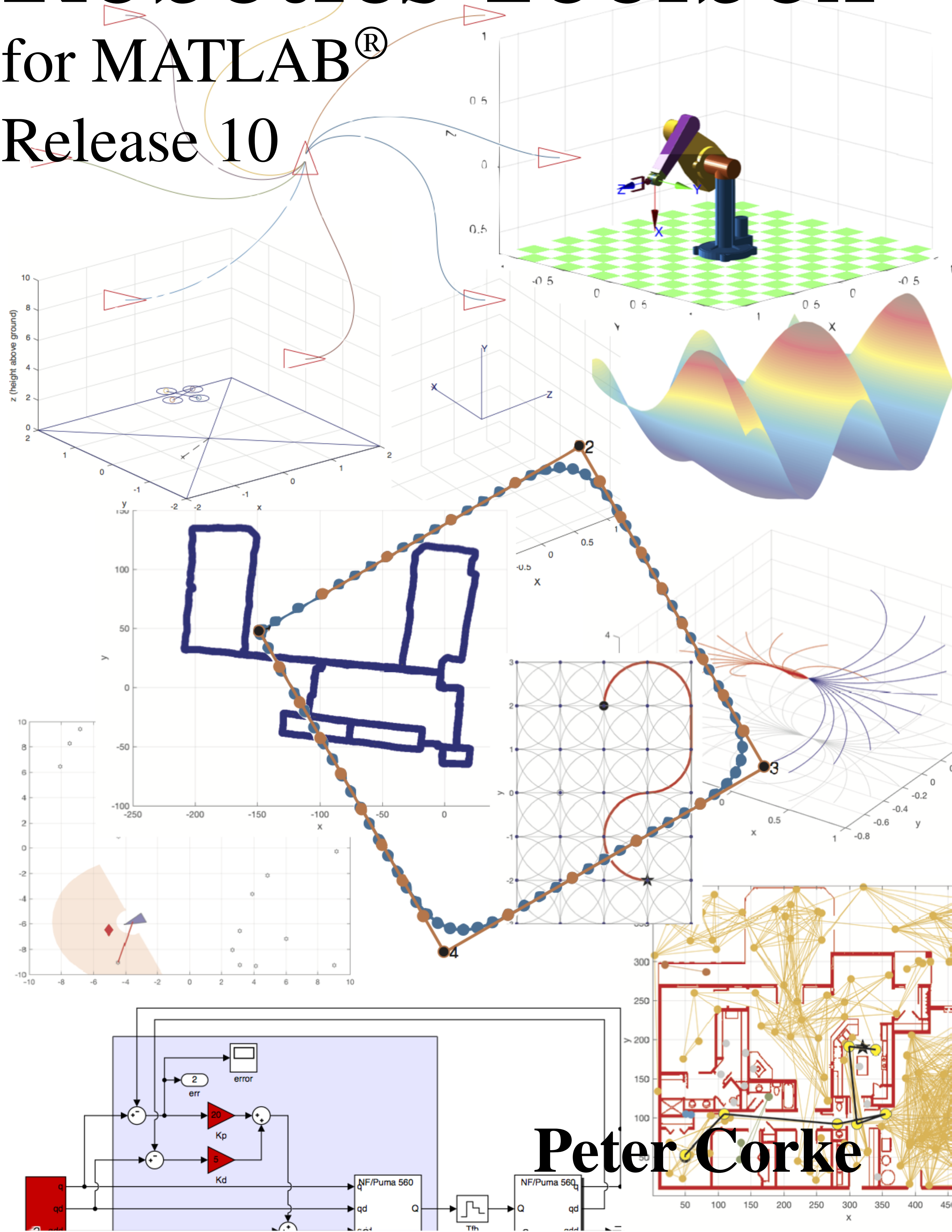
Robotics Toolbox

for MATLAB®

Release 10

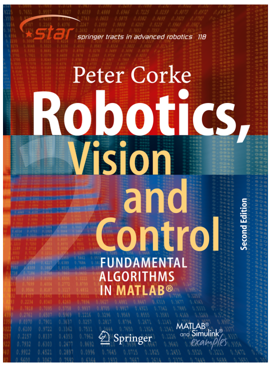


Peter Corke

Release	10.4
Release date	September 2020
Licence	LGPL
Toolbox home page	http://www.petercorke.com/robot
Discussion group	http://groups.google.com.au/group/robotics-tool-box

peter.i.corke@gmail.com
http://www.petercorke.com

Preface



This, the tenth major release of the Toolbox, representing over twenty five years of continuous development and a substantial level of maturity. This version corresponds to the **second edition** of the book "*Robotics, Vision & Control*" published in June 2017 – RVC2.

This MATLAB® Toolbox has a rich collection of functions that are useful for the study and simulation of robots: arm-type robot manipulators and mobile robots. For robot manipulators, functions include kinematics, trajectory generation, dynamics and control. For mobile robots, functions include path planning, kinodynamic planning, localization, map building and simultaneous localization and mapping (SLAM).

The Toolbox makes strong use of classes to represent robots and such things as sensors and maps. It includes Simulink® models to describe the evolution of arm or mobile robot state over time for a number of classical control strategies. The Toolbox also provides 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.

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, second edition*" 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.

Functions by category

Contents

Preface 2
Functions by category 5

1 Introduction **13**
1.1 Changes in RTB 10 13
1.1.1 Incompatible changes 13
1.1.2 New features 14
1.1.3 Enhancements 15
1.2 Changes in RTB 10.3 17
1.3 Changes in RTB 10.2 19
1.4 How to obtain the Toolbox 19
1.4.1 From .mltbx file 19
1.4.2 From .zip file 20
1.4.3 MATLAB Online™ 20
1.4.4 Simulink® 21
1.4.5 Notes on implementation and versions 24
1.4.6 Documentation 24
1.5 Compatible MATLAB versions 25
1.6 Use in teaching 25
1.7 Use in research 25
1.8 Support 25
1.9 Related software 26
1.9.1 Robotics System Toolbox™ 26
1.9.2 Octave 26
1.9.3 Machine Vision toolbox 26
1.10 Contributing to the Toolboxes 27
1.11 Acknowledgements 27

2 Functions and classes **29**
Astar 29
AstarMOO 35
AstarPO 41
Bicycle 47
Bug2 51
ccodefunctionstring 53
chi2inv_rtb 55
ctraj 55
delta2tr 56

DHFactor . . . 56
distancexform . . . 58
distributeblocks . . . 59
doesblockexist . . . 59
Dstar . . . 60
DstarMOO . . . 64
DstarPO . . . 70
Dubbins . . . 77
DXform . . . 78
EKF . . . 82
ETS . . . 90
ETS2 . . . 92
ETS3 . . . 101
Frame . . . 110
getprofilefunctionstats . . . 112
joy2tr . . . 113
joystick . . . 113
jsingu . . . 114
jtraj . . . 114
LandmarkMap . . . 115
Lattice . . . 118
bresenham . . . 121
circle . . . 121
colorname . . . 122
diff2 . . . 123
dockfigs . . . 123
edgelist . . . 123
filt1d . . . 124
gaussfunc . . . 125
mmlabel . . . 125
mplot . . . 126
mtools . . . 127
pickregion . . . 127
plotp . . . 128
polydiff . . . 128
Polygon . . . 128
randinit . . . 135
runscript . . . 135
rvcpath . . . 136
stlRead . . . 136
usefig . . . 137
xaxis . . . 137
yaxis . . . 138
about . . . 138
angdiff . . . 139
angvec2r . . . 139
angvec2tr . . . 140
Animate . . . 140
circle . . . 143
colnorm . . . 143

delta2tr 144
e2h 144
eul2jac 145
eul2r 145
eul2tr 146
h2e 147
homline 147
homtrans 147
ishomog 148
ishomog2 149
isrot 149
isrot2 150
isunit 150
isvec 151
lift23 151
numcols 152
numrows 152
oa2r 152
oa2tr 153
PGraph 154
plot2 172
plot_arrow 173
plot_box 173
plot_circle 175
plot_ellipse 176
plot_homline 177
plot_point 178
plot_poly 179
plot_ribbon 180
plot_sphere 181
plotvol 182
Plucker 182
Quaternion 192
r2t 204
randinit 204
rot2 205
rotx 205
roty 206
rotz 206
rpy2jac 206
rpy2r 207
rpy2tr 208
rt2tr 209
RTBPose 210
SE2 223
SE3 232
skew 251
skewa 252
SO2 253
SO3 261

SpatialAcceleration . . . 278
SpatialF6 . . . 280
SpatialForce . . . 281
SpatialInertia . . . 282
SpatialM6 . . . 285
SpatialMomentum . . . 287
SpatialVec6 . . . 288
SpatialVelocity . . . 292
stlRead . . . 293
t2r . . . 294
tb_optparse . . . 294
tr2angvec . . . 296
tr2delta . . . 297
tr2eul . . . 298
tr2jac . . . 298
tr2rpy . . . 299
tr2rt . . . 300
tranimate . . . 300
tranimate2 . . . 301
transl . . . 302
transl2 . . . 303
trchain . . . 304
trchain2 . . . 305
trexp . . . 306
trexp2 . . . 307
trinterp . . . 308
trinterp2 . . . 309
trlog . . . 310
trnorm . . . 311
trot2 . . . 311
trotx . . . 312
troty . . . 312
trotz . . . 313
trplot . . . 313
trplot2 . . . 315
trprint . . . 316
trprint2 . . . 317
trscale . . . 317
Twist . . . 318
unit . . . 324
UnitQuaternion . . . 325
vex . . . 347
vexa . . . 348
xyzlabel . . . 349
Link . . . 349
lspb . . . 360
makemap . . . 361
models . . . 362
mdl_ball . . . 362
mdl_baxter . . . 363

mdl_cobra600 . . . 364
mdl_coil . . . 364
mdl_fanuc10L . . . 365
mdl_hyper2d . . . 366
mdl_hyper3d . . . 366
mdl_irb140 . . . 367
mdl_irb140_mdh . . . 368
mdl_jaco . . . 369
mdl_KR5 . . . 370
mdl_LWR . . . 370
mdl_M16 . . . 371
mdl_mico . . . 372
mdl_motomanHP6 . . . 373
mdl_nao . . . 373
mdl_offset6 . . . 374
mdl_onelink . . . 375
mdl_p8 . . . 376
mdl_panda . . . 376
mdl_phantomx . . . 377
mdl_planar1 . . . 378
mdl_planar2 . . . 378
mdl_planar2_sym . . . 379
mdl_planar3 . . . 380
mdl_puma560 . . . 380
mdl_puma560akb . . . 381
mdl_quadrotor . . . 382
mdl_S4ABB2p8 . . . 383
mdl_sawyer . . . 384
mdl_simple6 . . . 384
mdl_stanford . . . 385
mdl_stanford_mdh . . . 386
mdl_twolink . . . 386
mdl_twolink_mdh . . . 387
mdl_twolink_sym . . . 388
mdl_ur10 . . . 389
mdl_ur3 . . . 390
mdl_ur5 . . . 390
mstraj . . . 391
mtraj . . . 392
multidfprintf . . . 393
Navigation . . . 394
ParticleFilter . . . 402
plot_vehicle . . . 407
plotbotopt . . . 409
PoseGraph . . . 409
Prismatic . . . 410
PrismaticMDH . . . 412
PRM . . . 415
purepursuit . . . 418
qplot . . . 419

RandomPath 419
RangeBearingSensor 422
ReedsShepp 427
Revolute 428
RevoluteMDH 430
RobotArm 433
RRT 437
rtbdemo 441
RTBPlot 441
Sensor 442
simulinkext 444
startup_rtb 445
sym2 445
symexpr2slblock 446
test_jacob_dot 447
tpoly 447
Unicycle 448
Vehicle 452
wtrans 460

Chapter 1

Introduction

1.1 Changes in RTB 10

RTB 10 is largely backward compatible with RTB 9.

1.1.1 Incompatible changes

- The class `Vehicle` no longer represents an Ackerman/bicycle vehicle model. `Vehicle` is now an abstract superclass of `Bicycle` and `Unicycle` which represent car-like and differentially-steered vehicles respectively.
- The class `LandmarkMap` replaces `PointMap`.
- Robot-arm forward kinematics now returns an `SE3` object rather than a 4×4 matrix.
- The `Quaternion` class used to represent both unit and non-unit quaternions which was untidy and confusing. They are now represented by two classes `UnitQuaternion` and `Quaternion`.
- The method to compute the arm-robot Jacobian in the end-effector frame has been renamed from `jacobn` to `jacobe`.
- The path planners, subclasses of `Navigation`, the method to find a path has been renamed from `path` to `query`.
- The Jacobian methods for the `RangeBearingSensor` class have been renamed to `Hx`, `Hp`, `Hw`, `Gx`,`Gz`.
- The function `se2` has been replaced with the class `SE2`. On some platforms (Mac) this is the same file. Broadly similar in function, the former returns a 3×3 matrix, the latter returns an object.
- The function `se3` has been replaced with the class `SE3`. On some platforms (Mac) this is the same file. Broadly similar in function, the former returns a 4×4 matrix, the latter returns an object.

RTB 9	RTB 10
Vehicle	Bicycle
Map	LandmarkMap
jacobn	jacobe
path	query
H_x	Hx
H_xf	Hp
H_w	Hw
G_x	Gx
G_z	Gz

Table 1.1: Function and method name changes

These changes are summarized in Table 1.1.

