SO2.isa

Test if matrix belongs to SO(2)

`SO2.ISA(T)` is true (1) if the argument `T` is of dimension 2×2 or $2 \times 2 \times N$, else false (0).

`SO2.ISA(T, true)` as above, but also checks the validity of the rotation matrix, ie. that its determinant is +1.

Notes

- The first form is a fast, but incomplete, test for a transform in SO(2).

See also

SO3.ISA, SE2.ISA, SE2.ISA, ishomog2

SO2.log

Logarithm

`so2 = P.log()` is the Lie algebra corresponding to the **SO2** object P. It is a skew-symmetric matrix (2×2).

See also

SO2.exp, Twist, logm, vex, skew

SO2.new

Construct a new object of the same type

Create a new object of the same type as the RTBPose derived instance object.

`P.new(X)` creates a new object of the same type as P, by invoking the **SO2** constructor on the matrix `X` (2×2).

`P.new()` as above but assumes an identity matrix.

Notes

- Serves as a dynamic constructor.

- This method is polymorphic across all RTBPose derived classes, and

allows easy creation of a new object of the same class as an existing one without needing to explicitly determine its type.

See also

SE3.new, SO3.new, SE2.new

SO2.R

Get rotation matrix

`R = P.R()` is the rotation matrix (2×2) associated with the **SO2** object P. If P is a vector ($1 \times N$) then `R` ($2 \times 2 \times N$) is a stack of rotation matrices, with the third dimension corresponding to the index of P.

See also

SO2.T

SO2.rand

Construct a random SO(2) object

`SO2.rand()` is an **SO2** object where the angle is drawn from a uniform random orientation. Random numbers are in the interval 0 to 2π.

See also

rand

SO2.SE2

Convert to SE2 object

`P.SE2()` is an SE2 object formed from the rotational component of the SO2 object P and with a zero translational component.

See also

SE2

SO2.T

Get homogeneous transformation matrix

`T = P.T()` is the homogeneous transformation matrix (3×3) associated with the SO2 object P, and has zero translational component. If P is a vector ($1 \times N$) then `T` ($3 \times 3 \times N$) is a stack of rotation matrices, with the third dimension corresponding to the index of P.

See also

SO2.T

SO2.theta

Rotation angle

`P.theta()` is the rotation angle, in radians, associated with the SO2 object P.

Notes

- Deprecated, use angle() instead.

See also

SO2.angle

SO3

Representation of 3D rotation

This subclasss of RTBPose is an object that represents rotation in 3D. Internally this is a 3×3 orthonormal matrix belonging to the group SO(3).

Constructor methods

SO3	general constructor
SO3.exp	exponentiate an so(3) matrix
SO3.angvec	rotation about vector
SO3.eul	rotation defined by Euler angles
SO3.oa	rotation defined by o- and a-vectors
SO3.Rx	rotation about x-axis
SO3.Ry	rotation about y-axis
SO3.Rz	rotation about z-axis
SO3.rand	random orientation
SO3.rpy	rotation defined by roll-pitch-yaw angles
new	new SO3 object from instance

Display and print methods

plot	^graphically display coordinate frame for pose
animate	^graphically animate coordinate frame for pose
print	^print the pose in single line format
display	^print the pose in human readable matrix form

Group operations

*	^mtimes: multiplication (group operator, transform point)

.*	times: multiplication (group operator) followed by normalization
/	^mrdivide: multiply by inverse
./	rdivide: multiply by inverse followed by normalization
^	^mpower: exponentiate (integer only)
.^	power: exponentiate followed by normalization
inv	^inverse rotation
prod	^product of elements

Methods

det	determinant of matrix value (is 1)
eig	eigenvalues of matrix value
interp	interpolate between rotations
log	logarithm of matrix value
norm	normalize matrix
simplify	^apply symbolic simplication to all elements
subs	^symbolic substitution
vpa	^symbolic variable precision arithmetic

Information and test methods

dim	^returns 3
isSE	^returns false
issym	^test if rotation matrix has symbolic elements
SO3.isa	test if matrix is SO(3)

Conversion methods

char	^convert to human readable matrix as a string
SO3.convert	convert SO3 object or SO(3) matrix to SO3 object
double	convert to rotation matrix

R	convert to rotation matrix
SE3	convert to SE3 object with zero translation
T	convert to homogeneous transformation matrix with zero translation
toangvec	convert to rotation about vector form
toeul	convert to Euler angles
torpy	convert to roll-pitch-yaw angles
UnitQuaternion	convert to UnitQuaternion object

Compatibility methods

isrot	^returns true
ishomog	^returns false
trprint	^print single line representation
trplot	^plot coordinate frame
tranimate	^animate coordinate frame
tr2eul	convert to Euler angles
tr2rpy	convert to roll-pitch-yaw angles
trnorm	normalize rotation matrix

Operators

+	^plus: elementwise addition, result is a matrix
-	^minus: elementwise subtraction, result is a matrix
==	^eq: test equality
∼=	^ne: test inequality

^inherited from RTBPose class.

Properties

n	normal (x) vector
o	orientation (y) vector
a	approach (z) vector

See also

SE2, SO2, SE3, RTBPose

SO3.SO3

Construct SO3 object

`P = SO3()` is the identity element, a null rotation.

`P = SO3(R)` is an **SO3** object formed from the rotation matrix `R` (3×3).

`P = SO3(T)` is an **SO3** object formed from the rotational part of the homogeneous transformation matrix `T` (4×4).

`P = SO3(Q)` is an **SO3** object that is a copy of the **SO3** object `Q`.

Notes

- For matrix arguments `R` or `T` the rotation submatrix is checked for validity.

See also

SE3, SO2

SO3.angdist

Distance metric

`R1.angdist(R2)` is a distance metric between two SO(3) rotation matrices.

`R1.angdist(R2, M)` as above but uses the method specified by `M`

`M` = 1 (default): || I - R1 `R2`'|| in [0,2]

M = 2: || log R1 `R2`'|| in [0,π]

See also: trlog, UnitQuaternion.angdist

SO3.angvec

Construct SO3 from angle and axis vector

`R = SO3.angvec(THETA, V)` is an **SO3** object representitng a rotation of `THETA` about the vector `V`.

Notes

- If `THETA` == 0 then return null group element (zero rotation, identity matrix).
- If `THETA` $\neq$ 0 then `V` must have a finite length, does not have to be unit length.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p41-42.

See also

SE3.angvec, eul2r, rpy2r, tr2angvec

SO3.convert

Convert value to SO3

`Q = SO3.convert(X)` is an **SO3** object equivalent to `X` where `X` is either an SO3 object, an SO(3) rotation matrix (3×3), an SE3 object, or an SE(3) homogeneous transformation matrix (4×4).

SO3.det

Determinant

`det(P)` is the determinant of the **SO3** object `P` and should always be +1.

SO3.eig

Eigenvalues and eigenvectors

`E = eig(P)` is a column vector containing the eigenvalues of the underlying rotation matrix.

`[V,D] = eig(P)` produces a diagonal matrix `D` of eigenvalues and a full matrix `V` whose columns are the corresponding eigenvectors such that A*`V` = `V`*`D`.

See also

eig

SO3.eul

Construct SO3 from Euler angles

`P = SO3.eul(PHI, THETA, PSI, OPTIONS)` is an **SO3** object equivalent to the specified Euler angles. These correspond to rotations about the Z, Y, Z axes respectively. If `PHI`, `THETA`, `PSI` are column vectors ($N \times 1$) then they are assumed to represent a trajectory then `P` is a vector ($1 \times N$) of SO3 objects.

`P = SO3.eul(EUL, OPTIONS)` as above but the Euler angles are taken from consecutive columns of the passed matrix `EUL` = [`PHI THETA PSI`]. If `EUL` is a matrix ($N \times 3$) then they are assumed to represent a trajectory then `P` is a vector ($1 \times N$) of SO3 objects.

Options

'deg'	Angles are specified in degrees (default radians)

Note

- The vectors `PHI`, `THETA`, `PSI` must be of the same length.

Reference

- Robotics, Vision & Control: Second Edition, `P`. Corke, Springer 2016; p36-37.

See also

SO3.rpy, SE3.eul, eul2tr, rpy2tr, tr2eul

SO3.exp

Construct SO3 from Lie algebra

`R = SO3.exp(X)` is the **SO3** rotation corresponding to the so(3) Lie algebra element SIGMA (3×3).

`R = SO3.exp(TW)` as above but the Lie algebra is represented as a twist vector `TW` (3×1).

Notes

- `TW` is the non-zero elements of `X`.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p42-43.

See also

trexp, skew

SO3.get.a

Get approach vector

`P.a` is the approach vector (3×1), the third column of the rotation matrix, which is the z-axis unit vector.

See also

SO3.n, SO3.o

SO3.get.n

Get normal vector

`P.n` is the normal vector (3×1), the first column of the rotation matrix, which is the x-axis unit vector.

See also

SO3.o, SO3.a

SO3.get.o

Get orientation vector

`P.o` is the orientation vector (3×1), the second column of the rotation matrix, which is the y-axis unit vector..

See also

SO3.n, SO3.a

SO3.interp

Interpolate between rotations

`P1.interp(P2, s)` is an **SO3** object representing a slerp interpolation between rotations represented by SO3 objects P1 and `P2`. `s` varies from 0 (P1) to 1 (`P2`). If `s` is a vector ($1 \times N$) then the result will be a vector of SO3 objects.

`P1.interp(P2,N)` as above but returns a vector ($1 \times$ `N`) of **SO3** objects interpolated between P1 and `P2` in `N` steps.

Notes

- It is an error if any element of S is outside the interval 0 to 1.

See also

UnitQuaternion

SO3.inv

Inverse

`Q = inv(P)` is an **SO3** object representing the inverse of the **SO3** object `P`.

Notes

- This is a group operator: input and output in the SO(3) group.
- This is simply the transpose of the underlying matrix.
- `P*Q` will be the identity group element (zero rotation, identity matrix).

SO3.isa

Test if a rotation matrix

`SO3.ISA(R)` is true (1) if the argument is of dimension 3×3 or $3 \times 3 \times N$, else false (0).

`SO3.ISA(R, 'valid')` as above, but also checks the validity of the rotation matrix, ie. that its determinant is +1.

Notes

- The first form is a fast, but incomplete, test for a rotation in SO(3).

See also

SE3.ISA, SE2.ISA, SO2.ISA

SO3.log

Logarithm

`P.log()` is the Lie algebra corresponding to the **SO3** object P. It is a skew-symmetric matrix (3×3).

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p42-43.

See also

SO3.exp, Twist, trlog, skew, vex

SO3.new

Construct a new object of the same type

Create a new object of the same type as the RTBPose derived instance object.

`P.new(X)` creates a new object of the same type as P, by invoking the **SO3** constructor on the matrix `X` (3×3).

`P.new()` as above but assumes an identity matrix.

Notes

- Serves as a dynamic constructor.
- This method is polymorphic across all RTBPose derived classes, and allows easy creation of a new object of the same class as an existing
- one without needing to explicitly determine its type.

See also
SE3.new, SO2.new, SE2.new

SO3.norm

Normalize rotation
`P.norm()` is an **SO3** object equivalent to P but with a rotation matrix guaranteed to be orthogonal.

Notes
- Overrides the classic RTB function trnorm for an SO3 object.

See also
trnorm

SO3.oa

Construct SO3 from orientation and approach vectors
`P = SO3.oa(O, A)` is an **SO3** object for the specified orientation and approach vectors (3×1) formed from 3 vectors such that R = [N `O` `A`] and N = `O` x `A`.

Notes
- The rotation matrix is guaranteed to be orthonormal so long as `O` and `A`

are not parallel.

- The vectors `O` and `A` are parallel to the Y- and Z-axes of the coordinate

frame.

References
- Robot manipulators: mathematis, programming and control Richard Paul, MIT Press, 1981.
- Robotics, Vision & Control: Second Edition, `P`. Corke, Springer 2016; p40-41.

SO3.R

Get rotation matrix
`R = P.R()` is the rotation matrix (3×3) associated with the **SO3** object P. If P is a vector ($1 \times N$) then `R` ($3 \times 3 \times N$) is a stack of rotation matrices, with the third dimension corresponding to the index of P.

See also
SO3.T

SO3.rand

Construct random SO3
`SO3.rand()` is an **SO3** object with a random orientation drawn from a uniform distribution.

See also

rand, UnitQuaternion.rand

SO3.rdivide

Compose SO3 object with inverse and normalize

`P ./ Q` is an **SO3** object representing the composition of **SO3** object `P` by the inverse of SO3 object `Q`. This is matrix multiplication of their orthonormal rotation matrices followed by normalization.

If either, or both, of P1 or P2 are vectors, then the result is a vector.

- if P1 is a vector ($1 \times N$) then R is a vector ($1 \times N$) such that R(i) = P1(i).*P2.
- if P2 is a vector ($1 \times N$) then R is a vector ($1 \times N$) such that R(i) = P1.*P2(i).
- if both P1 and P2 are vectors ($1 \times N$) then R is a vector ($1 \times N$) such that R(i) = P1(i).*P2(i).