1.1.2 New features

- `SerialLinkplot3d()` renders realistic looking 3D models of robots. STL models from the package ARTE by Arturo Gil (https://arvc.umh.es/arte) are now included with RTB, by kind permission.
- `ETS2` and `ETS3` packages provide a gentle (non Denavit-Hartenberg) introduction to robot arm kinematics, see Chapter 7 for details.
- Distribution as an `.mltbx` format file.
- A comprehensive set of functions to handle rotations and transformations in 2D, these functions end with the suffix 2, eg. `transl2`, `rot2`, `trot2` etc.
- Matrix exponentials are handled by `trexp`, `trlog`, `trexp2` and `trlog2`.
- The class `Twist` represents a twist in 3D or 2D. Respectively, it is a 6-vector representation of the Lie algebra $se(3)$, or a 3-vector representation of $se(2)$.
- The method `SerialLink.jointdynamics` returns a vector of `tf` objects representing the dynamics of the joint actuators.
- The class `Lattice` is a simple kino-dynamic lattice path planner.
- The class `PoseGraph` solves graph relaxation problems and can be used for bundle adjustment and pose graph SLAM.
- The class `Plucker` represents a line using Plúcker coordinates.
- The folder `RST` contains Live Scripts that demonstrate some capabilities of the MATLAB Robotics System Toolbox™.
- The folder `symbolic` contains Live Scripts that demonstrate use of the MATLAB Symbolic Math Toolbox™ for deriving Jacobians used in EKF SLAM (vehicle and sensor), inverse kinematics for a 2-joint planar arm and solving for roll-pitch-yaw angles given a rotation matrix.
- All the robot models, prefixed by `mdl_`, now reside in the folder `models`.

- New robot models include Universal Robotics UR3, UR5 and UR10; and Kuka light weight robot arm.
- A new folder `data` now holds various data files as used by examples in RVC2: STL models, occupancy grids, Hershey font, Toro and G2O data files.

Since its inception RTB has used matrices[1] to represent rotations and transformations in 2D and 3D. A trajectory, or sequence, was represented by a 3-dimensional matrix, eg. $4 \times 4 \times N$. 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[2]. 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[3]. 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
>> 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
```

1.1.3 Enhancements

- Dependencies on the Machine Vision Toolbox for MATLAB (MVTB) have been removed. The fast dilation function used for path planning is now searched for in MVTB and the MATLAB Image Processing Toolbox (IPT) and defaults to a provided M-function.
- A major pass over all code and method/function/class documentation.
- Reworking and refactoring all the manipulator graphics, work in progress.
- An “app" is included: `tripleangle` which allows graphical experimentation with Euler and roll-pitch-yaw angles.
- A tidyup of all Simulink models. Red blocks now represent user settable parameters, and shaded boxes are used to group parts of the models.

[1] Early versions of RTB, before 1999, used vectors to represent quaternions but that changed to an object once objects were added to the language.

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

[3] 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.

Orientation		Pose	
Classic	**New**	**Classic**	**New**
`rot2`	`SO2`	`trot2`	`SE2`
		`transl2`	`SE2`
`trplot2`	`.plot`	`trplot2`	`.plot`
`rotx, roty, rotz`	`SO3.Rx, SO3.Ry, SO3.Rz`	`trotx, troty, trotz`	`SE3.Rx, SE3.Ry, SE3.Rz`
		`T = transl(v)`	`SE3(v)`
`eul2r, rpy2r`	`SO3.eul, SO3.rpy`	`eul2tr, rpy2tr`	`SE3.eul, SE3.rpy`
`angvec2r`	`SO3.angvec`	`angvec2tr`	`SE3.angvec`
`oa2r`	`SO3.oa`	`oa2tr`	`SE3.oa`
		`v = transl(T)`	`.t, .transl`
`tr2eul, tr2rpy`	`.toeul, .torpy`	`tr2eul, tr2rpy`	`.toeul, .torpy`
`tr2angvec`	`.toangvec`	`tr2angvec`	`.toangvec`
`trexp`	`SO3.exp`	`trexp`	`SE3.exp`
`trlog`	`.log`	`trlog`	`.log`
`trplot`	`.plot`	`trplot`	`.plot`

Functions starting with dot are methods on the new objects. You can use them in functional form `toeul(R)` or in dot form `R.toeul()` or `R.toeul`. It's a personal preference. The trailing parentheses are not required if no arguments are passed, but it is a useful convention and reminder that you that you are invoking a method not reading a property. The old function `transl` appears twice since it maps a vector to a matrix as well as the inverse.

	Output type										
Input type	t	Euler	RPY	θ, v	R	T	Twist vector	Twist	Unit-Quaternion	SO3	SE3
t (3-vector)						`transl`		`Twist('T')`			`SE3()`
Euler (3-vector)					`eul2r`	`eul2tr`			`UnitQuaternion.eul()`	`SO3.eul()`	`SE3.eul()`
RPY (3-vector)					`rpy2r`	`rpy2tr`			`UnitQuaternion.rpy()`	`SO3.rpy()`	`SE3.rpy()`
θ, v (scalar + 3-vector)					`angvec2r`	`angvec2tr`			`UnitQuaternion.angvec()`	`SO3.angvec()`	`SE3.angvec()`
R (3×3 matrix)		`tr2eul`	`tr2rpy`	`tr2angvec`		`r2t`	`trlog`		`UnitQuaternion()`	`SO3()`	`SE3()`
T (4×4 matrix)	`transl`	`tr2eul`	`tr2rpy`	`tr2angvec`	`t2r`		`trlog`	`Twist()`	`UnitQuaternion()`	`SO3()`	`SE3()`
Twist vector (3- or 6-vector)					`trexp`	`trexp`		`Twist()`		`SO3.exp()`	`SE3.exp()`
Twist						`.T`	`.S`				`.SE`
Unit-Quaternion		`.toeul`	`.torpy`	`.toangvec`	`.R`	`.T`				`.SO3`	`.SE3`
SO3		`.toeul`	`.torpy`	`.toangvec`	`.R`	`.T`	`.log`		`.UnitQuaternion`		`.SE3`
SE3	`.t`	`.toeul`	`.torpy`	`.toangvec`	`.R`	`.T`	`.log`	`.Twist`	`.UnitQuaternion`	`.SO3`	

Dark grey boxes are not possible conversions. Light grey boxes are possible conversions but the Toolbox has no direct conversion, you need to convert via an intermediate type. Red text indicates classical Robotics Toolbox functions that work with native MATLAB® vectors and matrices. Class.type() indicates a static factory method that constructs a Class object from input of that type. Functions shown starting with a dot are a method on the class corresponding to that row.

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*"

- RangeBearingSensor animation
- All the java code that supports the `DHFactor` functionality now lives in the folder `java`. The `Makefile` in there can be used to recompile the code. There are java version issues and the shipped class files are built to java 1.7 which allows operation

1.2 Changes in RTB 10.3

This release includes minor new features and a number of bug fixes compared to 10.2:

- Serial-link manipulators
  - The Symbolic Robot Modeling Toolbox component by Jörn Malzahn has been updated. It offers amazing speedups by using symbolic algebra to create robot specific MATLAB code or MEX files and it can even generate optimised Simulink blocks. I've seen speedups of over 50,000x. You need to have the Symbolic Math Toolbox.
  - New robot kinematic models: Franka-Emika PANDA and Rethink Sawyer.
  - Methods `DH` and `MDH` on the `SerialLink` class convert models between DH and MDH kinematics. Dynamics not yet supported.
  - `plot3d` behaves like `plot` for the `'trail'` and `'movie'` options.
  - Experimental feature: Manipulator configuration (joint angle) vectors can be kept *inside* the `SerialLink` object. At constructor time the option `'configs', {'qz', qz, 'qr', qr}` adds these two configurations to the class instance, and they can be referenced later as, for example, `p560.qz`. This reduces the number of workspace variables and confusion when working with several robots at the same time.
  - Fix bug in the `'trail'` option for `SerialLink.plot`.

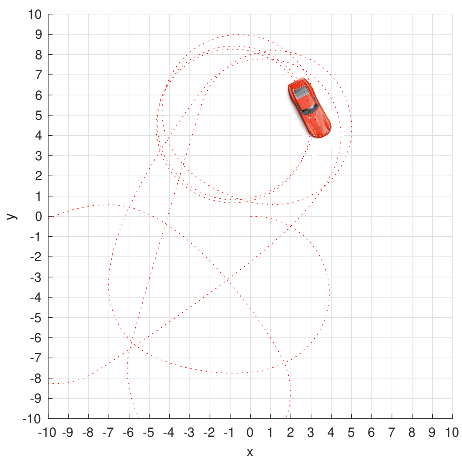


Figure 1.2: Car animation drawn with `demos/car_anim` using `plot_vehicle`

    - Fixed bug in `ikunc`, `ikcon` which ignored `q0`.

- Mobile robotics
    - Added the ability to animate a picture of a vehicle to `plot_vehicle`, see `demos/car_demo` and Figure 1.2. Also added a `'trail'` feature, and updated documentation.
    - Experimental feature : A Reeds-Shepp path planner, see `rReedsShepp.m` and `demos/reedsshepp.mlx`, this is not (yet) properly integrated into the `Navigation` class architecture.
- Simulink
    - Simulink blocks for Euler angles now have a checkbox to allow degrees mode.
    - Simulink blocks for roll-pitch-yaw angles now have a checkbox to allow degrees mode and radio buttons to select the angle sequence.
    - New Simulink block for `mstraj` gives full access to all capabilities of that function.
    - A folder `simulink/R2015a` contains all the Simulink models exported as `.slx` files for Simulink R2015a. This might ease problems for those using older versions of Simulink on the models in the top folder, many of which have been edited and saved under R2018a. Check the README file for details.
- A new script `rvccheck` which attempts to diagnose installation and MATLAB path issues.
- The `demos` folder now includes LiveScript versions of each demo, these are `.mlx` files. I've done a first pass at formatting the content and in a few cases updating the content a little. From here on, the `.m` files are deprecated. You need MATLAB 2016a or later to run the LiveScripts.
- Major tidyup and documentation improvements for the `Twist` and `Plucker` objects.
- Changes to the `RTBPose.mtimes` method which now allows you to:
    - postmultiply an `SE3` object by a `Plucker` object which returns a `Plucker` object. This applies a rigid-body transformation to the line in space.
    - postmultiply an `SE2` object by a MATLAB `polyshape` object which returns a `polyshape` object. This applies a rigid-body transformation to the polygon.
- Added a `disp` method to various toolbox objects, invokes `display`, which provides a display of the type from within the debugger.
- `Quaternion` == operator
- `UnitQuaternion` == accounts for double mapping
- `UnitQuaternion` has a `rand` method that generates a randomly distributed rotation, also used by `SO3.rand` and `SE3.rand`.

- `tr2rpy` fixed a long standing bug with the pitch angle in certain corner cases, the pitch angle now lies in the range $[-\pi, +\pi)$.
- Remove dependency on `numrows()` and `numcols()` for `rt2tr`, `tr2rt`, `transl`, `transl2` which simplifies standalone operation.
- A campaign to reduce the size of the RTB distribution file:
  - `tripleangle` uses updated STL files with reduced triangle counts for faster loading.
  - This manual is compressed.
  - Removal of extraneous files.
- Options to RTB functions can now be strings or character arrays, ie. `rotx(45, 'deg')` or `rotx(45, "deg")`. If you don't yet know about MATLAB strings (with double quotes) check them out.
- General tidyup to code and documentation, added missing files from earlier releases.

1.3 Changes in RTB 10.2

This release has a relatively small number of bug fixes compared to 10.1:

- Fixed bugs in `jacobe` and `coriolis` when using symbolic arguments.
- New robot models: UR3, UR5, UR10, LWR.
- Fixed bug for `interp` method of `SE3` object.
- Fixed bug with detecting Optimisation Toolbox for `ikcon` and `ikunc`.
- Fixed bug in `ikine_sym`.
- Fixed various bugs related to plotting robots with prismatic joints.

1.4 How to obtain the Toolbox

The Robotics Toolbox is freely available from the Toolbox home page at

http://www.petercorke.com

The file is available in MATLABtoolbox format (`.mltbx`) or zip format (`.zip`).

1.4.1 From .mltbx file

Since MATLAB R2014b toolboxes can be packaged as, and installed from, files with the extension `.mltbx`. Download the most recent version of `robot.mltbx` or `vision.mltbx` to your computer. Using MATLAB navigate to the folder where you downloaded the file and double-click it (or right-click then select Install). The

Toolbox will be installed within the local MATLAB file structure, and the paths will be appropriately configured for this, and future MATLAB sessions.

1.4.2 From .zip file

Download the most recent version of robot.zip or vision.zip to your computer. Use your favourite unarchiving tool to unzip the files that you downloaded. To add the Toolboxes to your MATLAB path execute the command

```
>> addpath RVCDIR ;
>> startup_rvc
```

where `RVCDIR` is the full pathname of the folder where the folder `rvctools` was created when you unzipped the Toolbox files. The script `startup_rvc` adds various subfolders to your path and displays the version of the Toolboxes. After installation the files for both Toolboxes reside in a top-level folder called `rvctools` and beneath this are a number of folders:

`robot`	The Robotics Toolbox
`vision`	The Machine Vision Toolbox
`common`	Utility functions common to the Robotics and Machine Vision Toolboxes
`simulink`	Simulink blocks for robotics and vision, as well as examples
`contrib`	Code written by third-parties

If you already have the Machine Vision Toolbox installed then download the zip file to the folder above the existing `rvctools` directory, and then unzip it. The files from this zip archive will properly interleave with the Machine Vision Toolbox files.

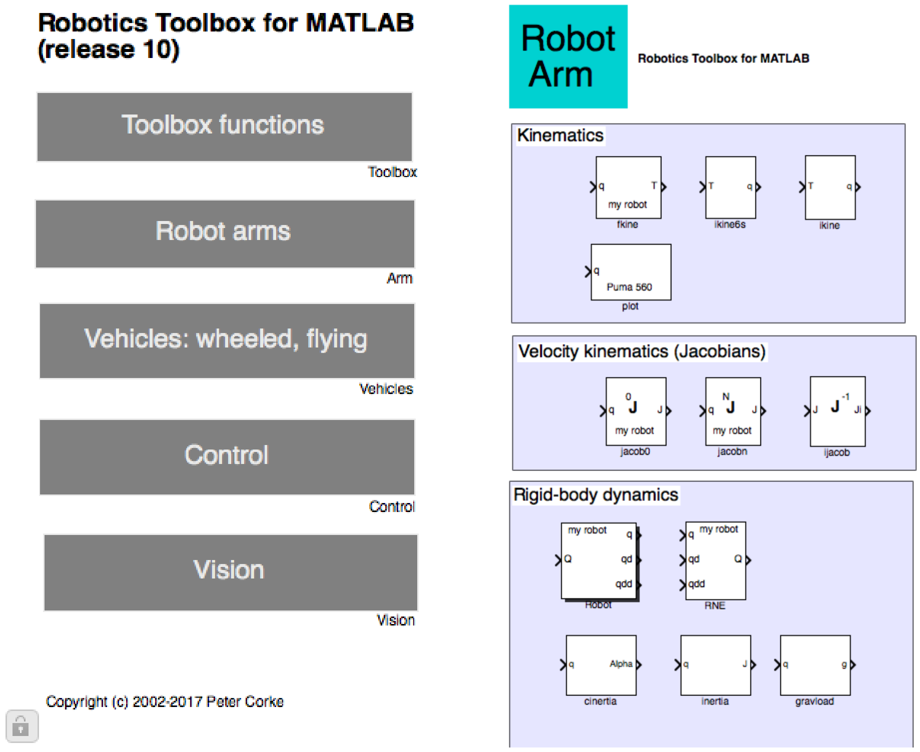
You need to setup the path every time you start MATLAB but you can automate this by setting up environment variables, editing your `startup.m` script, using `pathtool` and saving the path, or by pressing the "Update Toolbox Path Cache" button under MATLAB General preferences. You can check the path using the command `path` or `pathtool`.

A menu-driven demonstration can be invoked by

```
>> rtbdemo
```

1.4.3 MATLAB Online^{TM}

The Toolbox works well with MATLAB Online^{TM} which lets you access a MATLAB session from a web browser, tablet or even a phone. The key is to get the RTB files into the filesystem associated with your Online account. The easiest way to do this is to install MATLAB Drive^{TM} from MATLAB File Exchange or using the Get Add-Ons option from the MATLAB GUI. This functions just like Google Drive or Dropbox, a local filesystem on your computer is synchronized with your MATLAB Online account. Copy the RTB files into the local MATLAB Drive cache and they will soon be synchronized, invoke `startup_rvc` to setup the paths and you are ready to simulate robots on your mobile device or in a web browser.



(a) (b)

Figure 1.3: The Robotics Toolbox blockset.

1.4.4 Simulink®

Simulink® is the block-diagram-based simulation environment for MATLAB. It provides a very convenient way to create and visualize complex dynamic systems, and is particularly applicable to robotics. RTB includes a library of blocks for use in constructing robot kinematic and dynamic models. The block library is opened by

```
>> roblocks
```

and a window like that shown in Figure 1.3(a) will be displayed. Double click a particular category and it will expand into a palette of blocks, like Figure 1.3(b), that can be dragged into your model.

Users with no previous Simulink experience are advised to read the relevant Mathworks manuals and experiment with the examples supplied. Experienced Simulink users should find the use of the Robotics blocks quite straightforward. Generally there is a one-to-one correspondence between Simulink blocks and Toolbox functions. Several demonstrations have been included with the Toolbox in order to illustrate common topics in robot control and demonstrate Toolbox Simulink usage. These could be considered as starting points for your own work, just select the model closest to what you want and start changing it. Details of the blocks can be found using the File/ShowBrowser option on the block library window.

Arm robots	
`Robot`	represents a robot, with generalized joint force input and joint coordinates, velocities and accelerations as outputs. The parameters are the robot object to be simulated and the initial joint angles. It is similar to the `fdyn()` function and represents the forward dynamics of the robot.
`rne`	computes the inverse dynamics using the recursive Newton-Euler algorithm (function `rne`). Inputs are joint coordinates, velocities and accelerations and the output is the generalized joint force. The robot object is a parameter.
`cinertia`	computes the manipulator Cartesian inertia matrix. The parameters are the robot object to be simulated and the initial joint angles.
`inertia`	computes the manipulator joint-space inertia matrix. The parameters are the robot object to be simulated and the initial joint angles.
`inertia`	computes the gravity load. The parameters are the robot object to be simulated and the initial joint angles.
`jacob0`	outputs a manipulator Jacobian matrix, with respect to the world frame, based on the input joint coordinate vector. outputs the Jacobian matrix. The robot object is a parameter.
`jacobn`	outputs a manipulator Jacobian matrix, with respect to the end-effector frame, based on the input joint coordinate vector. outputs the Jacobian matrix. The robot object is a parameter.
`ijacob`	inverts a Jacobian matrix. Currently limited to square Jacobians only, ie. for 6-axis robots.
`fkine`	outputs a homogeneous transformation for the pose of the end-effector corresponding to the input joint coordinates. The robot object is a parameter.
`plot`	creates a graphical animation of the robot in a new window. The robot object is a parameter.
Mobile robots	
`Bicycle`	is the kinematic model of a mobile robot that uses the bicycle model. The inputs are speed and steer angle and the outputs are position and orientation.
`Unicycle`	is the kinematic model of a mobile robot that uses the unicycle, or differential steering, model. The inputs are speed and turn raate and the outputs are position and orientation.
`Quadrotor`	is the dynamic model of a quadrotor. The inputs are rotor speeds and the output is translational and angular position and velocity. Parameter is a quadrotor structure.
`N-rotor`	is the dynamic model of a N-rotor flyer. The inputs are rotor speeds and the output is translational and angular position and velocity. Parameter is a quadrotor structure.
`ControlMixer`	accepts thrust and torque commands and outputs rotor speeds for a quadrotor.
`Quadrotor plot`	creates a graphical animation of the quadrotor in a new window. Parameter is a quadrotor structure.
Trajectory	

`jtraj`	outputs coordinates of a point following a quintic polynomial as a function of time, as well as its derivatives. Initial and final velocity are assumed to be zero. The parameters include the initial and final points as well as the overall motion time.
`lspb`	outputs coordinates of a point following an LSPB trajectory as a function of time. The parameters include the initial and final points as well as the overall motion time.
`circle`	outputs the xy-coordinates of a point around a circle. Parameters are the centre, radius and angular frequency.
Vision	
`camera`	input is a camera pose and the output is the coordinates of points projected on the image plane. Parameters are the camera object and the point positions.
`camera2`	input is a camera pose and point coordinate frame pose, and the output is the coordinates of points projected on the image plane. Parameters are the camera object and the point positions relative to the point frame.
`image Jacobian`	input is image points and output is the point feature Jacobian. Parameter is the camera object.
`image Jacobian sphere`	input is image points in spherical coordinates and output is the point feature Jacobian. Parameter is a spherical camera object. computes camera pose from image points. Parameter is the camera object.
`Pose estimation`	computes camera pose from image points. Parameter is the camera object.
Miscellaneous	
`Inverse`	outputs the inverse of the input matrix.
`Pre multiply`	outputs the input homogeneous transform pre-multiplied by the constant parameter.
`Post multiply`	outputs the input homogeneous transform post-multiplied by the constant parameter.
`inv Jac`	inputs are a square Jacobian $\mathbf{J}$ and a spatial velocity ν and outputs are $\mathbf{J}^{-1}$ and the condition number of $\mathbf{J}$.
`pinv Jac`	inputs are a Jacobian $\mathbf{J}$ and a spatial velocity ν and outputs are $\mathbf{J}^{+}$ and the condition number of $\mathbf{J}$.
`tr2diff`	outputs the difference between two homogeneous transformations as a 6-vector comprising the translational and rotational difference.
`xyz2T`	converts a translational vector to a homogeneous transformation matrix.
`rpy2T`	converts a vector of roll-pitch-yaw angles to a homogeneous transformation matrix.
`eul2T`	converts a vector of Euler angles to a homogeneous transformation matrix.
`T2xyz`	converts a homogeneous transformation matrix to a translational vector.
`T2rpy`	converts a homogeneous transformation matrix to a vector of roll-pitch-yaw angles.

T2eul	converts a homogeneous transformation matrix to a vector of Euler angles.
angdiff	computes the difference between two input angles modulo 2π.

A number of models are also provided:

Robot manipulator arms	
sl_rrmc	Resolved-rate motion control
sl_rrmc2	Resolved-rate motion control (relative)
sl_ztorque	Robot collapsing under gravity
sl_jspace	Joint space control
sl_ctorque	Computed torque control
sl_fforward	Torque feedforward control
sl_opspace	Operational space control
sl_sea	Series-elastic actuator
vloop_test	Puma 560 velocity loop
ploop_test	Puma 560 position loop
Mobile ground robot	
sl_braitenberg	Braitenberg vehicle moving to a source
sl_lanechange	Lane changing control
sl_drivepoint	Drive to a point
sl_driveline	Drive to a line
sl_drivepose	Drive to a pose
sl_pursuit	Drive along a path
Flying robot	
sl_quadrotor	Quadrotor control
sl_quadrotor_vs	Control visual servoing to a target

1.4.5 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.

1.4.6 Documentation

This document `robot.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.

1.5 Compatible MATLAB versions

The Toolbox has been tested under R2019b and R2020aPRE. Compatibility problems are increasingly likely the older your version of MATLAB is.

1.6 Use in teaching

This is definitely encouraged! You are free to put the PDF manual (`robot.pdf` or the web-based documentation `html/*.html` on a server for class use. If you plan to distribute paper copies of the PDF manual then every copy must include the first two pages (cover and licence).

Link to other resources such as MOOCs or the Robot Academy can be found at `www.petercorke.com/moocs`.

1.7 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.

1.8 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.

1.9 Related software

1.9.1 Robotics System Toolbox™

The Robotics System Toolbox™ (RST) from MathWorks is an official and supported product. System toolboxes (see also the Computer Vision System Toolbox) are aimed at developers of systems. RST has a growing set of functions for mobile robots, arm robots, ROS integration and pose representations but its design (classes and functions) and syntax is quite different to RTB. A number of examples illustrating the use of RST are given in the folder `RST` as Live Scripts (extension `.mlx`), but you need to have the Robotics System Toolbox™ installed in order to use it.

1.9.2 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.

1.9.3 Machine Vision toolbox

Machine Vision toolbox (MVTB) for MATLAB. This was described in an article

```
@article{Corke05d,
        Author = {P.I. Corke},
        Journal = {IEEE Robotics and Automation Magazine},
        Month = nov,
        Number = {4},
        Pages = {16-25},
        Title = {Machine Vision Toolbox},
        Volume = {12},
        Year = {2005}}
```

and provides a very wide range of useful computer vision functions and is used to illustrate principals in the Robotics, Vision & Control book. You can obtain this from http://www.petercorke.com/vision. More recent products such as MATLAB Image Processing Toolbox and MATLAB Computer Vision System Toolbox provide functionality that overlaps with MVTB.

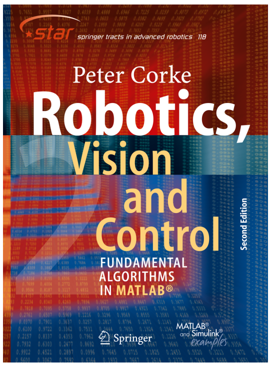
1.10 Contributing to the Toolboxes

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

1.11 Acknowledgements

I have corresponded with a great many people via email since the first release of this Toolbox. Some have identified bugs and shortcomings in the documentation, and even better, some have provided bug fixes and even new modules, thankyou. See the file `CONTRIB` for details.

I would especially like to thank the following. Giorgio Grisetti and Gian Diego Tipaldi for the core of the pose graph solver. Arturo Gil for allowing me to ship the STL robot models from ARTE. Jörn Malzahn has donated a considerable amount of code, his Robot Symbolic Toolbox for MATLAB. Bryan Moutrie has contributed parts of his open-source package phiWARE to RTB, the remainder of that package can be found online. Other special mentions to Gautam Sinha, Wynand Smart for models of industrial robot arm, Pauline Pounds for the quadrotor and related models, Paul Newman for inspiring the mobile robot code, and Giorgio Grissetti for inspiring the pose graph code.


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

Chapter 2

Functions and classes

Astar

s

`A* navigation` class

A concrete subclass of the Navigation class that implements the `A* navigation` algorithm. Methods included are for the standard case, multiobjective optimization (MOO) – i.e. optimizes over several objectives/criteria – and the `A*`-PO algorithms for MOO that utilizes Pareto optimality.

Methods:

plan	Compute the cost map given a goal and map
path	Compute a path to the goal
visualize	Display the obstacle map (deprecated)
plot	Display the obstacle map

costmap_modify Modify the costmap

costmap_get	Return the current costmap
costmap_set	Set the current costmap
display	Print the parameters in human readable form
char	Convert to string

Properties: TBD

Example 1

```
load map1                % load map
goal = [50;30];
```



```
start=[20;10];
as = Astar(map);          % create Navigation object
as.plan(goal,2,3,0);      % setup costmap for specified goal;

                          % standard D* algorithm w/ 2 objectives
                          % and 3 costmap layers

as.path(start);           % plan solution path start-to-goal, animate
P = as.path(start);       % plan solution path start-to-goal, return

                          % path
```

Example 2

```
goal = [100;100];
start = [1;1];
as = Astar(0);             % create Navigation object with pseudo-