Notes

- Overloaded operator './'.
- This is a group operator: `P`, `Q` and result all belong to the SO(3) group.

See also

SO3.mrdivide, SO3.times, trnorm

SO3.rpy

Construct SO3 from roll-pitch-yaw angles

`P = SO3.rpy(ROLL, PITCH, YAW, OPTIONS)` is an **SO3** object equivalent to the specified roll, pitch, yaw angles angles. These correspond to rotations about the Z, Y, X axes respectively. If `ROLL`, `PITCH`, `YAW` are column vectors ($N \times 1$) then they are assumed to represent a trajectory then `P` is a vector ($1 \times N$) of SO3 objects.

`P = SO3.rpy(RPY, OPTIONS)` as above but the roll, pitch, yaw angles angles angles are taken from consecutive columns of the passed matrix `RPY` = [`ROLL`, `PITCH`, `YAW`]. If `RPY` is a matrix ($N \times 3$) then they are assumed to represent a trajectory and `P` is a vector ($1 \times N$) of SO3 objects.

Options

'deg'	Compute angles in degrees (radians default)
'xyz'	Rotations about X, Y, Z axes (for a robot gripper)
'yxz'	Rotations about Y, X, Z axes (for a camera)

Reference

- Robotics, Vision & Control: Second Edition, `P`. Corke, Springer 2016; p37-38

See also

SO3.eul, SE3.rpy, tr2rpy, eul2tr

SO3.Rx

Construct SO3 from rotation about X axis

`P = SO3.Rx(THETA)` is an **SO3** object representing a rotation of `THETA` radians about the x-axis. If the `THETA` is a vector ($1 \times N$) then `P` will be a vector ($1 \times N$) of corresponding SO3 objects.

`P = SO3.Rx(THETA, 'deg')` as above but `THETA` is in degrees.

See also

SO3.Ry, SO3.Rz, rotx

SO3.Ry

Construct SO3 from rotation about Y axis

`P = SO3.Ry(THETA)` is an **SO3** object representing a rotation of `THETA` radians about the y-axis. If the `THETA` is a vector ($1 \times N$) then `P` will be a vector ($1 \times N$) of corresponding SO3 objects.

`P = SO3.Ry(THETA, 'deg')` as above but `THETA` is in degrees.

See also

SO3.Rx, SO3.Rz, roty

SO3.Rz

Construct SO3 from rotation about Z axis

`P = SO3.Rz(THETA)` is an **SO3** object representing a rotation of `THETA` radians about the z-axis. If the `THETA` is a vector ($1 \times N$) then `P` will be a vector ($1 \times N$) of corresponding SO3 objects.

`P = SO3.Rz(THETA, 'deg')` as above but `THETA` is in degrees.

See also

SO3.Rx, SO3.Ry, rotz

SO3.SE3

Convert to SE3 object

`Q = P.SE3()` is an SE3 object with a rotational component given by the SO3 object P, and with a zero translational component. If P is a vector of SO3 objects then `Q` will a same length vector of SE3 objects.

See also

SE3

SO3.T

Get homogeneous transformation matrix

`T = P.T()` is the homogeneous transformation matrix (4×4) associated with the SO3 object P, and has zero translational component. If P is a vector ($1 \times N$) then `T` ($4 \times 4 \times N$) is a stack of rotation matrices, with the third dimension corresponding to the index of P.

See also

SO3.T

SO3.times

Compose SO3 objects and normalize

`R = P1 .* P2` is an **SO3** object representing the composition of the two rotations described by the SO3 objects `P1` and `P2`. This is matrix multiplication of their orthonormal rotation matrices followed by normalization.

If either, or both, of `P1` or `P2` are vectors, then the result is a vector.

- if `P1` is a vector ($1 \times N$) then `R` is a vector ($1 \times N$) such that `R`(i) = `P1`(i).*`P2`.
- if `P2` is a vector ($1 \times N$) then `R` is a vector ($1 \times N$) such that `R`(i) = `P1`.*`P2`(i).
- if both `P1` and `P2` are vectors ($1 \times N$) then `R` is a vector ($1 \times N$) such that `R`(i) = `P1`(i).*`P2`(i).

Notes

- Overloaded operator '.*'.
- This is a group operator: P, Q and result all belong to the SO(3) group.

See also

RTBPose.mtimes, SO3.divide, trnorm

SO3.toangvec

Convert to angle-vector form

`[THETA,V] = P.toangvec(OPTIONS)` is rotation expressed in terms of an angle `THETA` about the axis `V` (1×3) equivalent to the rotational part of the SO3 object P.

If P is a vector ($1 \times N$) then `THETA` ($N \times 1$) is a vector of angles for corresponding elements of the vector and `V` ($N \times 3$) are the corresponding axes, one per row.

Options

'deg'	Return angle in degrees (default radians)

Notes

- If no output arguments are specified the result is displayed.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p41-42.

See also

angvec2r, angvec2tr, trlog

SO3.toeul

Convert to Euler angles

`EUL = P.toeul(OPTIONS)` are the ZYZ Euler angles (1×3) corresponding to the rotational part of the SO3 object P. The three angles `EUL`=[PHI,THETA,PSI] correspond to sequential rotations about the Z, Y and Z axes respectively.

If P is a vector ($1 \times N$) then each row of `EUL` corresponds to an element of the vector.

Options

'deg'	Compute angles in degrees (default radians)
'flip'	Choose PHI to be in quadrant 2 or 3.

Notes

- There is a singularity when THETA=0 in which case PHI is arbitrarily set to zero and PSI is the sum (PHI+PSI).

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p36-37.

See also

SO3.torpy, eul2tr, tr2rpy

SO3.torpy

Convert to roll-pitch-yaw angles

`RPY = P.torpy(options)` are the roll-pitch-yaw angles (1×3) corresponding to the rotational part of the SO3 object P. The 3 angles `RPY`=[ROLL,PITCH,YAW] correspond to sequential rotations about the Z, Y and X axes respectively.

If P is a vector ($1 \times N$) then each row of `RPY` corresponds to an element of the vector.

Options

'deg'	Compute angles in degrees (default radians)
'xyz'	Return solution for sequential rotations about X, Y, Z axes
'yxz'	Return solution for sequential rotations about Y, X, Z axes

Notes

- There is a singularity for the case where PITCH=$\pi/2$ in which case ROLL is arbitrarily set to zero and YAW is the sum (ROLL+YAW).

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p37-38.

See also

SO3.toeul, rpy2tr, tr2eul

SO3.tr2eul

Convert to Euler angles (compatibility)

`tr2eul(P, OPTIONS)` is a vector (1×3) of ZYZ Euler angles equivalent to the rotation `P` (SO3 object).

Notes

- Overrides the classic RTB function tr2eul for an SO3 object.
- All the options of tr2eul apply.

See also

tr2eul

SO3.tr2rpy

Convert to RPY angles (compatibility)

`tr2rpy(P, OPTIONS)` is a vector (1×3) of roll-pitch-yaw angles equivalent to the rotation `P` (SO3 object).

Notes

- Overrides the classic RTB function tr2rpy for an SO3 object.
- All the options of tr2rpy apply.
- Defaults to ZYX order.

See also

tr2rpy

SO3.trnorm

Normalize rotation (compatibility)

`trnorm(P)` is an **SO3** object equivalent to `P` but with a rotation matrix guaranteed to be orthogonal.

Notes

- Overrides the classic RTB function trnorm for an SO3 object.

See also

trnorm

SO3.UnitQuaternion

Convert to UnitQuaternion object

`P.UnitQuaternion()` is a UnitQuaternion object equivalent to the rotation described by the SO3 object P.

See also

UnitQuaternion

SpatialAcceleration

Spatial acceleration class

Concrete subclass of SpatialM6 and represents the translational and rotational acceleration of a rigid-body moving in 3D space.

```
SpatialVec6 (abstract handle class)
  |
  +--- SpatialM6 (abstract)
  |     |
  |     +---SpatialVelocity
  |     +---SpatialAcceleration
  |
  +---SpatialF6 (abstract)
       |
       +---SpatialForce
       +---SpatialMomentum
```

Methods

SpatialAcceleration	^constructor invoked by subclasses
char	^convert to string
cross	^^cross product
display	^display in human readable form
double	^convert to a $6 \times N$ double
new	construct new concrete class of same type

Operators

+	^add spatial vectors of the same type
-	^subtract spatial vectors of the same type
-	^unary minus of spatial vectors
*	^^^premultiplication by SpatialInertia yields SpatialForce
*	^^^^premultiplication by Twist yields transformed SpatialAcceleration

Notes:

- ^is inherited from SpatialVec6.
- ^^is inherited from SpatialM6.
- ^^^are implemented in SpatialInertia.
- ^^^^are implemented in Twist.

References

- Robot Dynamics Algorithms, R. Featherstone, volume 22, Springer International Series in Engineering and Computer Science,
- Springer, 1987.
- A beginner's guide to 6-d vectors (part 1), R. Featherstone, IEEE Robotics Automation Magazine, 17(3):83-94, Sep. 2010.

SpatialAcceleration.new

Construct a new object of the same type

`A2 = A.new(X)` creates a new object of the same type as A, with the value `X` (6×1).

Notes

- Serves as a dynamic constructor.
- This method is polymorphic across all SpatialVec6 derived classes, and allows easy creation of a new object of the same class as an existing
- one without needing to explicitly determine its type.

SpatialF6

Abstract spatial force class

Abstract superclass that represents spatial force. This class has two concrete subclasses:

```
SpatialVec6 (abstract handle class)
  |
  +--- SpatialM6 (abstract)
  |     |
  |     +---SpatialVelocity
  |     +---SpatialAcceleration
  |
  +---SpatialF6 (abstract)
       |
       +---SpatialForce
       +---SpatialMomentum
```

Methods

SpatialF6	^constructor invoked by subclasses
char	^convert to string
display	^display in human readable form
double	^convert to a $6 \times N$ double

Operators

+	^add spatial vectors of the same type
-	^subtract spatial vectors of the same type
-	^unary minus of spatial vectors

Notes:

- ^is inherited from SpatialVec6.
- Subclass of the MATLAB handle class which means that pass by reference semantics apply.
- Spatial vectors can be placed into arrays and indexed.

References

- Robot Dynamics Algorithms, R. Featherstone, volume 22, Springer International Series in Engineering and Computer Science,
- Springer, 1987.
- A beginner's guide to 6-d vectors (part 1), R. Featherstone, IEEE Robotics Automation Magazine, 17(3):83-94, Sep. 2010.

See also

SpatialForce, SpatialMomentum, SpatialInertia, SpatialM6

SpatialForce

Spatial force class

Concrete subclass of SpatialF6 and represents the translational and rotational forces and torques acting on a rigid-body in 3D space.

```
SpatialVec6 (abstract handle class)
  |
  +--- SpatialM6 (abstract)
  |     |
  |     +---SpatialVelocity
  |     +---SpatialAcceleration
  |
  +---SpatialF6 (abstract)
       |
       +---SpatialForce
       +---SpatialMomentum
```

Methods

SpatialForce	^constructor invoked by subclasses
char	^convert to string
display	^display in human readable form
double	^convert to a $6 \times N$ double
new	construct new concrete class of same type

Operators

+	^add spatial vectors of the same type
-	^subtract spatial vectors of the same type
-	^unary minus of spatial vectors
$*$	^^^premultiplication by SE3 yields transformed SpatialForce
$*$	^^^^premultiplication by Twist yields transformed SpatialForce

Notes:

- ^is inherited from SpatialVec6.
- ^^is inherited from SpatialM6.
- ^^^are implemented in RTBPose.
- ^^^^are implemented in Twist.

References

- Robot Dynamics Algorithms, R. Featherstone, volume 22, Springer International Series in Engineering and Computer Science,
- Springer, 1987.
- A beginner's guide to 6-d vectors (part 1), R. Featherstone, IEEE Robotics Automation Magazine, 17(3):83-94, Sep. 2010.

See also

SpatialVec6, SpatialF6, SpatialMomentum

SpatialForce.new

Construct a new object of the same type

`A2 = A.new(X)` creates a new object of the same type as A, with the value `X` (6×1).

Notes

- Serves as a dynamic constructor.
- This method is polymorphic across all SpatialVec6 derived classes, and allows easy creation of a new object of the same class as an existing
- one without needing to explicitly determine its type.

SpatialInertia

Spatial inertia class

Concrete class representing spatial inertia.

Methods

SpatialInertia	constructor
char	convert to string
display	display in human readable form
double	convert to a $6 \times N$ double

Operators

+	plus: add spatial inertia of connected bodies
*	mtimes: compute force or momentum

Notes

- Subclass of the MATLAB handle class which means that pass by reference semantics apply.
- Spatial inertias can be placed into arrays and indexed.

References

- Robot Dynamics Algorithms, R. Featherstone, volume 22, Springer International Series in Engineering and Computer Science,
- Springer, 1987.

- A beginner's guide to 6-d vectors (part 1), R. Featherstone, IEEE Robotics Automation Magazine, 17(3):83-94, Sep. 2010.