                           % random occupancy grid

ds.addCost(terrain);       % terrain is a 100x100 matrix of

                           % elevations [0,1]
```

ds.plan(goal,3,4,0); % setup costmap for specified goal

```
                           % (3 and 4 include the added terrain cost)

as.path(start);            % plan solution path start-goal, animate
P = as.path(start);        % plan solution path start-goal, return

                           % path
```

Notes

- Obstacles are represented by Inf in the costmap.

References

- A Pareto Optimal D* Search Algorithm for Multiobjective Path Planning, A. Lavin.
- A Pareto Front-Based Multiobjective Path Planning Algorithm, A. Lavin.
- Robotics, Vision & Control, Sec 5.2.2, Peter Corke, Springer, 2011.

See Also Navigation, Dstar

Astar.Astar

A* constructor

AS = Astar(MAP, OPTIONS) is a A* navigation object, and MAP is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied). The occupancy grid is coverted to a costmap with a unit cost for traversing a cell.

Options

'world'= 0	will call for a pseudo-random occupancy grid
'goal',G	Specify the goal point (2×1)
'metric',M (default) or 'cityblock'	Specify the distance metric as 'Euclidean'
'inflate',K	Inflate all obstacles by K cells
'quiet'	Don't display the progress spinner

Other options are supported by the Navigation superclass.

See also

Navigation.Navigation

Astar.addCost

Add an additional cost layer

`AS.addCost(values)` adds the matrix specified by `values` as a cost layer. Inputs

`values`: normalized matrix the size of the environment

Astar.char

Convert Navigation object to string

`AS.char()` is a string representing the state of the **Astar** object in human-readable form.

See also

Astar.display, Navigation.char

Astar.cost_get

Get the specified cost layer

Astar.costmap_get

Get the current costmap

`C = AS.costmap_get()` is the current costmap. The value of each element represents the cost of traversing the cell. It is autogenerated by the class constructor from the occupancy grid such that:

- free cell (occupancy 0) has a cost of 1
- occupied cell (occupancy >0) has a cost of Inf

See also

Astar.costmap_set, Astar.costmap_modify

Astar.costmap_modify

Modify cost map

`AS.costmap_modify(P, NEW)` modifies the cost map at `P`=[X,Y] to have the value `NEW`. If `P` ($2 \times M$) and `NEW` ($1 \times M$) then the cost of the points defined by the columns of `P` are set to the corresponding elements of `NEW`.

Notes

- After one or more point costs have been updated the path should be replanned by calling AS.plan().

See also

Astar.costmap_set, Astar.costmap_get

Astar.costmap_set

Set the current costmap

`AS.costmap_set(C)` sets the current costmap. This method accepts the full costmap – i.e. all layers.

Notes:

- After the cost map is changed the path should be replanned by calling AS.plan().

See also

Astar.costmap_get, Astar.costmap_modify

Astar.dc

the distance cost of moving from state X to state Y

Astar.goal_change

Changes the costlayers due to new goal

position

Astar.heurstic_get

Get the current heuristic map

`C = AS.heuristice_get()` is the current heuristic layer. It is computed in Astar.plan.

See also

Astar.plan

Astar.INSERT

state X to the openlist with objective space values

specified by pt.

Astar.neighbors

indices of neighbor states (max 8) as a row vector

Astar.next

by Navigation.step

Backpropagate from goal to start Return [col;row] of previous step

---

Astar.path

Find a path between two points

`AS.path(START)` finds and displays a path from `START` to GOAL which is overlaid on the occupancy grid.

`P = AS.path(START)` returns the path ($2 \times M$) from `START` to GOAL.

Astar.plan

Prep the grid for planning.

`AS.plan()` updates AS with a costmap of distance to the goal from every non-obstacle point in the map. The goal is as specified to the constructor.

Inputs:

goal: goal state coordinates N: number of optimization objectives; standard A* is 2 (i.e. distance and heuristic) layers: number of cost layers in costmap algorithm: specify standard A*(0), A*-MOO (1), A*-PO (2)

Astar.plot

Visualize navigation environment

`AS.plot()` displays the occupancy grid and the goal distance in a new figure. The goal distance is shown by intensity which increases with distance from the goal. Obstacles are overlaid and shown in red.

`AS.plot(P)` as above but also overlays a path given by the set of points `P` ($M \times 2$).

See also

Navigation.plot

Astar.projectCost

the projection of state a into objective space. If

specified, location is moving from b to a (case 3).

Astar.reset

Reset the planner

`AS.reset()` resets the A* planner. The next instantiation of AS.plan() will perform a global replan.

Astar.updateCosts

Only for costs that accumulate (i.e. sum) over the

path, and for dynamic costs. E.g. the heuristic parameter only needs updating when the goal state changes; its values are stored for each cell.

Location moving from state b to a.

The costs are coded to be (1) distance, (2) heuristic, (3) elevation, (4) solar deviation, and (5) risk. If deviating from these costs (in this order) you MUST EDIT THIS METHOD.

Astar.vc

the robot unit vector – direction of moving from

state X to state Y

AstarMOO

A*-MOO navigation class

A concrete subclass of the Navigation class that implements the A* navigation algorithm for multiobjective optimization (MOO) - i.e. optimizes over several objectives/criteria.

Methods:

plan	Compute the cost map given a goal and map
path	Compute a path to the goal
visualize	Display the obstacle map (deprecated)
plot	Display the obstacle map
costmap_modify	Modify the costmap
costmap_get	Return the current costmap
costmap_set	Set the current costmap
distancemap_get	Set the current distance map
heuristic_get	Get the current heuristic map
display	Print the parameters in human readable form
char	Convert to string

Properties: TBD

Example

```
load map1             % load map
goal = [50;30];
start = [20;10];
as = AstarMOO(map);      % create Navigation object
as.plan(goal,2);         % setup costmap for specified goal
as.path(start);          % plan solution path star-goal, animate
P = as.path(start);      % plan solution path star-goal, return path
```

Example 2:

```
goal = [100;100];
start = [1;1];
as = AstarMOO(0);     % create Navigation object with random occupancy grid
as.addCost(1,L);      % add 1st add'l cost layer L
as.plan(goal,3);      % setup costmap for specified goal
as.path(start);       % plan solution path start-goal, animate
P = as.path(start);      % plan solution path start-goal, return path
```

Notes

- Obstacles are represented by Inf in the costmap.

References

- A Pareto Optimal D* Search Algorithm for Multiobjective Path Planning, A. Lavin.
- A Pareto Front-Based Multiobjective Path Planning Algorithm, A. Lavin.
- Robotics, Vision & Control, Sec 5.2.2, Peter Corke, Springer, 2011.

Author

Alexander Lavin

See also

Navigation, Astar, AstarPO

AstarMOO.AstarMOO

A*-MOO constructor

`AS = AstarMOO(MAP, OPTIONS)` is a A* navigation object, and `MAP` is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied). The occupancy grid is coverted to a costmap with a unit cost for traversing a cell.

Options

'goal',G	Specify the goal point (2×1)
'metric',M or 'cityblock'.	Specify the distance metric as 'euclidean'(default)
'inflate',K	Inflate all obstacles by K cells.
'quiet'	Don't display the progress spinner

Other options are supported by the Navigation superclass.

Notes

- If `MAP` == 0 a random map is created.

See also

Navigation.Navigation

AstarMOO.addCost

Add an additional cost layer

`AS.addCost(LAYER, VALUES)` adds the matrix specified by values as a cost layer. The layer number is given by `LAYER`, and `VALUES` has the same size as the original occupancy grid.

AstarMOO.char

Convert navigation object to string

`AS.char()` is a string representing the state of the Astar object in human-readable form.

See also

AstarMOO.display, Navigation.char

AstarMOO.cost_get

Get the specified cost layer

AstarMOO.costmap_get

Get the current costmap

`C = AS.costmap_get()` is the current costmap. The cost map is the same size as the occupancy grid and the value of each element represents the cost of traversing the cell. It is autogenerated by the class constructor from the occupancy grid such that:

- free cell (occupancy 0) has a cost of 1
- occupied cell (occupancy $>$0) has a cost of Inf

See also

Astar.costmap_set, Astar.costmap_modify

AstarMOO.costmap_modify

Modify cost map

`AS.costmap_modify(P, NEW)` modifies the cost map at `P`=[X,Y] to have the value `NEW`. If `P` $(2 \times M)$ and `NEW` $(1 \times M)$ then the cost of the points defined by the columns of `P` are set to the corresponding elements of `NEW`.

Notes

- After one or more point costs have been updated the path should be replanned by calling AS.plan().

See also

AstarMOO.costmap_set, AstarMOO.costmap_get

AstarMOO.costmap_set

Set the current costmap

`AS.costmap_set(C)` sets the current costmap. The cost map is the same size as the occupancy grid and the value of each element represents the cost of traversing the cell. A high value indicates that the cell is more costly (difficult) to traverese. A value of Inf indicates an obstacle.

Notes

- After the cost map is changed the path should be replanned by calling AS.plan().

See also

Astar.costmap_get, Astar.costmap_modify

AstarMOO.heuristic_get

Get the current heuristic map

`C = AS.heuristic_get()` is the current heuristic map. This map is the same size as the occupancy grid and the value of each element is the shortest distance from the corresponding point in the map to the current goal. It is computed by Astar.plan.

See also

Astar.plan

AstarMOO.next

from goal to start

Return [col;row] of previous step

AstarMOO.path

Find a path between two points

`AS.path(START)` finds and displays a path from `START` to GOAL which is overlaid on the occupancy grid.

`P = AS.path(START)` returns the path ($2 \times M$) from `START` to GOAL.

AstarMOO.plan

Prep the grid for planning.

`AS.plan()` updates AS with a costmap of distance to the goal from every non-obstacle point in the map. The goal is as specified to the constructor.

Inputs:

goal: goal state coordinates N: number of optimization objectives; standard A* is 2 (i.e. distance and heuristic)

AstarMOO.plot

Visualize navigation environment

`AS.plot()` displays the occupancy grid and the goal distance in a new figure. The goal distance is shown by intensity which increases with distance from the goal. Obstacles are overlaid and shown in red.

`AS.plot(P)` as above but also overlays a path given by the set of points `P` ($M \times 2$).

See also

Navigation.plot

AstarMOO.reset

Reset the planner

`AS.reset()` resets the A* planner. The next instantiation of AS.plan() will perform a global replan.

AstarPO

(A*-PO)

`A*PO` navigation class

`A` concrete subclass of the Navigation class that implements the `A*` navigation algorithm for multiobjective optimization (MOO) - i.e. optimizes over several objectives/criteria.

Methods

plan	Compute the cost map given a goal and map
path	Compute a path to the goal
visualize	Display the obstacle map (deprecated)
plot	Display the obstacle map

costmap_modify Modify the costmap

costmap_get	Return the current costmap
costmap_set	Set the current costmap
distancemap_get	Set the current distance map
heuristic_get	Get the current heuristic map
display	Print the parameters in human readable form
char	Convert to string

Properties

TBD

Example

```
 load map1          % load map

goal = [50;30];
start = [20;10];
as = AstarPO(map); % create Navigation object
as.plan(goal,2);   % setup costmap for specified goal
as.path(start);    % plan solution path star-goal, animate
P = as.path(start); % plan solution path star-goal, return path
```

Example 2:

```
goal = [100;100];
start = [1;1];
as = AstarPO(0);    % create Navigation object with random occupancy grid
as.addCost(1,L);    % add 1st add'l cost layer L
as.plan(goal,3);    % setup costmap for specified goal
as.path(start);     % plan solution path start-goal, animate
P = as.path(start); % plan solution path start-goal, return path
```

Notes

- Obstacles are represented by Inf in the costmap.

References

- A Pareto Optimal D* Search Algorithm for Multiobjective Path Planning, A. Lavin.
- A Pareto Front-Based Multiobjective Path Planning Algorithm, A. Lavin.
- Robotics, Vision & Control, Sec 5.2.2, Peter Corke, Springer, 2011.

Author

Alexander Lavin

See also

Navigation, Astar, AstarMOO

AstarPO.AstarPO

A*-PO constructor

`AS = AstarPO(MAP, OPTIONS)` is a A* navigation object, and `MAP` is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied). The occupancy grid is coverted to a costmap with a unit cost for traversing a cell.

Options

'world'= 0	will call for a random occupancy grid to be built
'goal',G	Specify the goal point (2×1)
'metric',M or 'cityblock'.	Specify the distance metric as 'euclidean'(default)
'inflate',K	Inflate all obstacles by K cells.
'quiet'	Don't display the progress spinner

Other options are supported by the Navigation superclass.

See also

Navigation.Navigation

AstarPO.addCost

Add an additional cost layer

`AS.addCost(LAYER, VALUES)` adds the matrix specified by values as a cost layer. The layer number is given by `LAYER`, and `VALUES` has the same size as the original occupancy grid.

AstarPO.char

Convert navigation object to string

`AS.char()` is a string representing the state of the Astar object in human-readable form.

See also

AstarMOO.display, Navigation.char

AstarPO.cost_get

Get the specified cost layer

AstarPO.costmap_get

Get the current costmap

`C = AS.costmap_get()` is the current costmap. The cost map is the same size as the occupancy grid and the value of each element represents the cost of traversing the cell. It is autogenerated by the class constructor from the occupancy grid such that:

- free cell (occupancy 0) has a cost of 1
- occupied cell (occupancy >0) has a cost of Inf

See also

Astar.costmap_set, Astar.costmap_modify

AstarPO.costmap_modify

Modify cost map

`AS.costmap_modify(P, NEW)` modifies the cost map at `P`=[X,Y] to have the value `NEW`. If `P` $(2 \times M)$ and `NEW` $(1 \times M)$ then the cost of the points defined by the columns of `P` are set to the corresponding elements of `NEW`.

Notes

- After one or more point costs have been updated the path should be replanned by calling AS.plan().

See also

AstarMOO.costmap_set, AstarMOO.costmap_get

AstarPO.costmap_set

Set the current costmap

`AS.costmap_set(C)` sets the current costmap. The cost map is the same size as the occupancy grid and the value of each element represents the cost of traversing the cell. A high value indicates that the cell is more costly (difficult) to traverese. A value of Inf indicates an obstacle.

Notes:

- After the cost map is changed the path should be replanned by calling AS.plan().

See also

Astar.costmap_get, Astar.costmap_modify

AstarPO.heurstic_get

Get the current heuristic map

`C = AS.heuristice_get()` is the current heuristic map. This map is the same size as the occupancy grid and the value of each element is the shortest distance from the corresponding point in the map to the current goal. It is computed by Astar.plan.

See also

Astar.plan

AstarPO.next

from goal to start

Return [col;row] of previous step

AstarPO.path

Find a path between two points

`AS.path(START)` finds and displays a path from `START` to GOAL which is overlaid on the occupancy grid.

`P = AS.path(START)` returns the path ($2 \times M$) from `START` to GOAL.

AstarPO.plan

Prep the grid for planning.

`AS.plan()` updates AS with a costmap of distance to the goal from every non-obstacle point in the map. The goal is as specified to the constructor.

Inputs:

goal: goal state coordinates N: number of optimization objectives; standard A* is 2 (i.e. distance and heuristic)

AstarPO.plot

Visualize navigation environment

`AS.plot()` displays the occupancy grid and the goal distance in a new figure. The goal distance is shown by intensity which increases with distance from the goal. Obstacles are overlaid and shown in red.

`AS.plot(P)` as above but also overlays a path given by the set of points `P` ($M \times 2$).

See also

Navigation.plot

AstarPO.reset

Reset the planner

`AS.reset()` resets the A* planner. The next instantiation of AS.plan() will perform a global replan.

Bicycle

Car-like vehicle class

This concrete class models the kinematics of a car-like vehicle (bicycle or Ackerman model) on a plane. For given steering and velocity inputs it updates the true vehicle state and returns noise-corrupted odometry readings.

Methods

Bicycle	constructor
add_driver	attach a driver object to this vehicle
control	generate the control inputs for the vehicle
deriv	derivative of state given inputs
init	initialize vehicle state
f	predict next state based on odometry
Fx	Jacobian of f wrt x
Fv	Jacobian of f wrt odometry noise
update	update the vehicle state
run	run for multiple time steps
step	move one time step and return noisy odometry

Plotting/display methods

char	convert to string
display	display state/parameters in human readable form
plot	plot/animate vehicle on current figure
plot_xy	plot the true path of the vehicle
Vehicle.plotv	plot/animate a pose on current figure

Properties (read/write)

x	true vehicle state: x, y, theta (3×1)
V	odometry covariance (2×2)
odometry	distance moved in the last interval (2×1)

rdim	dimension of the robot (for drawing)

L	length of the vehicle (wheelbase)
alphalim	steering wheel limit
maxspeed	maximum vehicle speed
T	sample interval

verbose	verbosity
x_hist	history of true vehicle state ($N \times 3$)
driver	reference to the driver object
x0	initial state, restored on init()

Examples

Odometry covariance (per timstep) is

V = diag([0.02, 0.5*π/180].^{2});

Create a vehicle with this noisy odometry

v = Bicycle('covar', diag([0.1 0.01].^{2});

and display its initial state

v

now apply a speed (0.2m/s) and steer angle (0.1rad) for 1 time step

`odo = v.step`(0.2, 0.1)

where `odo` is the noisy odometry estimate, and the new true vehicle state

`v`

We can add a driver object

`v.add`_driver(RandomPath(10))

which will move the vehicle within the region -10<x<10, -10<y<10 which we can see by

`v.run`(1000)

which shows an animation of the vehicle moving for 1000 time steps between randomly selected wayoints.

Notes

- Subclasses the MATLAB handle class which means that pass by reference semantics apply.

Reference

Robotics, Vision & Control, Chap 6 Peter Corke, Springer 2011

See also

RandomPath, EKF

Bicycle.Bicycle

Vehicle object constructor

`V = Bicycle(OPTIONS)` creates a **Bicycle** object with the kinematics of a bicycle (or Ackerman) vehicle.

Options

'steermax',M	Maximu steer angle [rad] (default 0.5)
'accelmax',M	Maximum acceleration [m/s2] (default Inf)
'covar',C	specify odometry covariance (2×2) (default 0)
'speedmax',S	Maximum speed (default 1m/s)
'L',L	Wheel base (default 1m)
'x0',x0	Initial state (default (0,0,0))
'dt',T	Time interval (default 0.1)
'rdim',R	Robot size as fraction of plot window (default 0.2)
'verbose'	Be verbose

Notes

- The covariance is used by a "hidden" random number generator within the class.
- Subclasses the MATLAB handle class which means that pass by reference semantics apply.

Notes

- Subclasses the MATLAB handle class which means that pass by reference semantics apply.

Bicycle.char

Convert to a string

`s = V.char()` is a string showing vehicle parameters and state in a compact human readable format.

See also

Bicycle.display

Bicycle.deriv

Time derivative of state

`DX = V.deriv(T, X, U)` is the time derivative of state (3×1) at the state `X` (3×1) with input `U` (2×1).

Notes

- The parameter `T` is ignored but called from a continuous time integrator such as ode45 or Simulink.

Bicycle.f

Predict next state based on odometry

`XN = V.f(X, ODO)` is the predicted next state `XN` (1×3) based on current state `X` (1×3) and odometry `ODO` (1×2) = [distance, heading_change].

`XN = V.f(X, ODO, W)` as above but with odometry noise `W`.

Notes

- Supports vectorized operation where `X` and `XN` ($N \times 3$).

Bicycle.Fv

Jacobian df/dv

`J = V.Fv(X, ODO)` is the Jacobian df/dv (3×2) at the state `X`, for odometry input `ODO` (1×2) = [distance, heading_change].

See also

Bicycle.F, Vehicle.Fx

Bicycle.Fx

Jacobian df/dx

`J = V.Fx(X, ODO)` is the Jacobian df/dx (3×3) at the state `X`, for odometry input `ODO` (1×2) = [distance, heading_change].

See also

Bicycle.f, Vehicle.Fv

Bicycle.update

Update the vehicle state

`ODO = V.update(U)` is the true odometry value for motion with `U`=[speed,steer].

Notes

- Appends new state to state history property x_hist.
- Odometry is also saved as property odometry.

Bug2

Bug navigation class

A concrete subclass of the abstract Navigation class that implements the bug2 navigation algorithm. This is a simple automaton that performs local planning, that is, it can only sense the immediate presence of an obstacle.

Methods

Bug2	Constructor
query	Find a path from start to goal
plot	Display the obstacle map
display	Display state/parameters in human readable form
char	Convert to string

Example

```
load map1              % load the map
bug = Bug2(map);       % create navigation object
start = [20,10];
goal = [50,35];
bug.query(start, goal);    % animate path
```

Reference

- Dynamic path planning for a mobile automaton with limited information on the environment,, V. Lumelsky and A. Stepanov,
- IEEE Transactions on Automatic Control, vol. 31, pp. 1058-1063, Nov. 1986.
- Robotics, Vision & Control, Sec 5.1.2, Peter Corke, Springer, 2011.

See also

Navigation, DXform, Dstar, PRM

Bug2.Bug2

Construct a Bug2 navigation object

`B = Bug2(MAP, OPTIONS)` is a bug2 navigation object, and `MAP` is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied).

Options

'goal',G	Specify the goal point (1×2)
'inflate',K	Inflate all obstacles by K cells.

See also

Navigation.Navigation

Bug2.query

Find a path

`B.query(START, GOAL, OPTIONS)` is the path ($N \times 2$) from `START` (1×2) to `GOAL` (1×2). Row are the coordinates of successive points along the path. If either `START` or `GOAL` is [] the grid map is displayed and the user is prompted to select a point by clicking on the plot.

Options

'animate'	show a simulation of the robot moving along the path
'movie',M	create a movie
'current'	show the current position position as a black circle

Notes

- `START` and `GOAL` are given as X,Y coordinates in the grid map, not as MATLAB row and column coordinates.
- `START` and `GOAL` are tested to ensure they lie in free space.
- The Bug2 algorithm is completely reactive so there is no planning method.
- If the bug does a lot of back tracking it's hard to see the current position, use the 'current'option.
- For the movie option if M contains an extension a movie file with that extension is created. Otherwise a folder will be created containing
- individual frames.

See also

Animate

ccodefunctionstring

Converts a symbolic expression into a C-code function

`[FUNSTR, HDRSTR] = ccodefunctionstring(SYMEXPR, ARGLIST)` returns a string representing a C-code implementation of a symbolic expression `SYMEXPR`. The C-code implementation has a signature of the form:

```
void funname(double[][n_o] out, const double in1,
  const double* in2, const double[][n_i] in3);
```

depending on the number of inputs to the function as well as the dimensionality of the inputs (n_i) and the output (n_o). The whole C-code implementation is returned in `FUNSTR`, while `HDRSTR` contains just the signature ending with a semi-colon (for the use in header files).

Options

'funname',name	this optional argument is omitted, the variable name	Specify the name of the of the first input argume
'output',outVar		Defines the identifier of
'vars',varCells	elements of this cell array contain the symbolic variables required to	The inputs to the C-code compute the output. The symbolic variables. The as exemplified above.
'flag',sig		Specifies if function sign

Example

```
% Create symbolic variables
syms q1 q2 q3

Q = [q1 q2 q3];
% Create symbolic expression
myrot = rotz(q3)*roty(q2)*rotx(q1)

% Generate C-function string
[funstr, hdrstr] = ccodefunctionstring(myrot,'output','foo', ...
'vars',{Q},'funname','rotate_xyz')
```

Notes

- The function wraps around the built-in Matlab function 'ccode'. It does not check for proper C syntax. You must take care of proper
- dimensionality of inputs and outputs with respect to your symbolic
- expression on your own. Otherwise the generated C-function may not
- compile as desired.

Author

Joern Malzahn, (joern.malzahn@tu-dortmund.de)

See also

ccode, matlabFunction

---

chi2inv_rtb

Inverse chi-squared function

`X = CHI2INV_RTB(P, N)` is the inverse chi-squared CDF function of `N`-degrees of freedom.

Notes

- only works for `N`=2
- uses a table lookup with around 6 figure accuracy
- an approximation to chi2inv() from the Statistics & Machine Learning Toolbox

See also

chi2inv

---

ctraj

Cartesian trajectory between two poses

`TC = CTRAJ(T0, T1, N)` is a Cartesian trajectory ($4 \times 4 \times$ `N`) from pose `T0` to `T1` with `N` points that follow a trapezoidal velocity profile along the path. The Cartesian trajectory is a homogeneous transform sequence and the last subscript being the point index, that is, T(:,:,i) is the i'th point along the path.

`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

- If T0 or T1 is equal to [] it is taken to be the identity matrix.
- 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, UnitQuaternion.interp, SE3.ctraj

---

delta2tr

Convert differential motion to a homogeneous transform

T = DELTA2TR(D) is a homogeneous transform (4×4) representing differential translation and rotation. The vector D=(dx, dy, dz, dRx, dRy, dRz) represents an infinitessimal motion, and is an approximation to the spatial velocity multiplied by time.

See also

tr2delta, SE3.delta

DHFactor

Simplify symbolic link transform expressions

F = DHFactor(S) is an object that encodes the kinematic model of a robot provided by a string S that represents a chain of elementary transforms from the robot's base to its tool tip. The chain of elementary rotations and translations is symbolically factored into a sequence of link transforms described by DH parameters.

For example:

s = 'Rz(q1).Rx(q2).Ty(L1).Rx(q3).Tz(L2)';

indicates a rotation of q1 about the z-axis, then rotation of q2 about the x-axis, translation of L1 about the y-axis, rotation of q3 about the x-axis and translation of L2 along the z-axis.

Methods

base	the base transform as a Java string
tool	the tool transform as a Java string
command representing the specified kinematics	a command string that will create a SerialLink() object
char	convert to string representation
display	display in human readable form

Example

>> s = 'Rz(q1).Rx(q2).Ty(L1).Rx(q3).Tz(L2)'; >> dh = DHFactor(s); >> dh DH(q1+90, 0, 0, +90).DH(q2, L1, 0, 0).DH(q3-90, L2, 0, 0).Rz(+90).Rx(-90).Rz(-90) >> r = eval(dh.command('myrobot'));

Notes

- Variables starting with q are assumed to be joint coordinates.
- Variables starting with L are length constants.
- Length constants must be defined in the workspace before executing the last line above.
- Implemented in Java.
- Not all sequences can be converted to DH format, if conversion cannot be achieved an error is reported.

Reference

- A simple and systematic approach to assigning Denavit-Hartenberg parameters, P.Corke, IEEE Transaction on Robotics, vol. 23, pp. 590-594, June 2007.
- Robotics, Vision & Control, Sec 7.5.2, 7.7.1, Peter Corke, Springer 2011.

See also

SerialLink

distancexform

Distance transform

D = DISTANCEXFORM(IM, OPTIONS) is the distance transform of the binary image IM. The elements of D have a value equal to the shortest distance from that element to a non-zero pixel in the input image IM.

D = DISTANCEXFORM(OCCGRID, GOAL, OPTIONS) is the distance transform of the occupancy grid OCCGRID with respect to the specified goal point GOAL = [X,Y]. The cells of the grid have values of 0 for free space and 1 for obstacle. The resulting matrix D has cells whose value is the shortest distance to the goal from that cell, or NaN if the cell corresponds to an obstacle (set to 1 in OCCGRID).

Options:

'euclidean'	Use Euclidean (L2) distance metric (default)
'cityblock'	Use cityblock or Manhattan (L1) distance metric
'animate'	Show the iterations of the computation
'delay',D	Delay of D seconds between animation frames (default 0.2s)
'movie',M	Save animation to a movie file or folder
'noipt'	Don't use Image Processing Toolbox, even if available
'novlfeat'	Don't use VLFeat, even if available
'nofast'	Don't use IPT, VLFeat or imorph, even if available.

'delay'

Notes

- For the first case Image Processing Toolbox (IPT) or VLFeat will be used if available, searched for in that order. They use a 2-pass rather than
- iterative algorithm and are much faster.
- Options can be used to disable use of IPT or VLFeat.
- If IPT or VLFeat are not available, or disabled, then imorph is used.
- If IPT, VLFeat or imorph are not available a slower M-function is used.
- If the 'animate'option is given then the MATLAB implementation is used.
- Using imorph requires iteration and is slow.
  - For the second case the Machine Vision Toolbox function imorph is required.
  - imorph is a mex file and must be compiled.
- The goal is given as [X,Y] not MATLAB [row,col] format.

See also

imorph, DXform, Animate

distributeblocks

Distribute blocks in Simulink block library

`distributeBlocks(MODEL)` equidistantly distributes blocks in a Simulink block library named `MODEL`.

Notes

- The MATLAB functions to create Simulink blocks from symbolic expresssions actually place all blocks on top of each other. This
- function scans a simulink model and rearranges the blocks on an
- equidistantly spaced grid.
- The Simulink model must already be opened before running this function!

Author

Joern Malzahn, (joern.malzahn@tu-dortmund.de)

See also

symexpr2slblock, doesblockexist

doesblockexist

Check existence of block in Simulink model

`RES = doesblockexist(MDLNAME, BLOCKADDRESS)` is a logical result that indicates whether or not the block `BLOCKADDRESS` exists within the Simulink model `MDLNAME`.

Author

Joern Malzahn, (joern.malzahn@tu-dortmund.de)

See also

symexpr2slblock, distributeblocks

Dstar

D* navigation class

A concrete subclass of the abstract Navigation class that implements the D* navigation algorithm. This provides minimum distance paths and facilitates incremental replanning.

Methods

Dstar	Constructor
plan	Compute the cost map given a goal and map
query	Find a path
plot	Display the obstacle map
display	Print the parameters in human readable form
char	Convert to string% costmap_modify Modify the costmap
modify_cost	Modify the costmap

Properties (read only)

distancemap	Distance from each point to the goal.
costmap	Cost of traversing cell (in any direction).
niter	Number of iterations.

Example