See also SpatialM6, SpatialF6, SpatialVelocity, SpatialAcceleration, SpatialForce, SpatialMomentum.

SpatialInertia.SpatialInertia

Constructor

`SI = SpatialInertia(M, C, I)` is a spatial inertia object for a rigid-body with mass `M`, centre of mass at `C` relative to the link frame, and an inertia matrix (3×3) about the centre of mass.

`SI = SpatialInertia(I)` is a spatial inertia object with a value equal to `I` (6×6).

SpatialInertia.char

Convert to string

`s = SI.char()` is a string showing spatial inertia parameters in a compact format. If SI is an array of spatial inertia objects return a string with the inertia values in a vertical list.

See also

SpatialInertia.display

SpatialInertia.display

Display parameters

`SI.display()` displays the spatial inertia parameters in compact format. If SI is an array of spatial inertia objects it displays them in a vertical list.

Notes

- This method is invoked implicitly at the command line when the result of an expression is a spatial inerita object and the command has
- no trailing semicolon.

See also

SpatialInertia.char

SpatialInertia.double

Convert to matrix

`double(V)` is a native matrix (6×6) with the value of the spatial inertia. If `V` is an array $(1 \times N)$ the result is a matrix $(6 \times 6 \times N)$.

SpatialInertia.mtimes

Multiplication operator

`SI * A` is the SpatialForce required for a body with **SpatialInertia** `SI` to accelerate with the SpatialAcceleration `A`.

`SI * V` is the SpatialMomemtum of a body with **SpatialInertia** `SI` and SpatialVelocity `V`.

Notes

- These products must be written in this order, `A*SI` and `V*SI` are not defined.

SpatialInertia.plus

Addition operator

`SI1 + SI2` is the **SpatialInertia** of a composite body when bodies with **SpatialInertia** `SI1` and `SI2` are connected.

SpatialM6

Abstract spatial motion class

Abstract superclass that represents spatial motion. This class has two concrete subclasses:

```
SpatialVec6 (abstract handle class)
  |
  +--- SpatialM6 (abstract)
  |     |
  |     +---SpatialVelocity
  |     +---SpatialAcceleration
  |
  +---SpatialF6 (abstract)
       |
       +---SpatialForce
       +---SpatialMomentum
```

Methods

SpatialM6	^constructor invoked by subclasses
char	^convert to string
cross	cross product
display	^display in human readable form
double	^convert to a $6 \times N$ double

Operators

+	^add spatial vectors of the same type
-	^subtract spatial vectors of the same type
-	^unary minus of spatial vectors

Notes:

- ^is inherited from SpatialVec6.
- Subclass of the MATLAB handle class which means that pass by reference semantics apply.
- Spatial vectors can be placed into arrays and indexed.

References

- Robot Dynamics Algorithms, R. Featherstone, volume 22, Springer International Series in Engineering and Computer Science,
- Springer, 1987.
- A beginner's guide to 6-d vectors (part 1), R. Featherstone, IEEE Robotics Automation Magazine, 17(3):83-94, Sep. 2010.

See also

SpatialForce, SpatialMomentum, SpatialInertia, SpatialM6

SpatialM6.cross

Spatial velocity cross product

`cross(V1, V2)` is a SpatialAcceleration object where `V1` and `V2` are **SpatialM6** subclass instances.

`cross(V, F)` is a SpatialForce object where `V1` is a **SpatialM6** subclass instances and `F` is a SpatialForce subclass instance.

Notes

- The first form is Featherstone's "x" operator.
- The second form is Featherstone's "x*" operator.

SpatialMomentum

Spatial momentum class

Concrete subclass of SpatialF6 and represents the translational and rotational momentum of a rigid-body moving in 3D space.

```
SpatialVec6 (abstract handle class)
  |
  +--- SpatialM6 (abstract)
  |      |
  |      +---SpatialVelocity
  |      +---SpatialAcceleration
  |
  +---SpatialF6 (abstract)
       |
       +---SpatialForce
       +---SpatialMomentum
```

Methods

SpatialMomentum	^constructor invoked by subclasses
new	construct new concrete class of same type
double	^convert to a $6 \times N$ double
char	^convert to string
cross	^^cross product
display	^display in human readable form

Operators

+	^add spatial vectors of the same type
-	^subtract spatial vectors of the same type
-	^unary minus of spatial vectors

Notes:

- ^is inherited from SpatialVec6.
- ^^is inherited from SpatialM6.

References

- Robot Dynamics Algorithms, R. Featherstone, volume 22, Springer International Series in Engineering and Computer Science,
- Springer, 1987.
- A beginner's guide to 6-d vectors (part 1), R. Featherstone, IEEE Robotics Automation Magazine, 17(3):83-94, Sep. 2010.

See also

SpatialVec6, SpatialF6, SpatialForce

SpatialMomentum.new

Construct a new object of the same type

`A2 = A.new(X)` creates a new object of the same type as A, with the value `X` (6×1).

Notes

- Serves as a dynamic constructor.
- This method is polymorphic across all SpatialVec6 derived classes, and allows easy creation of a new object of the same class as an existing
- one without needing to explicitly determine its type.

SpatialVec6

Abstract spatial 6-vector class

Abstract superclass for spatial vector functionality. This class has two abstract subclasses, which each have concrete subclasses:

SpatialVec6 (abstract handle class)

```
    |
    +--- SpatialM6 (abstract)
    |     |
    |     +---SpatialVelocity
    |     +---SpatialAcceleration
    |
    +---SpatialF6 (abstract)

         |
         +---SpatialForce
         +---SpatialMomentum
```

Methods

SpatialV6	constructor invoked by subclasses
double	convert to a $6 \times N$ double
char	convert to string
display	display in human readable form

Operators

+	add spatial vectors of the same type
-	subtract spatial vectors of the same type
-	unary minus of spatial vectors

Notes

- Subclass of the MATLAB handle class which means that pass by reference semantics apply.
- Spatial vectors can be placed into arrays and indexed.

References

- Robot Dynamics Algorithms, R. Featherstone, volume 22, Springer International Series in Engineering and Computer Science,
- Springer, 1987.
- A beginner's guide to 6-d vectors (part 1), R. Featherstone, IEEE Robotics Automation Magazine, 17(3):83-94, Sep. 2010.

See also SpatialM6, SpatialF6, SpatialVelocity, SpatialAcceleration, SpatialForce, SpatialMomentum, SpatialInertia.

SpatialVec6.SpatialVec6

Constructor

`SpatiaVecXXX(V)` is a spatial vector of type SpatiaVecXXX with a value from `V` (6×1). If `V` $(6 \times N)$ then an $(N \times 1)$ array of spatial vectors is returned.

This constructor is inherited by all the concrete subclasses.

See also

SpatialVelocity, SpatialAcceleration, SpatialForce, SpatialMomentum

---

SpatialVec6.char

Convert to string

`s = V.char()` is a string showing spatial vector parameters in a compact single line format. If V is an array of spatial vector objects return a string with one line per element.

See also

SpatialVec6.display

SpatialVec6.display

Display parameters

`V.display()` displays the spatial vector parameters in compact single line format. If V is an array of spatial vector objects it displays one per line.

Notes

- This method is invoked implicitly at the command line when the result of an expression is a serial vector subclass object and the command has
- no trailing semicolon.

See also

SpatialVec6.char

SpatialVec6.double

Convert to matrix

`double(V)` is a native matrix (6×1) with the value of the spatial vector. If `V` is an array $(1 \times N)$ the result is a matrix $(6 \times N)$.

SpatialVec6.minus

Subtraction operator

`V1 - V2` is a spatial vector of the same type as `V1` and `V2` whose value is the difference of `V1` and `V2`. If both are arrays of spatial vectors `V1` $(1 \times N)$ and `V2` $(1 \times N)$ the result is an array $(1 \times N)$.

See also

SpatialVec6.uminus, SpatialVec6.plus

SpatialVec6.plus

Addition operator

`V1 + V2` is a spatial vector of the same type as `V1` and `V2` whose value is the sum of `V1` and `V2`. If both are arrays of spatial vectors `V1` $(1 \times N)$ and `V2` $(1 \times N)$ the result is an array $(1 \times N)$.

See also

SpatialVec6.minus

SpatialVec6.uminus

Unary minus operator

- V is a spatial vector of the same type as V whose value is

the negative of V. If V is an array V ($1 \times N$) then the result is an array ($1 \times N$).

See also

SpatialVec6.minus, SpatialVec6.plus

SpatialVelocity

Spatial velocity class

Concrete subclass of SpatialM6 and represents the translational and rotational velocity of a rigid-body moving in 3D space.

```
SpatialVec6 (abstract handle class)
  |
  +--- SpatialM6 (abstract)
  |     |
  |     +---SpatialVelocity
  |     +---SpatialAcceleration
  |
  +---SpatialF6 (abstract)
       |
       +---SpatialForce
       +---SpatialMomentum
```

Methods

SpatialVelocity	^constructor invoked by subclasses
char	^convert to string
cross	^^cross product
display	^display in human readable form
double	^convert to a $6 \times N$ double
new	construct new concrete class of same type

Operators

+	^add spatial vectors of the same type
-	^subtract spatial vectors of the same type
-	^unary minus of spatial vectors
*	^^^premultiplication by SpatialInertia yields SpatialMomentum
*	^^^^premultiplication by Twist yields transformed SpatialVelocity

Notes:

- ^is inherited from SpatialVec6.

- ^^is inherited from SpatialM6.
- ^^^are implemented in SpatialInertia.
- ^^^^are implemented in Twist.

References

- Robot Dynamics Algorithms, R. Featherstone, volume 22, Springer International Series in Engineering and Computer Science,
- Springer, 1987.
- A beginner's guide to 6-d vectors (part 1), R. Featherstone, IEEE Robotics Automation Magazine, 17(3):83-94, Sep. 2010.

See also

SpatialVec6, SpatialM6, SpatialAcceleration, SpatialInertia, SpatialMomentum

---

SpatialVelocity.new

Construct a new object of the same type

`A2 = A.new(X)` creates a new object of the same type as A, with the value `X` (6×1).

Notes

- Serves as a dynamic constructor.
- This method is polymorphic across all SpatialVec6 derived classes, and allows easy creation of a new object of the same class as an existing
- one without needing to explicitly determine its type.

---

stlRead

Reads STL file

`[v, f, n, objname] = stlRead(fileName)` reads the STL format file (ASCII or binary) and returns:

V (Mx3)	each row is the 3D coordinate of a vertex
F (Nx3)	each row is a list of vertex indices that defines a triangular face
N (Nx3)	each row is a unit-vector defining the face normal
OBJNAME	is the name of the STL object (NOT the name of the STL file).

Authors

- From MATLAB File Exchange by Pau Mico, `https://au.mathworks.com/matlabcentral/fileexchange/51200-stltools`
- Copyright (c) 2015, Pau Mico
- Copyright (c) 2013, Adam H. Aitkenhead
- Copyright (c) 2011, Francis Esmonde-White

t2r

Rotational submatrix

`R = T2R(T)` is the orthonormal rotation matrix component of homogeneous transformation matrix `T`. Works for `T` in SE(2) or SE(3)

- If `T` is 4×4, then `R` is 3×3.
- If `T` is 3×3, then `R` is 2×2.

Notes

- For a homogeneous transform sequence ($K \times K \times N$) returns a rotation matrix sequence ($K - 1 \times K$-1 $\times N$).
- The validity of rotational part is not checked

See also

r2t, tr2rt, rt2tr

tb_optparse

Standard option parser for Toolbox functions

`OPTOUT = TB_OPTPARSE(OPT, ARGLIST)` is a generalized option parser for Toolbox functions. `OPT` is a structure that contains the names and default values for the options, and `ARGLIST` is a cell array containing option parameters, typically it comes from VARARGIN. It supports options that have an assigned value, boolean or enumeration types (string or int).

`[OPTOUT,ARGS] = TB_OPTPARSE(OPT, ARGLIST)` as above but returns all the unassigned options, those that don't match anything in `OPT`, as a cell array of all unassigned arguments in the order given in `ARGLIST`.

`[OPTOUT,ARGS,LS] = TB_OPTPARSE(OPT, ARGLIST)` as above but if any unmatched option looks like a MATLAB LineSpec (eg. 'r:') it is placed in `LS` rather than in `ARGS`.

`[OBJOUT,ARGS,LS] = TB_OPTPARSE(OPT, ARGLIST, OBJ)` as above but properties of `OBJ` with matching names in `OPT` are set.

The software pattern is:

```
function myFunction(a, b, c, varargin)
   opt.foo = false;
   opt.bar = true;
   opt.blah = [];
   opt.stuff = {};
   opt.choose = {'this', 'that', 'other'};
   opt.select = {'#no', '#yes'};
   opt.old = '@foo';
```

```
        opt = tb_optparse(opt, varargin);
```

Optional arguments to the function behave as follows:

'foo'	sets opt.foo := true
'nobar'	sets opt.foo := false
'blah', 3	sets opt.blah := 3
'blah',x,y	sets opt.blah := {x,y}
'that'	sets opt.choose := 'that'
'yes'	sets opt.select := 2 (the second element)
'stuff', 5	sets opt.stuff to {5}
'stuff', 'k',3	sets opt.stuff to {'k',3}
'old'	synonym, is the same as the option foo

and can be given in any combination.

If neither of 'this', 'that'or 'other'are specified then opt.choose := 'this'. Alternatively if:

```
    opt.choose = {[], 'this', 'that', 'other'};
```

then if neither of 'this', 'that'or 'other'are specified then opt.choose := [].

If neither of 'no'or 'yes'are specified then opt.select := 1.

The return structure is automatically populated with fields: verbose and debug. The following options are automatically parsed:

'verbose'	sets opt.verbose := true
'verbose=2'	sets opt.verbose := 2 (very verbose)
'verbose=3'	sets opt.verbose := 3 (extremeley verbose)
'verbose=4'	sets opt.verbose := 4 (ridiculously verbose)
'debug', N	sets opt.debug := N
'showopt'	displays opt and arglist
'setopt',S opt.foo is set to 4.	sets opt := S, if S.foo=4, and opt.foo is present, then

The allowable options are specified by the names of the fields in the structure `OPT`. By default if an option is given that is not a field of `OPT` an error is declared.

Notes

- That the enumerator names must be distinct from the field names.
- That only one value can be assigned to a field, if multiple values are required they must placed in a cell array.
- If the option is seen multiple times the last (rightmost) instance applies.
- To match an option that starts with a digit, prefix it with 'd_', so the field 'd_3d'matches the option '3d'.
- Any input argument or element of the opt struct can be a string instead of a char array.

tr2angvec

Convert rotation matrix to angle-vector form

`[THETA,V] = TR2ANGVEC(R, OPTIONS)` is rotation expressed in terms of an angle `THETA` (1×1) about the axis `V` (1×3) equivalent to the orthonormal rotation matrix `R` (3×3).

`[THETA,V] = TR2ANGVEC(T, OPTIONS)` as above but uses the rotational part of the homogeneous transform `T` (4×4).

If `R` ($3 \times 3 \times K$) or `T` ($4 \times 4 \times K$) represent a sequence then `THETA` ($K \times 1$)is a vector of angles for corresponding elements of the sequence and `V` ($K \times 3$) are the corresponding axes, one per row.

Options

'deg'	Return angle in degrees (default radians)

Notes

- For an identity rotation matrix both `THETA` and `V` are set to zero.
- The rotation angle is always in the interval $[0\ \pi]$, negative rotation is handled by inverting the direction of the rotation axis.
- If no output arguments are specified the result is displayed.

See also

angvec2r, angvec2tr, trlog

tr2delta

Convert SE(3) homogeneous transform to differential motion

`D = TR2DELTA(T0, T1)` is the differential motion (6×1) corresponding to infinitessimal motion (in the `T0` frame) from pose `T0` to `T1` which are homogeneous transformations (4×4) or SE3 objects.

The vector `D`=(dx, dy, dz, dRx, dRy, dRz) represents infinitessimal translation and rotation, and is an approximation to the instantaneous spatial velocity multiplied by time step.

`D = TR2DELTA(T)` as above but the motion is from the world frame to the SE3 pose `T`.

Notes

- `D` is only an approximation to the motion `T`, and assumes that `T0`≈`T1` or `T`≈eye(4,4).
- Can be considered as an approximation to the effect of spatial velocity over a a time interval, average spatial velocity multiplied by time.

Reference

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p67.

See also

delta2tr, skew, SE3.todelta

tr2eul

Convert SO(3) or SE(3) matrix to Euler angles

`EUL = TR2EUL(T, OPTIONS)` are the ZYZ Euler angles (1×3) corresponding to the rotational part of a homogeneous transform `T` (4×4). The 3 angles `EUL`=[PHI,THETA,PSI] correspond to sequential rotations about the Z, Y and Z axes respectively.

`EUL = TR2EUL(R, OPTIONS)` as above but the input is an orthonormal rotation matrix `R` (3×3).

If `R` ($3 \times 3 \times K$) or `T` ($4 \times 4 \times K$) represent a sequence then each row of `EUL` corresponds to a step of the sequence.

Options

'deg'	Compute angles in degrees (radians default)
'flip'	Choose first Euler angle to be in quadrant 2 or 3.

Notes

- There is a singularity for the case where THETA=0 in which case PHI is arbitrarily set to zero and PSI is the sum (PHI+PSI).

- Translation component is ignored.

See also

eul2tr, tr2rpy

tr2jac

Jacobian for differential motion

`J = TR2JAC(TAB)` is a Jacobian matrix (6×6) that maps spatial velocity or differential motion from frame $\{A\}$ to frame $\{B\}$ where the pose of $\{B\}$ relative to $\{A\}$ is represented by the homogeneous transform `TAB` (4×4).

`J = TR2JAC(TAB, 'samebody')` is a Jacobian matrix (6×6) that maps spatial velocity or differential motion from frame $\{A\}$ to frame $\{B\}$ where both are attached to the same moving body. The pose of $\{B\}$ relative to $\{A\}$ is represented by the homogeneous transform `TAB` (4×4).

See also

wtrans, tr2delta, delta2tr, SE3.velxform

tr2rpy

Convert SO(3) or SE(3) matrix to roll-pitch-yaw angles

`RPY = TR2RPY(T, options)` are the roll-pitch-yaw angles (1×3) corresponding to the rotation part of a homogeneous transform `T`. The 3 angles `RPY`=[ROLL,PITCH,YAW] correspond to sequential rotations about the Z, Y and X axes respectively. Roll and yaw angles are in $[-\pi, \pi)$ while pitch angle is in [-$\pi/2$,$\pi/2$).

`RPY = TR2RPY(R, options)` as above but the input is an orthonormal rotation matrix `R` (3×3).

If `R` ($3 \times 3 \times K$) or `T` ($4 \times 4 \times K$) represent a sequence then each row of `RPY` corresponds to a step of the sequence.

Options

'deg'	Compute angles in degrees (radians default)

'xyz'	Return solution for sequential rotations about X, Y, Z axes
'zyx'	Return solution for sequential rotations about Z, Y, X axes (default)
'yxz'	Return solution for sequential rotations about Y, X, Z axes
'arm'	Return solution for sequential rotations about X, Y, Z axes
'vehicle'	Return solution for sequential rotations about Z, Y, X axes
'camera'	Return solution for sequential rotations about Y, X, Z axes

Notes

- There is a singularity for the case where PITCH=$\pi/2$ in which case ROLL is arbitrarily set to zero and YAW is the sum (ROLL+YAW).
- Translation component is ignored.
- Toolbox rel 8-9 has XYZ angle sequence as default.
- 'arm', 'vehicle', 'camera'are synonyms for 'xyz', 'zyx'and 'yxz'respectively.
- these solutions are generated by symbolic/rpygen.mlx

See also

rpy2tr, tr2eul

---

tr2rt

Convert homogeneous transform to rotation and translation

`[R,t] = TR2RT(TR)` splits a homogeneous transformation matrix ($N \times N$) into an orthonormal rotation matrix `R` ($M \times M$) and a translation vector `t` ($M \times 1$), where N=M+1.

Works for `TR` in SE(2) or SE(3)

- If `TR` is 4×4, then `R` is 3×3 and T is 3×1.
- If `TR` is 3×3, then `R` is 2×2 and T is 2×1.

A homogeneous transform sequence `TR` ($N \times N \times K$) is split into rotation matrix sequence `R` ($M \times M \times K$) and a translation sequence `t` ($K \times M$).

Notes

- The validity of `R` is not checked.

See also

rt2tr, r2t, t2r

---

tranimate

Animate a 3D coordinate frame

`TRANIMATE(P1, P2, OPTIONS)` animates a 3D coordinate frame moving from pose X1 to pose X2. Poses X1 and X2 can be represented by:

- SE(3) homogeneous transformation matrices (4×4)
- SO(3) orthonormal rotation matrices (3×3)

`TRANIMATE(X, OPTIONS)` animates a coordinate frame moving from the identity pose to the pose `X` represented by any of the types listed above.

`TRANIMATE(XSEQ, OPTIONS)` animates a trajectory, where `XSEQ` is any of

- SE(3) homogeneous transformation matrix sequence ($4 \times 4 \times N$)
- SO(3) orthonormal rotation matrix sequence ($3 \times 3 \times N$)

Options

'fps', fps	Number of frames per second to display (default 10)
'nsteps', n	The number of steps along the path (default 50)
'axis',A	Axis bounds [xmin, xmax, ymin, ymax, zmin, zmax]
'movie',M	Save frames as a movie or sequence of frames
'cleanup'	Remove the frame at end of animation
'noxyz'	Don't label the axes
'rgb'	Color the axes in the order x=red, y=green, z=blue
'retain'	Retain frames, don't animate

Additional options are passed through to TRPLOT.

Notes

- Uses the Animate helper class to record the frames.

See also

trplot, Animate, SE3.animate

---

tranimate2

Animate a 2D coordinate frame

`TRANIMATE2(P1, P2, OPTIONS)` animates a 3D coordinate frame moving from pose X1 to pose X2. Poses X1 and X2 can be represented by:

- SE(2) homogeneous transformation matrices (3×3)
- SO(2) orthonormal rotation matrices (2×2)

`TRANIMATE2(X, OPTIONS)` animates a coordinate frame moving from the identity pose to the pose `X` represented by any of the types listed above.

`TRANIMATE2(XSEQ, OPTIONS)` animates a trajectory, where `XSEQ` is any of

- SE(2) homogeneous transformation matrix sequence ($3 \times 3 \times N$)
- SO(2) orthonormal rotation matrix sequence ($2 \times 2 \times N$)

Options

'fps', fps	Number of frames per second to display (default 10)
'nsteps', n	The number of steps along the path (default 50)
'axis',A	Axis bounds [xmin, xmax, ymin, ymax, zmin, zmax]
'movie',M	Save frames as a movie or sequence of frames
'cleanup'	Remove the frame at end of animation
'noxyz'	Don't label the axes
'rgb'	Color the axes in the order x=red, y=green, z=blue
'retain'	Retain frames, don't animate

Additional options are passed through to TRPLOT2.

Notes

- Uses the Animate helper class to record the frames.

See also

trplot, Animate, SE3.animate

transl

SE(3) translational homogeneous transform

Create a translational SE(3) matrix

`T = TRANSL(X, Y, Z)` is an SE(3) homogeneous transform (4×4) representing a pure translation of `X`, `Y` and `Z`.

`T = TRANSL(P)` is an SE(3) homogeneous transform (4×4) representing a translation of `P`=[`X`,`Y`,`Z`]. `P` ($M \times 3$) represents a sequence and `T` ($4 \times 4 \times M$) is a sequence of homogeneous transforms such that `T`(:,:,i) corresponds to the i'th row of `P`.

Extract the translational part of an SE(3) matrix

`P = TRANSL(T)` is the translational part of a homogeneous transform `T` as a 3-element column vector. `T` ($4 \times 4 \times M$) is a homogeneous transform sequence and the rows of `P` ($M \times 3$) are the translational component of the corresponding transform in the sequence.

`[X,Y,Z] = TRANSL(T)` is the translational part of a homogeneous transform `T` as three components. If `T` ($4 \times 4 \times M$) is a homogeneous transform sequence then `X,Y,Z` ($1 \times M$) are the translational components of the corresponding transform in the sequence.

Notes

- Somewhat unusually, this function performs a function and its inverse. An historical anomaly.

See also

SE3.t, SE3.transl

transl2

SE(2) translational homogeneous transform

Create a translational SE(2) matrix

`T = TRANSL2(X, Y)` is an SE(2) homogeneous transform (3×3) representing a pure translation.

`T = TRANSL2(P)` is a homogeneous transform representing a translation or point `P`=[`X`,`Y`]. `P` ($M \times 2$) represents a sequence and `T` ($3 \times 3 \times M$) is a sequence of homogeneous transforms such that `T`(:,:,i) corresponds to the i'th row of `P`.

Extract the translational part of an SE(2) matrix

`P = TRANSL2(T)` is the translational part of a homogeneous transform as a 2-element column vector. `T` ($3 \times 3 \times M$) is a homogeneous transform sequence and the rows of `P` ($M \times 2$) are the translational component of the corresponding transform in the sequence.

Notes

- Somewhat unusually, this function performs a function and its inverse. An historical anomaly.

See also

SE2.t, rot2, ishomog2, trplot2, transl

trchain

Compound SE(3) transforms from string

`T = TRCHAIN(S)` is a homogeneous transform (4×4) that results from compounding a number of elementary transformations defined by the string `S`. The string `S` comprises a number of tokens of the form X(ARG) where X is one of Tx, Ty, Tz, Rx, Ry, or Rz. ARG is an arbitrary MATLAB expression that can include constants or workspace variables. For example:

```
    trchain('Tx(1) Rx(90) Ry(45) Tz(2)')
```

is equivalent to computing

```
transl(1,0,0) * trotx(90, 'deg') * troty(45, 'deg') * transl(0,0,2)
```

`T = TRCHAIN(S, Q)` as above but the expression for ARG can also contain a variable 'qJ'which selects the Jth value from the passed vector `Q` ($1 \times N$). For example:

```
trchain('Rx(q1)Tx(a1)Ry(q2)Ty(a3)Rz(q3)', [1 2 3])
```

[`T`,TOK] = TRCHAIN(`S` ...) as above but return an array of tokens which can be passed in, instead of the string.