```
load map1               % load map
goal = [50,30];
start=[20,10];
ds = Dstar(map);        % create navigation object
ds.plan(goal)           % create plan for specified goal
ds.query(start)          % animate path from this start location
```


Notes

- Obstacles are represented by Inf in the costmap.
- The value of each element in the costmap is the shortest distance from the corresponding point in the map to the current goal.

References

- The D* algorithm for real-time planning of optimal traverses, A. Stentz,
- Tech. Rep. CMU-RI-TR-94-37, The Robotics Institute, Carnegie-Mellon University, 1994.
- https://www.ri.cmu.edu/pub_files/pub3/stentz_anthony__tony__1994_2/stentz_anthony__tony__1994_2.pc
- Robotics, Vision & Control, Sec 5.2.2, Peter Corke, Springer, 2011.

See also

Navigation, DXform, PRM

Dstar.Dstar

D* constructor

`DS = Dstar(MAP, OPTIONS)` is a D* navigation object, and `MAP` is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied). The occupancy grid is coverted to a costmap with a unit cost for traversing a cell.

Options

'goal',G	Specify the goal point (2×1)
'metric',M or 'cityblock'.	Specify the distance metric as 'euclidean'(default)
'inflate',K	Inflate all obstacles by K cells.
'progress'	Don't display the progress spinner

Other options are supported by the Navigation superclass.

See also

Navigation.Navigation

Dstar.char

Convert navigation object to string

`DS.char()` is a string representing the state of the **Dstar** object in human-readable form.

See also

Dstar.display, Navigation.char

Dstar.modify_cost

Modify cost map

`DS.modify_cost(P, C)` modifies the cost map for the points described by the columns of `P` $(2 \times N)$ and sets them to the corresponding elements of `C` $(1 \times N)$. For the particular case where `P` (2×2) the first and last columns define the corners of a rectangular region which is set to `C` (1×1).

Notes

- After one or more point costs have been updated the path should be replanned by calling DS.plan().

See also

Dstar.set_cost

Dstar.plan

Plan path to goal

`DS.plan(OPTIONS)` create a D* plan to reach the goal from all free cells in the map. Also updates a D* plan after changes to the costmap. The goal is as previously specified.

`DS.plan(GOAL,OPTIONS)` as above but goal given explicitly.

Options

'animate'	Plot the distance transform as it evolves
'progress'	Display a progress bar

Note

- If a path has already been planned, but the costmap was modified, then reinvoking this method will replan,
- incrementally updating the plan at lower cost than a full
- replan.
- The reset method causes a fresh plan, rather than replan.

See also

Dstar.reset

Dstar.plot

Visualize navigation environment

`DS.plot()` displays the occupancy grid and the goal distance in a new figure. The goal distance is shown by intensity which increases with distance from the goal. Obstacles are overlaid and shown in red.

`DS.plot(P)` as above but also overlays a path given by the set of points `P` ($M \times 2$).

See also

Navigation.plot

Dstar.reset

Reset the planner

`DS.reset()` resets the D* planner. The next instantiation of DS.plan() will perform a global replan.

Dstar.set_cost

Set the current costmap

`DS.set_cost(C)` sets the current costmap. The cost map is the same size as the occupancy grid and the value of each element represents the cost of traversing the cell. A high value indicates that the cell is more costly (difficult) to traverese. A value of Inf indicates an obstacle.

Notes

- After the cost map is changed the path should be replanned by calling DS.plan().

See also

Dstar.modify_cost

DstarMOO

D*-MOO navigation class

A concrete subclass of the Navigation class that implements the D* navigation algorithm; facilitates incremental replanning. This implementation of D* is intended for multiobjective optimization (MOO) problems - i.e. optimizes over several objectives/criteria.

Methods

plan	Compute the cost map given a goal and map
path	Compute a path to the goal
visualize	Display the obstacle map (deprecated)
plot	Display the obstacle map
cost_get	Return the specified cost layer
costmap_modify	Modify the costmap
modify_cost	Modify the costmap (deprecated, use costmap_modify)
costmap_get	Return the current costmap
costmap_set	Set the current costmap
distancemap_get	Set the current distance map
display	Print the parameters in human readable form
char	Convert to string

Properties

TBD

Example

```
load map1             % load map
goal = [50,30];
start=[20,10];
ds = DstarMOO(map);   % create navigation object
ds.plan(goal,1)       % create plan for specified goal
ds.path(start)        % animate path from this start location
```

Example 2:

goal = [100;100]; start = [1;1];

ds = DstarMOO(0);	% create Navigation object with random occupancy grid
ds.addCost(1,L);	% add 1st add'l cost layer L
ds.plan(goal,2);	% setup costmap for specified goal
ds.path(start);	% plan solution path start-goal, animate
P = as.path(start);	% plan solution path start-goal, return path

Notes

- Obstacles are represented by Inf in the costmap.

References

- The D* algorithm for real-time planning of optimal traverses, A. Stentz, Tech. Rep. CMU-RI-TR-94-37, The Robotics Institute,
- Carnegie-Mellon University, 1994.
- A Pareto Optimal D* Search Algorithm for Multiobjective Path Planning, A. Lavin.
- Robotics, Vision & Control, Sec 5.2.2, Peter Corke, Springer, 2011.

Author

Alexander Lavin based on Dstar by Peter Corke

See also

Navigation, Dstar, DstarPO, Astar, DXform

DstarMOO.DstarMOO

D*MOO constructor

`DS = DstarMOO(MAP, OPTIONS)` is a D* navigation object, and `MAP` is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied). The occupancy grid is coverted to a costmap with a unit cost for traversing a cell.

Options

'goal',G	Specify the goal point (2×1)
'metric',M or 'cityblock'.	Specify the distance metric as 'euclidean'(default)
'inflate',K	Inflate all obstacles by K cells.
'quiet'	Don't display the progress spinner

Other options are supported by the Navigation superclass.

Notes

- If `MAP` == 0 a random map is created.

See also

Navigation.Navigation

DstarMOO.addCost

Add an additional cost layer

`DS.addCost(layer,values)` adds the matrix specified by `values` as a cost `layer`. Inputs

`layer`: 1, 2, or 3 to specify which cost `layer` to add `values`: normalized matrix the size of the environment (100×100)

DstarMOO.char

Convert navigation object to string

`DS.char()` is a string representing the state of the Dstar object in human-readable form.

See also

Dstar.display, Navigation.char

DstarMOO.cost_get

Get the specified cost layer

DstarMOO.costmap_get

Get the current costmap

`C = DS.costmap_get()` is the current costmap. The cost map is the same size as the occupancy grid and the value of each element represents the cost of traversing the cell. It is autogenerated by the class constructor from the occupancy grid such that:

- free cell (occupancy 0) has a cost of 1
- occupied cell (occupancy >0) has a cost of Inf

See also

Dstar.costmap_set, Dstar.costmap_modify

DstarMOO.costmap_modify

Modify cost map

`DS.costmap_modify(P, NEW)` modifies the cost map at `P`=[X,Y] to have the value `NEW`. If `P` ($2 \times M$) and `NEW` ($1 \times M$) then the cost of the points defined by the columns of `P` are set to the corresponding elements of `NEW`.

Notes

- After one or more point costs have been updated the path should be replanned by calling DS.plan().
- Replaces modify_cost, same syntax.

See also

Dstar.costmap_set, Dstar.costmap_get

DstarMOO.costmap_set

Set the current costmap

`DS.costmap_set(C)` sets the current costmap. The cost map is the same size as the occupancy grid and the value of each element represents the cost of traversing the cell. A high value indicates that the cell is more costly (difficult) to traverese. A value of Inf indicates an obstacle.

Notes

- After the cost map is changed the path should be replanned by calling DS.plan().

See also

Dstar.costmap_get, Dstar.costmap_modify

DstarMOO.distancemap_get

Get the current distance map

`C = DS.distancemap_get()` is the current distance map. This map is the same size as the occupancy grid and the value of each element is the shortest distance from the corresponding point in the map to the current goal. It is computed by Dstar.plan.

See also

Dstar.plan

DstarMOO.INSERT

state X to the openlist with objective space values

specified by pt.

DstarMOO.plan

Plan path to goal

`DS.plan()` updates DS with a costmap of distance to the goal from every non-obstacle point in the map. The goal is as specified to the constructor.

Note

- If a path has already been planned, but the costmap was modified, then reinvoking this method will replan,
- incrementally updating the plan at lower cost than a full
- replan.

Inputs:

goal: goal state coordinates N: number of optimization objectives; standard D* is 2 (i.e. distance and heuristic)

DstarMOO.plot

Visualize navigation environment

`DS.plot()` displays the occupancy grid and the goal distance in a new figure. The goal distance is shown by intensity which increases with distance from the goal. Obstacles are overlaid and shown in red.

`DS.plot(P)` as above but also overlays a path given by the set of points `P` ($M \times 2$).

See also

Navigation.plot

DstarMOO.PROCESS_STATE

with the lowest k value are removed from the

open list

DstarMOO.projectCost

the projection of state a into objective space. If

specified, location is moving from b to a.

DstarMOO.reset

Reset the planner

`DS.reset()` resets the D* planner. The next instantiation of DS.plan() will perform a global replan.

DstarMOO.updateCosts

Only for costs that accumulate (i.e. sum) over the

path, and for dynamic costs. E.g. the heuristic parameter DS.cost_h only needs updating when the goal state changes; it's values are stored for each cell.

Location moving from state b to a.

DstarPO

D*-PO navigation class

A concrete subclass of the Navigation class that implements the D* navigation algorithm; facilitates incremental replanning. This implementation of D* is intended for multiobjective optimization (MOO) problems - i.e. optimizes over several objectives/criteria - with the use of Pareto fronts (see Lavin paper).

Methods

plan	Compute the cost map given a goal and map
path	Compute a path to the goal
visualize	Display the obstacle map (deprecated)
plot	Display the obstacle map
cost_get	Return the specified cost layer
costmap_modify	Modify the costmap
modify_cost	Modify the costmap (deprecated, use costmap_modify)
costmap_get	Return the current costmap
costmap_set	Set the current costmap
distancemap_get	Set the current distance map
display	Print the parameters in human readable form
char	Convert to string

Properties

TBD

Example

```
load map1                % load map
goal = [50,30];
start=[20,10];
ds = DstarPO(map);       % create navigation object
ds.plan(goal,1)          % create plan for specified goal
ds.path(start)           % animate path from this start location
```

Example 2:

goal = [100;100]; start = [1;1];

ds = DstarPO(0);	% create Navigation object with random occupancy grid
ds.addCost(1,L);	% add 1st add'l cost layer L
ds.plan(goal,2);	% setup costmap for specified goal
ds.path(start);	% plan solution path start-goal, animate
P = as.path(start);	% plan solution path start-goal, return path

Notes

- Obstacles are represented by Inf in the costmap.

References

- The D* algorithm for real-time planning of optimal traverses, A. Stentz, Tech. Rep. CMU-RI-TR-94-37, The Robotics Institute,

- Carnegie-Mellon University, 1994.
- A Pareto Optimal D* Search Algorithm for Multiobjective Path Planning, A. Lavin.
- Robotics, Vision & Control, Sec 5.2.2, Peter Corke, Springer, 2011.

Author

Alexander Lavin based on Dstar by Peter Corke

See also

Navigation, Dstar, DstarMOO, Astar, DXform

DstarPO.DstarPO

D*-PO constructor

`DS = Dstar(MAP, OPTIONS)` is a D* navigation object, and `MAP` is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied). The occupancy grid is coverted to a costmap with a unit cost for traversing a cell.

Options

'goal',G	Specify the goal point (2×1)
'metric',M or 'cityblock'.	Specify the distance metric as 'euclidean'(default)
'inflate',K	Inflate all obstacles by K cells.
'quiet'	Don't display the progress spinner

Other options are supported by the Navigation superclass.

Notes

- If `MAP` == 0 a random map is created.

See also

Navigation.Navigation

DstarPO.addCost

Add an additional cost layer

`DS.addCost(layer,values)` adds the matrix specified by `values` as a cost `layer`. Inputs

`layer`: 1, 2, or 3 to specify which cost `layer` to add `values`: normalized matrix the size of the environment (100×100)

DstarPO.char

Convert navigation object to string

`DS.char()` is a string representing the state of the Dstar object in human-readable form.

See also

Dstar.display, Navigation.char

DstarPO.cost_get

Get the specified cost layer

DstarPO.costmap_get

Get the current costmap

`C = DS.costmap_get()` is the current costmap. The cost map is the same size as the occupancy grid and the value of each element represents the cost of traversing the cell. It is autogenerated by the class constructor from the occupancy grid such that:

- free cell (occupancy 0) has a cost of 1
- occupied cell (occupancy >0) has a cost of Inf

See also

Dstar.costmap_set, Dstar.costmap_modify

DstarPO.costmap_modify

Modify cost map

`DS.costmap_modify(P, NEW)` modifies the cost map at `P`=[X,Y] to have the value `NEW`. If `P` ($2 \times M$) and `NEW` ($1 \times M$) then the cost of the points defined by the columns of `P` are set to the corresponding elements of `NEW`.

Notes

- After one or more point costs have been updated the path should be replanned by calling DS.plan().
- Replaces modify_cost, same syntax.

See also

Dstar.costmap_set, Dstar.costmap_get

DstarPO.costmap_set

Set the current costmap

`DS.costmap_set(C)` sets the current costmap. The cost map is the same size as the occupancy grid and the value of each element represents the cost of traversing the cell. A high value indicates that the cell is more costly (difficult) to traverese. A value of Inf indicates an obstacle.

Notes

- After the cost map is changed the path should be replanned by calling DS.plan().

See also

Dstar.costmap_get, Dstar.costmap_modify

DstarPO.distancemap_get

Get the current distance map

`C = DS.distancemap_get()` is the current distance map. This map is the same size as the occupancy grid and the value of each element is the shortest distance from the corresponding point in the map to the current goal. It is computed by Dstar.plan.

See also

Dstar.plan

DstarPO.INSERT

state X to the openlist with objective space values

specified by pt.

DstarPO.plan

Plan path to goal

`DS.plan()` updates DS with a costmap of distance to the goal from every non-obstacle point in the map. The goal is as specified to the constructor.

`DS.plan(GOAL)` as above but uses the specified goal.

Note

- If a path has already been planned, but the costmap was modified, then reinvoking this method will replan,
- incrementally updating the plan at lower cost than a full
- replan.

Inputs:

goal: goal state coordinates N: number of optimization objectives; standard D* is 2 (i.e. distance and heuristic)

DstarPO.plot

Visualize navigation environment

`DS.plot()` displays the occupancy grid and the goal distance in a new figure. The goal distance is shown by intensity which increases with distance from the goal. Obstacles are overlaid and shown in red.

`DS.plot(P)` as above but also overlays a path given by the set of points `P` ($M \times 2$).

See also

Navigation.plot

DstarPO.PROCESS_STATE

with the lowest cost value are removed from the

open list

DstarPO.projectCost

the projection of state a into objective space. If

specified, location is moving from b to a.

DstarPO.reset

Reset the planner

`DS.reset()` resets the D* planner. The next instantiation of DS.plan() will perform a global replan.

DstarPO.updateCosts

Only for costs that accumulate (i.e. sum) over the

path, and for dynamic costs. E.g. the heuristic parameter DS.cost_h only needs updating when the goal state changes; it's values are stored for each cell.

Location moving from state b to a.

Dubbins

path planner sample code

`P = Dubbins(q0, qf, maxc, dl)` finds the shortest path between configurations `q0` and `qf` where each is a vector [x y theta]. `maxc` is the maximum curvature

The robot can only move forwards and the path consists of 3 segments which have zero or maximum curvature `maxc`. There are discontinuities in velocity and steering commands (cusps) at the transitions between the segments.

Example

```
q0 = [1 1 pi/4]'; qf = [1 1 pi]';
p = Dubbins(q0, qf, 1, 0.05)
p.plot('circles', 'k--', 'join', {'Marker', 'o', 'MarkerFaceColor', 'k'});
```

or alternatively

```
Dubbins.test
```

References

- Dubins, L.E. On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents
- American Journal of Mathematics. 79(3), July 1957, pp497?516.
- doi:10.2307/2372560.

Acknowledgement

- Based on python code from Python Robotics by Atsushi Sakai https://github.com/AtsushiSakai/PythonRobotics

See also Navigation, ReedsShepp

Dubbins.generate_path

a list of all possible words

Dubbins.mod2pi

= theta - 2.0 * π * floor(theta / 2.0 / π)

Dubbins.pi_2_pi

= (angle + π) % (2 * math.π) - math.π

Dubbins.plot

Plot Dubbins path

`DP.plot(OPTIONS)` plots the optimal **Dubbins** path.

Options

'circle',LS	Plot the full circle corresponding to each curved segment
'join',LS	Plot a marker at the intermediate segment boundaries

Notes

- LS can be a simple LineSpec string or a cell array of Name,Value pairs.

DXform

Distance transform navigation class

A concrete subclass of the abstract Navigation class that implements the distance transform navigation algorithm which computes minimum distance paths.

Methods

DXform	Constructor
plan	Compute the cost map given a goal and map
query	Find a path
plot	Display the distance function and obstacle map
plot3d	Display the distance function as a surface
display	Print the parameters in human readable form
char	Convert to string

Properties (read only)

distancemap	Distance from each point to the goal.
metric	The distance metric, can be 'euclidean'(default) or 'cityblock'

Example

```
load map1              % load map
goal = [50,30];        % goal point
start = [20, 10];      % start point
dx = DXform(map);      % create navigation object
dx.plan(goal)          % create plan for specified goal
dx.query(start)        % animate path from this start location
```

Notes

- Obstacles are represented by NaN in the distancemap.
- The value of each element in the distancemap is the shortest distance from the corresponding point in the map to the current goal.

References

- Robotics, Vision & Control, Sec 5.2.1, Peter Corke, Springer, 2011.

See also

Navigation, Dstar, PRM, distancexform

DXform.DXform

Distance transform constructor

`DX = DXform(MAP, OPTIONS)` is a distance transform navigation object, and `MAP` is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied).

Options

'goal',G	Specify the goal point (2×1)
'metric',M or 'cityblock'.	Specify the distance metric as 'euclidean'(default)
'inflate',K	Inflate all obstacles by K cells.

Other options are supported by the Navigation superclass.

See also

Navigation.Navigation

DXform.char

Convert to string

`DX.char()` is a string representing the state of the object in human-readable form.

See also **DXform**.display, Navigation.char

DXform.plan

Plan path to goal

`DX.plan(GOAL, OPTIONS)` plans a path to the goal given to the constructor, updates the internal distancemap where the value of each element is the minimum distance from the corresponding point to the goal.

`DX.plan(GOAL, OPTIONS)` as above but goal is specified explicitly

Options

'animate'	Plot the distance transform as it evolves

Notes

- This may take many seconds.

See also

Navigation.path

DXform.plot

Visualize navigation environment

`DX.plot(OPTIONS)` displays the occupancy grid and the goal distance in a new figure. The goal distance is shown by intensity which increases with distance from the goal. Obstacles are overlaid and shown in red.

`DX.plot(P, OPTIONS)` as above but also overlays a path given by the set of points `P` ($M \times 2$).

Notes

- See Navigation.plot for options.

See also

Navigation.plot

DXform.plot3d

3D costmap view

`DX.plot3d()` displays the distance function as a 3D surface with distance from goal as the vertical axis. Obstacles are "cut out" from the surface.

`DX.plot3d(P)` as above but also overlays a path given by the set of points `P` ($M \times 2$).

`DX.plot3d(P, LS)` as above but plot the line with the MATLAB linestyle `LS`.

See also

Navigation.plot

EKF

Extended Kalman Filter for navigation

Extended Kalman filter for optimal estimation of state from noisy measurments given a non-linear dynamic model. This class is specific to the problem of state estimation for a vehicle moving in $\mathrm{SE}(2)$.

This class can be used for:

- dead reckoning localization
- map-based localization
- map making
- simultaneous localization and mapping (SLAM)

It is used in conjunction with:

- a kinematic vehicle model that provides odometry output, represented by a Vehicle sbuclass object.
- The vehicle must be driven within the area of the map and this is achieved by connecting the Vehicle subclass object to a Driver object.
- a map containing the position of a number of landmark points and is represented by a LandmarkMap object.
- a sensor that returns measurements about landmarks relative to the vehicle's pose and is represented by a Sensor object subclass.

The EKF object updates its state at each time step, and invokes the state update methods of the vehicle object. The complete history of estimated state and covariance is stored within the EKF object.

Methods

run	run the filter
plot_xy	plot the actual path of the vehicle
plot_P	plot the estimated covariance norm along the path
plot_map	plot estimated landmark points and confidence limits
plot_vehicle	plot estimated vehicle covariance ellipses
plot_error	plot estimation error with standard deviation bounds
display	print the filter state in human readable form
char	convert the filter state to human readable string

Properties

x_est	estimated state
P	estimated covariance

V_est	estimated odometry covariance
W_est	estimated sensor covariance

landmarks	maps sensor landmark id to filter state element

robot	reference to the Vehicle object
sensor	reference to the Sensor subclass object
history each time step	vector of structs that hold the detailed filter state from
verbose	show lots of detail (default false)
joseph	use Joseph form to represent covariance (default true)

Vehicle position estimation (localization)

Create a vehicle with odometry covariance V, add a driver to it, create a Kalman filter with estimated covariance V_est and initial state covariance P0

`veh = Vehicle(V);` `veh`.add_driver(RandomPath(20, 2)); ekf = EKF(`veh`, `V`_est, P0);

We run the simulation for 1000 time steps

`ekf.run`(1000);

then plot true vehicle path

`veh.plot_xy('b');`

and overlay the estimated path

`ekf.plot_xy('r');`

and overlay uncertainty ellipses

`ekf.plot_ellipse('g');`

We can plot the covariance against time as

clf `ekf`.plot_P();

Map-based vehicle localization

Create a vehicle with odometry covariance `V`, add a driver to it, create a map with 20 point landmarks, create a sensor that uses the map and vehicle state to estimate landmark range and bearing with covariance W, the Kalman filter with estimated covariances `V`_est and W_est and initial vehicle state covariance P0

`veh = Bicycle(V);` `veh`.add_driver(RandomPath(20, 2)); map = LandmarkMap(20); sensor = RangeBearingSensor(`veh`, map, W); `ekf` = EKF(`veh`, `V`_est, P0, sensor, W_est, map);

We `run` the simulation for 1000 time steps ekf.run(1000);

then plot the map and the true vehicle path

map.plot(); veh.plot_xy('b');

and overlay the estimatd path

ekf.plot_xy('r');

and overlay uncertainty ellipses

ekf.plot_ellipse('g');

We can plot the covariance against time as

clf ekf.plot_P();

Vehicle-based map making

Create a vehicle with odometry covariance V, add a driver to it, create a sensor that uses the map and vehicle state to estimate landmark range and bearing with covariance W, the Kalman filter with estimated sensor covariance W_est and a "perfect" vehicle (no covariance), then run the filter for N time steps.

veh = Vehicle(V); veh.add_driver(RandomPath(20, 2)); map = LandmarkMap(20); sensor = RangeBearingSensor(veh, map, W); ekf = EKF(veh, [], [], sensor, W_est, []);

We run the simulation for 1000 time steps

ekf.run(1000);

Then plot the true map

map.plot();

and overlay the estimated map with 97% confidence ellipses

ekf.plot_map('g', 'confidence', 0.97);

Simultaneous localization and mapping (SLAM)

Create a vehicle with odometry covariance V, add a driver to it, create a map with 20 point landmarks, create a sensor that uses the map and vehicle state to estimate landmark range and bearing with covariance W, the Kalman filter with estimated covariances V_est and W_est and initial state covariance P0, then run the filter to estimate the vehicle state at each time step and the map.

veh = Vehicle(V); veh.add_driver(RandomPath(20, 2)); map = PointMap(20); sensor = RangeBearingSensor(veh, map, W); ekf = EKF(veh, V_est, P0, sensor, W, []);

We run the simulation for 1000 time steps

ekf.run(1000);

then `plot` the map and the true vehicle path

`map.plot();veh.plot_xy('b');`

and overlay the estimated path

`ekf.plot_xy('r');`

and overlay uncertainty ellipses

`ekf.plot_ellipse('g');`

We can `plot` the covariance against time as

clf `ekf.plot_P();`

Then `plot` the true map

`map.plot();`

and overlay the estimated map with 3 sigma ellipses

`ekf.plot_map(3, 'g');`

References

Robotics, Vision & Control, Chap 6, Peter Corke, Springer 2011

Stochastic processes and filtering theory, AH Jazwinski Academic Press 1970

Acknowledgement

Inspired by code of Paul Newman, Oxford University, http://www.robots.ox.ac.uk/ pnewman

See also

Vehicle, RandomPath, RangeBearingSensor, PointMap, ParticleFilter

EKF.EKF

EKF object constructor

E = **EKF**(VEHICLE, V_EST, P0, OPTIONS) is an **EKF** that estimates the state of the VEHICLE (subclass of Vehicle) with estimated odometry covariance V_EST (2×2) and initial covariance (3×3).

E = **EKF**(VEHICLE, V_EST, P0, SENSOR, W_EST, MAP, OPTIONS) as above but uses information from a VEHICLE mounted sensor, estimated sensor covariance W_EST and a MAP (LandmarkMap class).

Options

'verbose'	Be verbose.
'nohistory'	Don't keep history.
'joseph'	Use Joseph form for covariance
'dim',D	Dimension of the robot's workspace.

- D scalar; X: -D to +D, Y: -D to +D
- D (1×2); X: -D(1) to +D(1), Y: -D(2) to +D(2)
- D (1×4); X: D(1) to D(2), Y: D(3) to D(4)

Notes

- If MAP is [] then it will be estimated.
- If V_EST and P0 are [] the vehicle is assumed error free and the filter will only estimate the landmark positions (map).
- If V_EST and P0 are finite the filter will estimate the vehicle pose and the landmark positions (map).
- EKF subclasses Handle, so it is a reference object.
- Dimensions of workspace are normally taken from the map if given.

See also

Vehicle, Bicycle, Unicycle, Sensor, RangeBearingSensor, LandmarkMap

EKF.char

Convert to string

`E.char()` is a string representing the state of the **EKF** object in human-readable form.

See also

EKF.display

EKF.display

Display status of EKF object

`E.display()` displays the state of the **EKF** object in human-readable form.

Notes

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

See also

EKF.char

EKF.get_map

Get landmarks

`P = E.get_map()` is the estimated landmark coordinates ($2 \times N$) one per column. If the landmark was not estimated the corresponding column contains NaNs.

See also

EKF.plot_map, EKF.plot_ellipse

EKF.get_P

Get covariance magnitude

`E.get_P()` is a vector of estimated covariance magnitude at each time step.

EKF.get_xy

Get vehicle position

`P = E.get_xy()` is the estimated vehicle pose trajectory as a matrix ($N \times 3$) where each row is x, y, theta.

See also

EKF.plot_xy, EKF.plot_error, EKF.plot_ellipse, EKF.plot_P

EKF.init

Reset the filter

`E.init()` resets the filter state and clears landmarks and history.

EKF.plot_ellipse

Plot vehicle covariance as an ellipse

`E.plot_ellipse()` overlay the current plot with the estimated vehicle position covariance ellipses for 20 points along the path.

`E.plot_ellipse(LS)` as above but pass line style arguments `LS` to plot_ellipse.

Options

'interval',I	Plot an ellipse every I steps (default 20)
'confidence',C	Confidence interval (default 0.95)

See also

plot_ellipse

EKF.plot_error

Plot vehicle position

`E.plot_error(OPTIONS)` plot the error between actual and estimated vehicle path (x, y, theta) versus time. Heading error is wrapped into the range $[-\pi, \pi)$

Options

'bound',S	Display the confidence bounds (default 0.95).
'color',C	Display the bounds using color C
LS	Use MATLAB linestyle LS for the plots

Notes

- The bounds show the instantaneous standard deviation associated with the state. Observations tend to decrease the uncertainty
- while periods of dead-reckoning increase it.
- Set bound to zero to not draw confidence bounds.
- Ideally the error should lie "mostly" within the +/-3sigma bounds.

See also

EKF.plot_xy, EKF.plot_ellipse, EKF.plot_P

EKF.plot_map

Plot landmarks

`E.plot_map(OPTIONS)` overlay the current plot with the estimated landmark position (a +-marker) and a covariance ellipses.

`E.plot_map(LS, OPTIONS)` as above but pass line style arguments `LS` to plot_ellipse.

Options

'confidence',C	Draw ellipse for confidence value C (default 0.95)

See also

EKF.get_map, EKF.plot_ellipse

EKF.plot_P

Plot covariance magnitude

`E.plot_P()` plots the estimated covariance magnitude against time step.

`E.plot_P(LS)` as above but the optional line style arguments `LS` are passed to plot.

EKF.plot_xy

Plot vehicle position

`E.plot_xy()` overlay the current plot with the estimated vehicle path in the xy-plane.

`E.plot_xy(LS)` as above but the optional line style arguments `LS` are passed to plot.

See also

EKF.get_xy, EKF.plot_error, EKF.plot_ellipse, EKF.plot_P

EKF.run

Run the filter

`E.run(N, OPTIONS)` runs the filter for `N` time steps and shows an animation of the vehicle moving.

Options

'plot'	Plot an animation of the vehicle moving

Notes

- All previously estimated states and estimation history are initially cleared.

ETS

Elementary Transform Sequence class

Manipulate a sequence (vector) of elementary transformations

- ETS.TX
- ETS.TY
- ETS.TZ

- ETS.RX
- ETS.RY
- ETS.RZ

Methods

ETS	Construct a sequence from string
isrot	True if rotational transform
istrans	True if translational transform
isjoint	Is ETS a function of qj
njoints	Maximum joint variable index
axis	Axis of translation or rotation
find	Find ETS that is a function of qj
subs	Substitute element of sequence
eval	Evaluate ETS
jacobian	Compute Jacobian of ETS
display	Display a sequence in human readable form
char	Convert sequence to a string

Example