`T` = TRCHAIN(TOK ...) as above but chain is defined by array of tokens instead of a string.

Options

- 'deg'	all angular variables are in degrees (default radians)
- 'qvar',V	treat the string V as the joint variable name rather than 'q'

Notes

- Variables used in the string must exist in the caller workspace.
- The string can contain arbitrary characters between the elements, for example space, +, *, . or even |.
- Works for symbolic variables in the workspace and/or passed in via the vector `Q`.
- For symbolic operations that involve use of the value π, make sure you define it first in the workspace: π = sym('π');
- The tokens are simply a parsed version of the input string and provide some efficiency for repeated calls on the same chain.

See also

trchain2, trotx, troty, trotz, transl, SerialLink.trchain, ets

trchain2

Compound SE(2) transforms from string

`T = TRCHAIN(S)` is a homogeneous transform (3×3) that results from compounding a number of elementary transformations defined by the string `S`. The string `S` comprises a number of tokens of the form X(ARG) where X is one of Tx, Ty, or R. ARG is an arbitrary MATLAB expression that can include constants or workspace variables. For example:

```
trchain('Tx(1) R(90) Ty(2)')
```

is equivalent to computing

```
transl2(1,0) * trot2(90, 'deg') * transl2(0,2)
```

`T = TRCHAIN(S, Q)` as above but the expression for ARG can also contain a variable 'qJ'which selects the Jth value from the passed vector `Q` ($1 \times N$). For example:

```
trchain('Tx(1) R(q1-90) Ty(2) R(q2)', [1 2])
```

[`T`,TOK] = TRCHAIN(`S` ...) as above but return an array of tokens which can be passed in, instead of the string.

`T` = TRCHAIN(TOK ...) as above but chain is defined by array of tokens instead of a string.

Options

- 'deg'	all angular variables are in degrees (default radians)
- 'qvar',V	treat the string V as the joint variable name rather than 'q'

Notes

- Variables used in the string must exist in the caller workspace.
- The string can contain arbitrary characters between the elements, for example space, +, *, . or even |.
- Works for symbolic variables in the workspace and/or passed in via the vector `Q`.
- For symbolic operations that involve use of the value π, make sure you define it first in the workspace: π = sym('π');
- The tokens are simply a parsed version of the input string and provide some efficiency for repeated calls on the same chain.

See also

trchain2, trotx, troty, trotz, transl, SerialLink.trchain, ets

trexp

Matrix exponential for so(3) and se(3)

For so(3)

`R = TREXP(OMEGA)` is the matrix exponential (3×3) of the so(3) element `OMEGA` that yields a rotation matrix (3×3).

`R = TREXP(OMEGA, THETA)` as above, but so(3) motion of `THETA*OMEGA`.

`R = TREXP(S, THETA)` as above, but rotation of `THETA` about the unit vector `S`.

`R = TREXP(W)` as above, but the so(3) value is expressed as a vector `W` (1×3) where `W` = `S` * `THETA`. Rotation by ||`W`|| about the vector `W`.

For se(3)

`T = TREXP(SIGMA)` is the matrix exponential (4×4) of the se(3) element `SIGMA` that yields a homogeneous transformation matrix (4×4).

`T = TREXP(SIGMA, THETA)` as above, but se(3) motion of `SIGMA*THETA`, the rotation part of `SIGMA` (4×4) must be unit norm.

`T = TREXP(TW)` as above, but the se(3) value is expressed as a twist vector `TW` (1×6).

`T = TREXP(TW, THETA)` as above, but se(3) motion of `TW*THETA`, the rotation part of `TW` (1×6) must be unit norm.

Notes

- Efficient closed-form solution of the matrix exponential for arguments that are so(3) or se(3).
- If `THETA` is given then the first argument must be a unit vector or a skew-symmetric matrix from a unit vector.
- Angle vector argument order is different to ANGVEC2R.

References

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p42-43.
- Mechanics, planning and control, Park & Lynch, Cambridge, 2017.

See also

angvec2r, trlog, trexp2, skew, skewa, Twist

---

trexp2

Matrix exponential for so(2) and se(2)

SO(2)

`R = TREXP2(OMEGA)` is the matrix exponential (2×2) of the so(2) element `OMEGA` that yields a rotation matrix (2×2).

`R = TREXP2(THETA)` as above, but rotation by `THETA` (1×1).

SE(2)

`T = TREXP2(SIGMA)` is the matrix exponential (3×3) of the se(2) element `SIGMA` that yields a homogeneous transformation matrix (3×3).

`T = TREXP2(SIGMA, THETA)` as above, but se(2) rotation of `SIGMA*THETA`, the rotation part of `SIGMA` (3×3) must be unit norm.

`T = TREXP2(TW)` as above, but the se(2) value is expressed as a vector `TW` (1×3).

`T = TREXP(TW, THETA)` as above, but se(2) rotation of `TW*THETA`, the rotation part of `TW` must be unit norm.

Notes

- Efficient closed-form solution of the matrix exponential for arguments that are so(2) or se(2).
- If `THETA` is given then the first argument must be a unit vector or a skew-symmetric matrix from a unit vector.

References

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p25-26.
- Mechanics, planning and control, Park & Lynch, Cambridge, 2017.

See also

trexp, skew, skewa, Twist

---

trinterp

Interpolate SE(3) homogeneous transformations

`TRINTERP(T0, T1, S)` is a homogeneous transform (4×4) interpolated between `T0` when `S`=0 and `T1` when `S`=1. `T0` and `T1` are both homogeneous transforms (4×4). If `S` ($N \times 1$) then T ($4 \times 4 \times N$) is a sequence of homogeneous transforms corresponding to the interpolation values in `S`.

`TRINTERP(T1, S)` as above but interpolated between the identity matrix when `S`=0 to `T1` when `S`=1.

`TRINTERP(T0, T1, M)` as above but `M` is a positive integer and return a sequence ($4 \times 4 \times$ `M`) of homogeneous transforms linearly interpolating between `T0` and `T1` in `M` steps.

`TRINTERP(T1, M)` as above but return a sequence ($4 \times 4 \times$ `M`) of homogeneous interpolating between identity matrix and `T1` in `M` steps.

Notes

- `T0` or `T1` can also be an SO(3) rotation matrix (3×3) in which case the result is ($3 \times 3 \times N$).
- Rotation is interpolated using quaternion spherical linear interpolation (slerp).
- To obtain smooth continuous motion `S` should also be smooth and continuous, such as computed by tpoly or lspb.

See also

trinterp2, ctraj, SE3.interp, UnitQuaternion, tpoly, lspb

---

trinterp2

Interpolate SE(2) homogeneous transformations

`TRINTERP2(T0, T1, S)` is a homogeneous transform (3×3) interpolated between `T0` when `S`=0 and `T1` when `S`=1. `T0` and `T1` are both homogeneous transforms (4×4). If `S` ($N \times 1$) then T ($3 \times 3 \times N$) is a sequence of homogeneous transforms corresponding to the interpolation values in `S`.

`TRINTERP2(T1, S)` as above but interpolated between the identity matrix when `S`=0 to `T1` when `S`=1.

`TRINTERP2(T0, T1, M)` as above but `M` is a positive integer and return a sequence ($4 \times 4 \times$ `M`) of homogeneous transforms linearly interpolating between `T0` and `T1` in `M` steps.

`TRINTERP2(T1, M)` as above but return a sequence ($4 \times 4 \times$ `M`) of homogeneous interpolating between identity matrix and `T1` in `M` steps.

Notes

- `T0` or `T1` can also be an SO(2) rotation matrix (2×2).
- Rotation angle is linearly interpolated.
- To obtain smooth continuous motion `S` should also be smooth and continuous, such as computed by tpoly or lspb.

See also

trinterp, SE3.interp, UnitQuaternion, tpoly, lspb

---

trlog

Logarithm of SO(3) or SE(3) matrix

`S = trlog(R)` is the matrix logarithm (3×3) of `R` (3×3) which is a skew symmetric matrix corresponding to the vector theta*w where theta is the rotation angle and w (3×1) is a unit-vector indicating the rotation axis.

`[theta,w] = trlog(R)` as above but returns directly `theta` the rotation angle and `w` (3×1) the unit-vector indicating the rotation axis.

`S = trlog(T)` is the matrix logarithm (4×4) of `T` (4×4) which has a skew-symmetric upper-left 3×3 submatrix corresponding to the vector `theta*w` where `theta` is the rotation angle and `w` (3×1) is a unit-vector indicating the rotation axis, and a translation component.

`[theta,twist] = trlog(T)` as above but returns directly `theta` the rotation angle and a `twist` vector (6×1) comprising [v w].

Notes

- Efficient closed-form solution of the matrix logarithm for arguments that are SO(3) or SE(3).
- Special cases of rotation by odd multiples of π are handled.
- Angle is always in the interval [0,π].
- There is no Toolbox function for SO(2) or SE(2), use LOGM instead.

References

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p43.
- Mechanics, planning and control, Park & Lynch, Cambridge, 2016.

See also

trexp, trexp2, Twist, logm

---

trnorm

Normalize an SO(3) or SE(3) matrix

`TRNORM(R)` is guaranteed to be a proper orthogonal matrix rotation matrix (3×3) which is "close" to the input matrix `R` (3×3). If `R` = [N,O,A] the O and A vectors are made unit length and the normal vector is formed from N = O x A, and then we ensure that O and A are orthogonal by O = A x N.

`TRNORM(T)` as above but the rotational submatrix of the homogeneous transformation `T` (4×4) is normalised while the translational part is unchanged.

If `R` ($3 \times 3 \times K$) or `T` ($4 \times 4 \times K$) representing a sequence then the normalisation is performed on each of the K planes.

Notes

- Only the direction of A (the z-axis) is unchanged.
- Used to prevent finite word length arithmetic causing transforms to become 'unnormalized'.
- There is no Toolbox function for SO(2) or SE(2).

See also

oa2tr, SO3.trnorm, SE3.trnorm

trot2

SE(2) rotation matrix

`T = TROT2(THETA)` is a homogeneous transformation (3×3) representing a rotation of `THETA` radians.

`T = TROT2(THETA, 'deg')` as above but `THETA` is in degrees.

Notes

- Translational component is zero.

See also

rot2, transl2, ishomog2, trplot2, trotx, troty, trotz, SE2

trotx

SE(3) rotation about X axis

`T = TROTX(THETA)` is a homogeneous transformation (4×4) representing a rotation of `THETA` radians about the x-axis.

`T = TROTX(THETA, 'deg')` as above but `THETA` is in degrees.

Notes

- Translational component is zero.

See also
rotx, troty, trotz, trot2, SE3.Rx

troty

SE(3) rotation about Y axis
`T = troty(THETA)` is a homogeneous transformation (4×4) representing a rotation of `THETA` radians about the y-axis.

`T = troty(THETA, 'deg')` as above but `THETA` is in degrees.

Notes
- Translational component is zero.

See also
roty, trotx, trotz, trot2, SE3.Ry

trotz

SE(3) rotation about Z axis
`T = trotz(THETA)` is a homogeneous transformation (4×4) representing a rotation of `THETA` radians about the z-axis.

`T = trotz(THETA, 'deg')` as above but `THETA` is in degrees.

Notes
- Translational component is zero.

See also
rotz, trotx, troty, trot2, SE3.Rz

trplot

Plot a 3D coordinate frame
`TRPLOT(T, OPTIONS)` draws a 3D coordinate frame represented by the SE(3) homogeneous transform `T` (4×4).

`H = TRPLOT(T, OPTIONS)` as above but returns a handle.

`TRPLOT(R, OPTIONS)` as above but the coordinate frame is rotated about the origin according to the orthonormal rotation matrix `R` (3×3).

`H = TRPLOT(R, OPTIONS)` as above but returns a handle.

`H = TRPLOT()` creates a default frame EYE(3,3) at the origin and returns a handle.

Animation

Firstly, create a plot and keep the the handle as per above.

`TRPLOT(T, 'handle', H)` moves the coordinate frame described by the handle `H` to the pose `T` (4×4).

Options

'handle',h	Update the specified handle
'axhandle',A	Draw in the MATLAB axes specified by the axi
'color',C	The color to draw the axes, MATLAB ColorSpe
'axes'	Show the MATLAB axes, box and ticks (defaul
'axis',A	Set dimensions of the MATLAB axes to A=[xm
'frame',F	The coordinate frame is named {F} and the sub
'framelabel',F	The coordinate frame is named {F}, axes have n
'framelabeloffset',O	Offset O=[DX DY] frame labels in units of text
'text_opts', opt	A cell array of MATLAB text properties
'length',s	Length of the coordinate frame arms (default 1)
'thick',t	Thickness of lines (default 0.5)
'text'	Enable display of X,Y,Z labels on the frame (de
'labels',L	Label the X,Y,Z axes with the 1st, 2nd, 3rd char
'rgb'	Display X,Y,Z axes in colors red, green, blue re
'rviz'	Display chunky rviz style axes%
'arrow'	Use arrows rather than line segments for the axe
'width', w	Width of arrow tips (default 1)
'perspective'	Display the axes with perspective projection (de
'3d'	Plot in 3D using anaglyph graphics
'anaglyph',A left and right (default colors 'rc'): chosen from	Specify anaglyph colors for '3d'as 2 characters f r)ed, g)reen, b)lue, c)yan, m)agenta.
'dispar',D	Disparity for 3d display (default 0.1)
'view',V for view toward origin of coordinate frame	Set plot view parameters V=[az el] angles, or 'a
'lefty'	Draw left-handed frame (dangerous)

Examples

`trplot(T, 'frame', 'A')` trplot(T, 'frame', 'A', 'color', 'b') trplot(T1, 'frame', 'A', 'text_opts', {'FontSize', 10, 'FontWeight', 'bold'}) trplot(T1, 'labels', 'NOA');

`h = trplot(T, 'frame', 'A', 'color', 'b');` trplot(h, T2);

3D anaglyph plot

`trplot(T, '3d');`

Notes

- Multiple frames can be added using the HOLD command
- When animating a coordinate frame it is best to set the axis bounds initially.
- The 'rviz'option is equivalent to 'rgb', 'notext', 'noarrow', 'thick', 5.
- The 'arrow'option requires https://www.mathworks.com/matlabcentral/fileexchange/14056-arrow3

trplot2

Plot a 2D coordinate frame

`TRPLOT2(T, OPTIONS)` draws a 2D coordinate frame represented by the SE(2) homogeneous transform `T` (3×3).

`H = TRPLOT2(T, OPTIONS)` as above but returns a handle.

`TRPLOT(R, OPTIONS)` as above but the coordinate frame is rotated about the origin according to the orthonormal rotation matrix `R` (2×2).

`H = TRPLOT(R, OPTIONS)` as above but returns a handle.

`H = TRPLOT2()` creates a default frame EYE(2,2) at the origin and returns a handle.

Animation

Firstly, create a plot and keep the the handle as per above.

`TRPLOT2(H, T)` moves the coordinate frame described by the handle `H` to the SE(2) pose `T` (3×3).

Options

'handle',h	Update the specified handle
'axhandle',A	Draw in the MATLAB axes specified by the axis handle A
'color', c	The color to draw the axes, MATLAB ColorSpec
'axes'	Show the MATLAB axes, box and ticks (default true)
'axis',A	Set dimensions of the MATLAB axes to A=[xmin xmax ymin ymax]
'frame',F	The frame is named {F} and the subscript on the axis labels is F.
'framelabel',F	The coordinate frame is named {F}, axes have no subscripts.
'framelabeloffset',O	Offset O=[DX DY] frame labels in units of text box height
'text_opts', opt	A cell array of Matlab text properties
'length',s	Length of the coordinate frame arms (default 1)
'thick',t	Thickness of lines (default 0.5)
'text'	Enable display of X,Y,Z labels on the frame (default true)
'labels',L	Label the X,Y,Z axes with the 1st and 2nd character of the string L
'arrow'	Use arrows rather than line segments for the axes
'width', w	Width of arrow tips
'lefty'	Draw left-handed frame (dangerous)

Examples

`trplot2(T, 'frame', 'A')` trplot2(T, 'frame', 'A', 'color', 'b') trplot2(T1, 'frame', 'A', 'text_opts', {'FontSize', 10, 'FontWeight', 'bold'})

Notes

- Multiple frames can be added using the HOLD command
- When animating a coordinate frame it is best to set the axis bounds initially.
- The 'arrow'option requires `https://www.mathworks.com/matlabcentral/fileexchange/14056-arrow3`

See also

trplot

trprint

Compact display of SE(3) homogeneous transformation

`TRPRINT(T, OPTIONS)` displays the homogoneous transform (4×4) in a compact single-line format. If `T` is a homogeneous transform sequence then each element is printed on a separate line.

`TRPRINT(R, OPTIONS)` as above but displays the SO(3) rotation matrix (3×3).

`S = TRPRINT(T, OPTIONS)` as above but returns the string.

TRPRINT `T OPTIONS` is the command line form of above.

---

trprint2

Compact display of SE(2) homogeneous transformation

`TRPRINT2(T, OPTIONS)` displays the homogoneous transform (3×3) in a compact single-line format. If `T` is a homogeneous transform sequence then each element is printed on a separate line.

`TRPRINT2(R, OPTIONS)` as above but displays the SO(2) rotation matrix (3×3).

`S = TRPRINT2(T, OPTIONS)` as above but returns the string.

TRPRINT2 `T` is the command line form of above, and displays in RPY format.

Options

'radian'	display angle in radians (default is degrees)
'fmt', f	use format string f for all numbers, (default %g)
'label',l	display the text before the transform

Examples

```
>> trprint2(T2)
t = (0,0), theta = -122.704 deg
```

See also

trprint

---

trscale

Homogeneous transformation for pure scale

`T = TRSCALE(S)` is a homogeneous transform (4×4) corresponding to a pure scale change. If `S` is a scalar the same scale factor is used for x,y,z, else it can be a 3-vector specifying scale in the x-, y- and z-directions.

Note

- This matrix does not belong to SE(3) and if compounded with any SE(3) matrix the result will not be in SE(3).

Twist

SE(2) and SE(3) Twist class

A Twist class holds the parameters of a twist, a representation of a rigid body displacement in SE(2) or SE(3).

Methods

S	twist vector (1×3 or 1×6)
se	twist as (augmented) skew-symmetric matrix (3×3 or 4×4)
T	convert to homogeneous transformation (3×3 or 4×4)
R	convert rotational part to matrix (2×2 or 3×3)
exp	synonym for T
ad	logarithm of adjoint
pitch	pitch of the screw, SE(3) only
pole	a point on the line of the screw
prod	product of a vector of Twists
theta	rotation about the screw
line	Plucker line object representing line of the screw
display	print the Twist parameters in human readable form
char	convert to string

Conversion methods

SE	convert to SE2 or SE3 object
double	convert to real vector

Overloaded operators

$*$	compose two Twists
$*$	multiply Twist by a scalar

Properties (read only)

v	moment part of twist (2×1 or 3×1)
w	direction part of twist (1×1 or 3×1)

References

- "Mechanics, planning and control" Park & Lynch, Cambridge, 2016.

See also

trexp, trexp2, trlog

Twist.Twist

Create Twist object

`TW = Twist(T)` is a Twist object representing the SE(2) or SE(3) homogeneous transformation matrix `T` (3×3 or 4×4).

`TW = Twist(V)` is a twist object where the vector is specified directly.

3D CASE::

`TW = Twist('R', A, Q)` is a Twist object representing rotation about the axis of direction `A` (3×1) and passing through the point `Q` (3×1).

`TW = Twist('R', A, Q, P)` as above but with a pitch of `P` (distance/angle).

`TW = Twist('T', A)` is a Twist object representing translation in the direction of `A` (3×1).

2D CASE::

`TW = Twist('R', Q)` is a Twist object representing rotation about the point `Q` (2×1).

`TW = Twist('T', A)` is a Twist object representing translation in the direction of `A` (2×1).

Notes

The argument '`P`'for prismatic is synonymous with '`T`'.

Twist.ad

Logarithm of adjoint

`TW.ad` is the logarithm of the adjoint matrix of the corresponding homogeneous transformation.

See also

SE3.Ad

Twist.Ad

Adjoint

`TW.Ad` is the adjoint matrix of the corresponding homogeneous transformation.

See also

SE3.Ad

Twist.char

Convert to string

`s = TW.char()` is a string showing Twist parameters in a compact single line format. If TW is a vector of Twist objects return a string with one line per Twist.

See also

Twist.display

Twist.display

Display parameters

`L.display()` displays the twist parameters in compact single line format. If L is a vector of Twist objects displays one line per element.

Notes

- This method is invoked implicitly at the command line when the result of an expression is a Twist object and the command has no trailing
- semicolon.

See also

Twist.char

Twist.double

Return the twist vector

`double(TW)` is the twist vector in se(2) or se(3) as a vector (3×1 or 6×1). If `TW` is a vector ($1 \times N$) of Twists the result is a matrix ($6 \times N$) with one column per twist.

Notes

- Sometimes referred to as the twist coordinate vector.

Twist.exp

Convert twist to homogeneous transformation

`TW.exp` is the homogeneous transformation equivalent to the twist (SE2 or SE3).

`TW.exp(THETA)` as above but with a rotation of `THETA` about the twist.

Notes

- For the second form the twist must, if rotational, have a unit rotational component.

See also

Twist.T, trexp, trexp2

Twist.line

Line of twist axis in Plucker form

`TW.line` is a Plucker object representing the `line` of the twist axis.

Notes

- For 3D case only.

See also

Plucker

Twist.mtimes

Multiply twist by twist or scalar

`TW1 * TW2` is a new **Twist** representing the composition of twists `TW1` and `TW2`.

`TW * T` is an SE2 or SE3 that is the composition of the twist `TW` and the homogeneous transformation object `T`.

`TW * S` with its twist coordinates scaled by scalar `S`.

`TW * T` compounds a twist with an SE2/3 transformation

Twist.pitch

Pitch of the twist

`TW.pitch` is the `pitch` of the **Twist** as a scalar in units of distance per radian.

Notes

- For 3D case only.

Twist.pole

Point on the twist axis

`TW.pole` is a point on the twist axis (2×1 or 3×1).

Notes

- For pure translation this point is at infinity.

Twist.prod

Compound array of twists

`TW.prod` is a twist representing the product (composition) of the successive elements of `TW` ($1 \times N$), an array of Twists.

See also

RTBPose.prod, Twist.mtimes

Twist.S

Return the twist vector

`TW.S` is the twist vector in se(2) or se(3) as a vector (3×1 or 6×1).

Notes

- Sometimes referred to as the twist coordinate vector.

Twist.SE

Convert twist to SE2 or SE3 object

`TW.SE` is an SE2 or SE3 object representing the homogeneous transformation equivalent to the twist.

See also

Twist.T, SE2, SE3

Twist.se

Return the twist matrix

`TW.se` is the twist matrix in `se`(2) or `se`(3) which is an augmented skew-symmetric matrix (3×3 or 4×4).

Twist.T

Convert twist to homogeneous transformation

`TW.T` is the homogeneous transformation equivalent to the twist (3×3 or 4×4).

`TW.T(THETA)` as above but with a rotation of `THETA` about the twist.

Notes

- For the second form the twist must, if rotational, have a unit rotational component.

See also

Twist.exp, trexp, trexp2, trinterp, trinterp2

Twist.theta

Twist rotation

`TW.theta` is the rotation (1×1) about the twist axis in radians.

Twist.unit

Return a unit twist

`TW.unit()` is a **Twist** object representing a unit aligned with the **Twist** TW.

---

unit

Unitize a vector

`VN = UNIT(V)` is a unit-vector parallel to `V`.

Note

- Reports error for the case where V is non-symbolic and norm(V) is zero

UnitQuaternion

Unit quaternion class

A UnitQuaternion 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>.

A UnitQuaternion is one for which $s^2+vx^2+vy^2+vz^2 = 1$. It can be considered as a rotation by an angle theta about a unit-vector V in space where

```
q = cos (theta/2) < v sin(theta/2)>
```

Constructors

UnitQuaternion	general constructor
UnitQuaternion.angvec	constructor, from (angle and vector)
UnitQuaternion.eul	constructor, from Euler angles
UnitQuaternion.omega	constructor for angle*vector
UnitQuaternion.rpy	constructor, from roll-pitch-yaw angles
UnitQuaternion.Rx	constructor, from x-axis rotation
UnitQuaternion.Ry	constructor, from y-axis rotation
UnitQuaternion.Rz	constructor, from z-axis rotation
UnitQuaternion.vec	constructor, from 3-vector

Display and print methods

animate	animates a coordinate frame
display	print in human readable form
plot	plot a coordinate frame representing orientation of quaternion

Group operations

*	^quaternion (Hamilton) product
.*	quaternion (Hamilton) product and renormalize
/	^multiply by inverse
./	multiply by inverse and renormalize
^	^exponentiate (integer only)
exp	^exponential
inv	^inverse
log	^logarithm
prod	product of elements

Methods

angle	angle between two quaternions
conj	^conjugate
dot	derivative of quaternion with angular velocity
inner	^inner product

interp	interpolation (slerp) between two quaternions
norm	^norm, or length
unit	unitized quaternion
UnitQuaternion.qvmul	multiply unit-quaternions in 3-vector form

Conversion methods

char	convert to string
double	^convert to 4-vector
matrix	convert to 4×4 matrix
R	convert to 3×3 rotation matrix
SE3	convert to SE3 object
SO3	convert to SO3 object
T	convert to 4×4 homogeneous transform matrix
toangvec	convert to angle vector form
toeul	convert to Euler angles
torpy	convert to roll-pitch-yaw angles
tovec	convert to 3-vector

Operators

+	elementwise sum of quaternion elements (result is a Quaternion)
-	elementwise difference of quaternion elements (result is a Quaternion)
==	test for equality
$\sim=$	^test for inequality

^means inherited from Quaternion class.

Properties (read only)

s	real part
v	vector part

Notes

- A subclass of Quaternion
- Many methods and operators are inherited from the Quaternion superclass.
- UnitQuaternion objects can be used in vectors and arrays.
- The + and - operators return a Quaternion object not a UnitQuaternion since these are not group operators.
- For display purposes a Quaternion differs from a UnitQuaternion by using $<<$ $>>$ notation rather than $<$ $>$.
- To a large extent polymorphic with the SO3 class.

References

- Animating rotation with quaternion curves, K. Shoemake,
- in Proceedings of ACM SIGGRAPH, (San Fran cisco), pp. 245-254, 1985.
- On homogeneous transforms, quaternions, and computational efficiency, J. Funda, R. Taylor, and R. Paul,
- IEEE Transactions on Robotics and Automation, vol. 6, pp. 382-388, June 1990.
- Quaternions for Computer Graphics, J. Vince, Springer 2011.

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p44-45.

See also

Quaternion, SO3

UnitQuaternion.UnitQuaternion

Construct a unit quaternion object

Construct a **UnitQuaternion** from various other orientation representations.

`Q = UnitQuaternion()` is the identitity **UnitQuaternion** 1<0,0,0> representing a null rotation.

`Q = UnitQuaternion(Q1)` is a copy of the **UnitQuaternion** `Q1`, if `Q1` is a Quaternion it is normalised.

`Q = UnitQuaternion(S, V)` is a **UnitQuaternion** formed by specifying directly its scalar and vector parts which are normalised.

`Q = UnitQuaternion([S, V1, V2, V3])` is a **UnitQuaternion** formed by specifying directly its 4 elements which are normalised.

`Q = Quaternion(R)` is a **UnitQuaternion** corresponding to the SO(3) orthonormal rotation matrix `R` (3×3). If `R` $(3 \times 3 \times N)$ is a sequence then `Q` $(N \times 1)$ is a vector of Quaternions corresponding to the elements of `R`.

`Q = Quaternion(T)` is a **UnitQuaternion** equivalent to the rotational part of the SE(3) homogeneous transform `T` (4×4). If `T` $(4 \times 4 \times N)$ is a sequence then `Q` $(N \times 1)$ is a vector of Quaternions corresponding to the elements of `T`.

Notes

- Only the `R` and `T` forms are vectorised.
- To convert an SO3 or SE3 object to a UnitQuaternion use their UnitQuaternion conversion methods.

See also **UnitQuaternion**.eul, **UnitQuaternion**.rpy, **UnitQuaternion**.angvec, UnitQuaternion.omega, UnitQuaternion.Rx, UnitQuaternion.Ry, UnitQuaternion.Rz, SE3.UnitQuaternion, SO3.UnitQuaternion.

UnitQuaternion.angdist

Distance metric

`Q1.angdist(Q2)` is a distance metric between two unit quaternions.

`Q1.angdist(Q2, M)` as above but uses the method specified by `M`

`M` = 1 (default): 1 - | < q1, q2> | in [0,$\pi/2$]

M = 2: acos | <q1, q2> | in [0,1]

Note

- angdist(q, -q) is equal to zero due to double mapping

See also: SO3.angdist

UnitQuaternion.angle

Angle between two UnitQuaternions

`A = Q1.angle(Q2)` is the angle (in radians) between two UnitQuaternions Q1 and `Q2`.

Notes

- If either, or both, of Q1 or `Q2` are vectors, then the result is a vector.
  - if Q1 is a vector $(1 \times N)$ then `A` is a vector $(1 \times N)$ such that `A`(i) = P1(i).angle(`Q2`).
  - if `Q2` is a vector $(1 \times N)$ then `A` is a vector $(1 \times N)$ such that `A`(i) = P1.angle(P2(i)).
  - if both Q1 and `Q2` are vectors $(1 \times N)$ then `A` is a vector $(1 \times N)$ such that `A`(i) = P1(i).angle(`Q2`(i)).

References

- Metrics for 3D rotations: comparison and analysis, Du Q. Huynh, J.Math Imaging Vis. DOFI 10.1007/s10851-009-0161-2.

See also

Quaternion.angvec

UnitQuaternion.angvec

Construct UnitQuaternion from angle and rotation vector

`Q = UnitQuaternion.angvec(TH, V)` is a **UnitQuaternion** representing rotation of `TH` about the vector `V` (3×1).

See also

UnitQuaternion.omega

UnitQuaternion.animate

Animate UnitQuaternion object

`Q.animate(options)` animates a **UnitQuaternion** array Q $(1 \times N)$ as a 3D coordinate frame.

`Q.animate(QF, options)` animates a 3D coordinate frame moving from orientation Q to orientation `QF`.

Options

Options are passed to tranimate and include:

'fps', fps	Number of frames per second to display (default 10)
'nsteps', n	The number of steps along the path (default 50)
'axis',A	Axis bounds [xmin, xmax, ymin, ymax, zmin, zmax]
'movie',M	Save frames as files in the folder M
'cleanup'	Remove the frame at end of animation
'noxyz'	Don't label the axes
'rgb'	Color the axes in the order x=red, y=green, z=blue

'retain'	Retain frames, don't animate

Additional `options` are passed through to TRPLOT.

See also

tranimate, trplot

UnitQuaternion.char

Convert to string

`S = Q.char()` is a compact string representation of the **UnitQuaternion**'s value as a 4-tuple. If Q is a vector then `S` has one line per element.

Notes

- The vector part is delimited by single angle brackets, to differentiate from a Quaternion which is delimited by double angle brackets.

See also

Quaternion.char

UnitQuaternion.dot

UnitQuaternion derivative in world frame

`QD = Q.dot(omega)` is the rate of change of the **UnitQuaternion** Q expressed as a Quaternion in the world frame. Q represents the orientation of a body frame with angular velocity OMEGA (1×3).

Notes

- This is not a group operator, but it is useful to have the result as a Quaternion.

Reference

- Robotics, Vision & Control, 2nd edition, Peter Corke, pp.64.

See also

UnitQuaternion.dotb

UnitQuaternion.dotb

UnitQuaternion derivative in body frame

`QD = Q.dotb(omega)` is the rate of change of the **UnitQuaternion** Q expressed as a Quaternion in the body frame. Q represents the orientation of a body frame with angular velocity OMEGA (1×3).

Notes

- This is not a group operator, but it is useful to have the result as a quaternion.

Reference

- Robotics, Vision & Control, 2nd edition, Peter Corke, pp.64.

See also

UnitQuaternion.dot

UnitQuaternion.eq

Test for equality

Q1 == Q2 is true if the two UnitQuaternions represent the same rotation.

Notes

- The double mapping of the UnitQuaternion is taken into account, that is, UnitQuaternions are equal if Q1.s == -Q1.s && Q1.v == -Q2.v.
- If Q1 is a vector of UnitQuaternions, each element is compared to Q2 and the result is a logical array of the same length as Q1.
- If Q2 is a vector of UnitQuaternion, each element is compared to Q1 and the result is a logical array of the same length as Q2.
- If Q1 and Q2 are equal length vectors of UnitQuaternion, then the result is a logical array of the same length.

UnitQuaternion.eul

Construct UnitQuaternion from Euler angles

`Q = UnitQuaternion.eul(PHI, THETA, PSI, OPTIONS)` is a **UnitQuaternion** representing rotation equivalent to the specified Euler angles angles. These correspond to rotations about the Z, Y, Z axes respectively.

`Q = UnitQuaternion.eul(EUL, OPTIONS)` as above but the Euler angles are taken from the vector (1×3) `EUL` = [`PHI THETA PSI`]. If `EUL` is a matrix $(N \times 3)$ then `Q` is a vector $(1 \times N)$ of UnitQuaternion objects where the index corresponds to rows of `EUL` which are assumed to be [`PHI`,`THETA`,`PSI`].

Options

'deg'	Compute angles in degrees (default radians)

Notes

- Is vectorised, see eul2r for details.

See also

UnitQuaternion.rpy, eul2r

UnitQuaternion.increment

Update UnitQuaternion by angular displacement

`QU = Q.increment(OMEGA)` updates Q by an infinitessimal rotation which is given as a spatial displacement `OMEGA` (3×1) whose direction is the rotation axis and magnitude is the amount of rotation.

Notes

- `OMEGA` is an approximation to the instantaneous spatial velocity multiplied by time step.

See also

tr2delta

UnitQuaternion.interp

Interpolate UnitQuaternion

`QI = Q.scale(S, OPTIONS)` is a **UnitQuaternion** that interpolates between a null rotation (identity UnitQuaternion) for `S`=0 to Q for `S`=1.

`QI = Q1.interp(Q2, S, OPTIONS)` as above but interpolates a rotation between Q1 for `S`=0 and `Q2` for `S`=1.

If `S` is a vector `QI` is a vector of UnitQuaternions, each element corresponding to sequential elements of `S`.

Options

'shortest'	Take the shortest path along the great circle

Notes

- This is a spherical linear interpolation (slerp) that can be interpretted as interpolation along a great circle arc on a sphere.
- It is an error if any element of `S` is outside the interval 0 to 1.

References

- Animating rotation with quaternion curves, K. Shoemake, in Proceedings of ACM SIGGRAPH, (San Francisco), pp. 245-254, 1985.

See also

ctraj

UnitQuaternion.inv

Invert a UnitQuaternion

`Q.inv()` is a **UnitQuaternion** object representing the inverse of Q. If Q is a vector $(1 \times N)$ the result is a vector of elementwise inverses.

See also

Quaternion.conj

UnitQuaternion.mrdivide

Divide unit quaternions

`R = Q1/Q2` is a **UnitQuaternion** object formed by Hamilton product of `Q1` and inv(`Q2`) where `Q1` and `Q2` are both UnitQuaternion objects.

Notes

- Overloaded operator '/'.
- If either, or both, of `Q1` or `Q2` are vectors, then the result is a vector.
  - if `Q1` is a vector $(1 \times N)$ then `R` is a vector $(1 \times N)$ such that `R`(i) = `Q1`(i)/`Q2`.
  - if `Q2` is a vector $(1 \times N)$ then `R` is a vector $(1 \times N)$ such that `R`(i) = `Q1`/`Q2`(i).
  - if both `Q1` and `Q2` are vectors $(1 \times N)$ then `R` is a vector $(1 \times N)$ such

that R(i) = Q1(i)/Q2(i).

See also

Quaternion.mtimes, Quaternion.mpower, Quaternion.plus, Quaternion.minus

---

UnitQuaternion.mtimes

Multiply UnitQuaternion's

`R = Q1*Q2` is a **UnitQuaternion** object formed by Hamilton product of `Q1` and `Q2` where `Q1` and `Q2` are both UnitQuaternion objects.

Q$*$V   is a vector (3×1) formed by rotating the vector V $(3 \times 1)$by the UnitQuaternion Q.

Notes

- Overloaded operator '*'
- If either, or both, of `Q1` or `Q2` are vectors, then the result is a vector.
  - if `Q1` is a vector $(1 \times N)$ then `R` is a vector $(1 \times N)$ such that R(i) = Q1(i)*Q2.
  - if `Q2` is a vector $(1 \times N)$ then `R` is a vector $(1 \times N)$ such that R(i) = Q1*Q2(i).
  - if both `Q1` and `Q2` are vectors $(1 \times N)$ then `R` is a vector $(1 \times N)$ such

that R(i) = Q1(i)*Q2(i).

See also

Quaternion.mrdivide, Quaternion.mpower, Quaternion.plus, Quaternion.minus

---

UnitQuaternion.new

Construct a new UnitQuaternion

`QN = Q.new()` constructs a new **UnitQuaternion** object of the same type as Q.

`QN = Q.new([S, V1, V2, V3])` as above but specified directly by its 4 elements.

`QN = Q.new(S, V)` as above but specified directly by the scalar `S` and vector part `V` (1×3)

Notes

- Polymorphic with Quaternion and RTBPose derived classes. For any of these instance objects the new method creates a new instance object of the same type.

UnitQuaternion.omega

Construct UnitQuaternion from angle times rotation vector

`Q = UnitQuaternion.omega(W)` is a **UnitQuaternion** representing rotation of ‖W‖ about the vector `W` (3×1).

Notes

- The input representation is known as exponential coordinates.

See also

UnitQuaternion.angvec

UnitQuaternion.plot

Plot a quaternion object

`Q.plot(options)` plots the **UnitQuaternion** as an oriented coordinate frame.

`H = Q.plot(options)` as above but returns a handle which can be used for animation.

Animation

Firstly, create a plot and keep the the handle as per above.

`Q.plot('handle', H)` updates the coordinate frame described by the handle `H` to the orientation of Q.

Options

'color',C	The color to draw the axes, MATLAB colorspec C
'frame',F	The frame is named {F} and the subscript on the axis la
'view',V for view toward origin of coordinate frame	Set plot view parameters V=[az el] angles, or 'auto'
'handle',h	Update the specified handle

These `options` are passed to trplot, see trplot for more `options`.

See also

trplot

UnitQuaternion.prod

Product of unit quaternions

`prod(Q)` is the product of the elements of the vector of **UnitQuaternion** objects `Q`.

Note

- Multiplication is performed with the .* operator, ie. the product is renormalized at every step.

See also

UnitQuaternion.times, RTBPose.prod

UnitQuaternion.q2r

Convert unit quaternion as vector to SO(3) rotation matrix

`UnitQuaternion.q2r(V)` is an SO(3) orthonormal rotation matrix (3×3) representing the same 3D orientation as the elements of the unit quaternion `V` (1×4).

Notes

- Is a static class method.

Reference

- Funda, Taylor, IEEE Trans. Robotics and Automation, 6(3), June 1990, pp.382-388.

See also **UnitQuaternion**.tr2q

UnitQuaternion.qvmul

Multiply unit quaternions defined by vector part

`QV = UnitQuaternion.QVMUL(QV1, QV2)` multiplies two unit-quaternions defined only by their vector components `QV1` and `QV2` (3×1). The result is similarly the vector component of the Hamilton product (3×1).

Notes

- Is a static class method.

See also

UnitQuaternion.tovec, UnitQuaternion.vec

UnitQuaternion.R

Convert to SO(3) rotation matrix

`R = Q.R()` is the equivalent SO(3) orthonormal rotation matrix (3×3). If Q represents a sequence $(N \times 1)$ then `R` is $3 \times 3 \times N$.

See also

UnitQuaternion.T, UnitQuaternion.SO3

UnitQuaternion.rand

Construct a random UnitQuaternion

`UnitQuaternion.rand()` is a **UnitQuaternion** representing a random 3D rotation.

References

- Planning Algorithms, Steve LaValle, p164.

See also

SO3.rand, SE3.rand

UnitQuaternion.rdivide

Divide unit quaternions and unitize

Q1./Q2 is a UnitQuaternion object formed by Hamilton product of Q1 and

`inv(Q2)` where Q1 and `Q2` are both **UnitQuaternion** objects. The result is explicitly unitized.

Notes

- Overloaded operator './'.

- If either, or both, of Q1 or `Q2` are vectors, then the result is a vector.

  – if Q1 is a vector $(1 \times N)$ then R is a vector $(1 \times N)$ such that R(i) = Q1(i)./`Q2`.

- if `Q2` is a vector ($1 \times N$) then R is a vector ($1 \times N$) such that R(i) = Q1./`Q2`(i).
- if both Q1 and `Q2` are vectors ($1 \times N$) then R is a vector ($1 \times N$) such

that R(i) = Q1(i)./`Q2`(i).

See also

Quaternion.mtimes

UnitQuaternion.rpy

Construct UnitQuaternion from roll-pitch-yaw angles

`Q = UnitQuaternion.rpy(ROLL, PITCH, YAW, OPTIONS)` is a **UnitQuaternion** representing rotation equivalent to the specified roll, pitch, yaw angles angles. These correspond to rotations about the Z, Y, X axes respectively.

`Q = UnitQuaternion.rpy(RPY, OPTIONS)` as above but the angles are given by the passed vector `RPY` = [`ROLL`, `PITCH`, `YAW`]. If `RPY` is a matrix ($N \times 3$) then `Q` is a vector ($1 \times N$) of UnitQuaternion objects where the index corresponds to rows of `RPY` which are assumed to be [`ROLL`,`PITCH`,`YAW`].

Options

'deg'	Compute angles in degrees (default radians)
'zyx'	Return solution for sequential rotations about Z, Y, X axes (default)
'xyz'	Return solution for sequential rotations about X, Y, Z axes
'yxz'	Return solution for sequential rotations about Y, X, Z axes

Notes

- Is vectorised, see rpy2r for details.

See also

UnitQuaternion.eul, rpy2r

UnitQuaternion.Rx

Construct UnitQuaternion from rotation about x-axis

`Q = UnitQuaternion.Rx(ANGLE)` is a **UnitQuaternion** representing rotation of `ANGLE` about the x-axis.

`Q = UnitQuaternion.Rx(ANGLE, 'deg')` as above but THETA is in degrees.

See also

UnitQuaternion.Ry, UnitQuaternion.Rz

UnitQuaternion.Ry

Construct UnitQuaternion from rotation about y-axis

`Q = UnitQuaternion.Ry(ANGLE)` is a **UnitQuaternion** representing rotation of `ANGLE` about the y-axis.

`Q = UnitQuaternion.Ry(ANGLE, 'deg')` as above but THETA is in degrees.

See also

UnitQuaternion.Rx, UnitQuaternion.Rz

UnitQuaternion.Rz

Construct UnitQuaternion from rotation about z-axis

`Q = UnitQuaternion.Rz(ANGLE)` is a **UnitQuaternion** representing rotation of `ANGLE` about the z-axis.

`Q = UnitQuaternion.Rz(ANGLE, 'deg')` as above but THETA is in degrees.

See also

UnitQuaternion.Rx, UnitQuaternion.Ry

UnitQuaternion.SE3

Convert to SE3 object

`Q.SE3()` is an SE3 object with equivalent rotation and zero translation.

Notes

- The translational part of the SE3 object is zero
- If Q is a vector then an equivalent vector of SE3 objects is created.

See also

UnitQuaternion.SE3, SE3

UnitQuaternion.SO3

Convert to SO3 object

`Q.SO3()` is an SO3 object with equivalent rotation.

Notes

- If Q is a vector then an equivalent vector of SO3 objects is created.

See also

UnitQuaternion.SE3, SO3

UnitQuaternion.T

Convert to homogeneous transformation matrix

`T = Q.T()` is the equivalent SE(3) homogeneous transformation matrix (4×4). If Q is a sequence $(N \times 1)$ then `T` is $4 \times 4 \times N$.

Notes:

- Has a zero translational component.

See also

UnitQuaternion.R, UnitQuaternion.SE3

UnitQuaternion.times

Multiply UnitQuaternion's and unitize

`R = Q1.*Q2` is a **UnitQuaternion** object formed by Hamilton product of `Q1` and `Q2`. The result is explicitly unitized.

Notes

- Overloaded operator '.*'
- If either, or both, of `Q1` or `Q2` are vectors, then the result is a vector.
  - if `Q1` is a vector ($1 \times N$) then `R` is a vector ($1 \times N$) such that `R`(i) = `Q1`(i).*`Q2`.
  - if `Q2` is a vector ($1 \times N$) then `R` is a vector ($1 \times N$) such that `R`(i) = `Q1`.*`Q2`(i).
  - if both `Q1` and `Q2` are vectors ($1 \times N$) then `R` is a vector ($1 \times N$) such

that `R`(i) = `Q1`(i).*`Q2`(i).

See also

Quaternion.mtimes

UnitQuaternion.toangvec

Convert to angle-vector form

`TH = Q.toangvec(OPTIONS)` is the rotational angle, about some vector, corresponding to this UnitQuaternion. If Q is a UnitQuaternion vector ($1 \times N$) then `TH` ($1 \times N$) and V ($N \times 3$).

`[TH,V] = Q.toangvec(OPTIONS)` as above but also returns a unit vector parallel to the rotation axis.

`Q.toangvec(OPTIONS)` prints a compact single line representation of the rotational angle and rotation vector corresponding to this UnitQuaternion. If Q is a UnitQuaternion vector then print one line per element.

Options

'deg'	Display/return angle in degrees rather than radians

Notes

- Due to the double cover of the UnitQuaternion, the returned rotation angles will be in the interval [-2π, 2π).

See also

UnitQuaternion.angvec

UnitQuaternion.toeul

Convert to roll-pitch-yaw angle form.

`EUL = Q.toeul(OPTIONS)` are the Euler angles (1×3) corresponding to the UnitQuaternion Q. These correspond to rotations about the Z, Y, Z axes respectively. `EUL` = [PHI,THETA,PSI].

If Q is a vector ($1 \times N$) then each row of `EUL` corresponds to an element of the vector.

Options

'deg'	Compute angles in degrees (radians default)

Notes

- There is a singularity for the case where THETA=0 in which case PHI is arbitrarily set to zero and PSI is the sum (PHI+PSI).

See also

UnitQuaternion.torpy, tr2eul

UnitQuaternion.torpy

Convert to roll-pitch-yaw angle form.

`RPY = Q.torpy(OPTIONS)` are the roll-pitch-yaw angles (1×3) corresponding to the UnitQuaternion Q. These correspond to rotations about the Z, Y, X axes respectively. `RPY` = [ROLL, PITCH, YAW].

If Q is a vector ($1 \times N$) then each row of `RPY` corresponds to an element of the vector.

Options

'deg'	Compute angles in degrees (radians default)
'xyz'	Return solution for sequential rotations about X, Y, Z axes
'yxz'	Return solution for sequential rotations about Y, X, Z axes

Notes

- There is a singularity for the case where P=$\pi/2$ in which case R is arbitrarily set to zero and Y is the sum (R+Y).

See also

UnitQuaternion.toeul, tr2rpy

UnitQuaternion.tovec

Convert to unique 3-vector

`V = Q.tovec()` is a vector (1×3) that uniquely represents the **UnitQuaternion**. The scalar component can be recovered by 1 - norm(`V`) and will always be positive.

Notes

- UnitQuaternions have double cover of SO(3) so the vector is derived from the UnitQuaternion with positive scalar component.

- This unique and concise vector representation of a UnitQuaternion is often used in bundle adjustment problems.

See also

UnitQuaternion.vec, UnitQuaternion.qvmul

UnitQuaternion.tr2q

Convert SO(3) or SE(3) matrix to unit quaternion as vector

`[S,V] = UnitQuaternion.tr2q(R)` is the scalar `S` and vector `V` (1×3) elements of a unit quaternion equivalent to the SO(3) rotation matrix `R` (3×3).

`[S,V] = UnitQuaternion.tr2q(T)` as above but for the rotational part of the SE(3) matrix `T` (4×4).

Notes

- Is a static class method.

Reference

- Funda, Taylor, IEEE Trans. Robotics and Automation, 6(3), June 1990, pp.382-388.

UnitQuaternion.unit

Unitize unit-quaternion

`QU = Q.unit()` is a **UnitQuaternion** with a norm of 1. If Q is a vector ($1 \times N$) then `QU` is also a vector ($1 \times N$).

Notes

- This is UnitQuaternion of unit norm, not a Quaternion of unit norm.

See also

Quaternion.norm

UnitQuaternion.vec

Construct UnitQuaternion from 3-vector

`Q = UnitQuaternion.vec(V)` is a **UnitQuaternion** constructed from just its vector component (1×3) and the scalar part is 1 - norm(`V`) and will always be positive.

Notes

- This unique and concise vector representation of a UnitQuaternion is often used in bundle adjustment problems.

See also

UnitQuaternion.tovec, UnitVector.qvmul

vex

Convert skew-symmetric matrix to vector

`V = VEX(S)` is the vector which has the corresponding skew-symmetric matrix `S`.

In the case that `S` (2×2) =

```
| 0   -v |
| v    0 |
```

then `V` = [v]. In the case that `S` (3 × 3) =

```
|  0   -vz    vy |
| vz     0   -vx |
|-vy    vx     0 |
```

then `V` = [vx; vy; vz].

Notes

- This is the inverse of the function SKEW().
- Only rudimentary checking (zero diagonal) is done to ensure that the matrix is actually skew-symmetric.
- The function takes the mean of the two elements that correspond to each unique element of the matrix.
- The matrices are the generator matrices for so(2) and so(3).

References

- Robotics, Vision & Control: Second Edition, P. Corke, Springer 2016; p25+43.

See also

skew, vexa

vexa

Convert augmented skew-symmetric matrix to vector

`V = VEXA(S)` is the vector which has the corresponding augmented skew-symmetric matrix `S`.

In the case that `S` (3 × 3) =

```
|  0   -v3   v1 |
| v3     0   v2 |
|  0     0    0 |
```

then `V` = [v1; v2; v3]. In the case that `S` (6 × 6) =

```
|  0   -v6    v5   v1 |
| v6     0   -v4   v2 |
|-v5    v4     0   v3 |
|  0     0     0    0 |
```

then `V` = [v1; v2; v3; v4; v5; v6].

Notes

- This is the inverse of the function SKEWA().
- The matrices are the generator matrices for se(2) and se(3). The elements comprise the equivalent twist vector.

References

- Robotics, Vision & Control: Second Edition, Chap 2, P. Corke, Springer 2016.

See also

skewa, vex, Twist

---

xyzlabel

Label X, Y and Z axes

`XYZLABEL()` label the x-, y- and z-axes with 'X', 'Y', and 'Z'respectiveley.

`XYZLABEL(FMT)` as above but pass in a format string where %s is substituted for the axis label, eg.

```
xyzlabel('This is the %s axis')
```

See also

xlabel, ylabel, zlabel, sprintf

Bibliography