```
ets = ETS('Rx(q1)Tx(a1)Ry(q2)Ty(a3)Rz(q3)Rx(pi/2)')
ets.eval([1 2 3]);
```

Notes

- Still experimental

See also

trchain, trchain2

ETS.ETS

Construct elementary transform element or sequence

`e = ETS()` is a new **ETS** object.

`e = ETS(t)` is a clone of the **ETS** object `t` and all properties are copied.

`e = ETS(op, v)` is a new **ETS** object of type `op` and value `v`. OP can be any of

'Rx'	rotation about the x-axis
'Ry'	rotation about the y-axis
'Rz'	rotation about the z-axis
'Tx'	translation along the x-axis
'Ty'	translation along the y-axis
'Tz'	translation along the z-axis
'transl'	sequence of finite translations along the x-, y- and z-directions.
'rpy'	sequence of finite rotations about the x-, y- and z-directions.

`e = ETS(str)` is a sequence of **ETS** objects, each described by a subexpression in the string STR. Each subexpression comprises an operation as per the table above followed by parentheses and a value. For example:

```
ets = ETS('Rx(q1)Tx(a1)Ry(q2)Ty(a3)Rz(q3)Rx(pi/2)')
```

ETS.display

Display parameters

`ETS.display()` displays the transform parameters in compact single line format.

Notes

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

See also

Link.char, Link.dyn, SerialLink.showlink

ETS2

Elementary transform sequence in 2D

This class and package allows experimentation with sequences of spatial transformations in 2D.

```
import ETS2.*
a1 = 1; a2 = 1;
E = Rz('q1') * Tx(a1) * Rz('q2') * Tx(a2)
```

Operation methods

fkine	forward kinematics

Information methods

isjoint	test if transform is a joint
njoints	the number of joint variables

structure a string listing the joint types

Display methods

display	display value as a string
plot	graphically display the sequence as a robot
teach	graphically display as robot and allow user control

Conversion methods

char	convert to string
string	convert to string with symbolic variables

Operators

*	compound two elementary transforms
+	compound two elementary transforms

Notes

- The sequence is an array of objects of superclass ETS2, but with distinct subclasses: Rz, Tx, Ty.
- Use the command 'clear imports'after using ETS3.

See also

ETS3

ETS2.ETS2

Create an ETS2 object

`E = ETS2(W, V)` is a new **ETS2** object that defines an elementary transform where `W` is 'Rz', 'Tx'or 'Ty'and `V` is the paramter for the transform. If `V` is a string of the form 'qN'where N is an integer then the transform is considered to be a joint. Otherwise the transform is a constant.

`E = ETS2(E1)` is a new **ETS2** object that is a clone of the **ETS2** object `E1`.

See also

ETS2.Rz, ETS2.Tx, ETS2.Ty

ETS2.char

Convert to string

`E.char()` is a string showing transform parameters in a compact format. If E is a transform sequence ($1 \times N$) then the string describes each element in sequence in a single line format.

See also

ETS2.display

ETS2.display

Display parameters

`E.display()` displays the transform or transform sequence parameters in compact single line format.

Notes

- This method is invoked implicitly at the command line when the result of an expression is an ETS2 object and the command has no trailing
- semicolon.

See also

ETS2.char

ETS2.find

Find joints in transform sequence

`E.find(J)` is the index in the transform sequence ETS ($1 \times N$) corresponding to the `J`'th joint.

ETS2.fkine

Forward kinematics

`ETS.fkine(Q, OPTIONS)` is the forward kinematics, the pose of the end of the sequence as an SE2 object. `Q` ($1 \times N$) is a vector of joint variables.

`ETS.fkine(Q, N, OPTIONS)` as above but process only the first `N` elements of the transform sequence.

Options

'deg'	Angles are given in degrees.

ETS2.isjoint

Test if transform is a joint

`E.isjoint` is true if the transform element is a joint, that is, its parameter is of the form 'qN'.

ETS2.isprismatic

Test if transform is prismatic joint

`E.isprismatic` is true if the transform element is a joint, that is, its parameter is of the form 'qN'and it controls a translation.

ETS2.mtimes

Compound transforms

`E1 * E2` is a sequence of two elementary transform.

See also

ETS2.plus

ETS2.n

Number of joints in transform sequence

`E.njoints` is the number of joints in the transform sequence.

Notes

- Is a wrapper on `njoints`, for compatibility with SerialLink object.

See also

ETS2.n

ETS2.njoints

Number of joints in transform sequence

`E.njoints` is the number of joints in the transform sequence.

See also

ETS2.n

ETS2.plot

Graphical display and animation

`ETS.plot(Q, options)` displays a graphical animation of a robot based on the transform sequence. Constant translations are represented as pipe segments, rotational joints as cylinder, and prismatic joints as boxes. The robot is displayed at the joint angle `Q` ($1 \times N$), or if a matrix ($M \times N$) it is animated as the robot moves along the M-point trajectory.

Options

'workspace', W	Size of robot 3D workspace, W = [xmn, xmx ymn ymx zm
'floorlevel',L	Z-coordinate of floor (default -1)
'delay',D	Delay betwen frames for animation (s)
'fps',fps	Number of frames per second for display, inverse of 'delay'
'[no]loop'	Loop over the trajectory forever
'[no]raise'	Autoraise the figure
'movie',M	Save an animation to the movie M
'trail',L	Draw a line recording the tip path, with line style L
'scale',S	Annotation scale factor
'zoom',Z robot look bigger	Reduce size of auto-computed workspace by Z, makes
'ortho'	Orthographic view
'perspective'	Perspective view (default)
'view',V plan view, or general view by azimuth and elevation	Specify view V='x', 'y', 'top'or [az el] for side elevations, angle.
'top'	View from the top.
'[no]shading'	Enable Gouraud shading (default true)
'lightpos',L	Position of the light source (default [0 0 20])
'[no]name'	Display the robot's name
'[no]wrist'	Enable display of wrist coordinate frame
'xyz'	Wrist axis label is XYZ
'noa'	Wrist axis label is NOA
'[no]arrow'	Display wrist frame with 3D arrows
'[no]tiles'	Enable tiled floor (default true)
'tilesize',S	Side length of square tiles on the floor (default 0.2)

'tile1color',C	Color of even tiles [r g b] (default [0.5 1 0.5] light green)
'tile2color',C	Color of odd tiles [r g b] (default [1 1 1] white)

'[no]shadow'	Enable display of shadow (default true)
'shadowcolor',C	Colorspec of shadow, [r g b]
'shadowwidth',W	Width of shadow line (default 6)
'[no]jaxes'	Enable display of joint axes (default false)
'[no]jvec'	Enable display of joint axis vectors (default false)
'[no]joints'	Enable display of joints
'jointcolor',C	Colorspec for joint cylinders (default [0.7 0 0])
'jointcolor',C	Colorspec for joint cylinders (default [0.7 0 0])
'jointdiam',D	Diameter of joint cylinder in scale units (default 5)
'linkcolor',C	Colorspec of links (default 'b')
'[no]base'	Enable display of base 'pedestal'
'basecolor',C	Color of base (default 'k')
'basewidth',W	Width of base (default 3)

The `options` come from 3 sources and are processed in order:

- Cell array of `options` returned by the function PLOTBOTOPT (if it exists)
- Cell array of `options` given by the 'plotopt'option when creating the SerialLink object.
- List of arguments in the command line.

Many boolean `options` can be enabled or disabled with the 'no'prefix. The various option sources can toggle an option, the last value encountered is used.

Graphical annotations and `options`

The robot is displayed as a basic stick figure robot with annotations such as:

- shadow on the floor
- XYZ wrist axes and labels
- joint cylinders and axes

which are controlled by `options`.

The size of the annotations is determined using a simple heuristic from the workspace dimensions. This dimension can be changed by setting the multiplicative scale factor using the 'mag'option.

Figure behaviour

- If no figure exists one will be created and the robot drawn in it.
- If no robot of this name is currently displayed then a robot will be drawn in the current figure. If hold is enabled (hold on) then the
- robot will be added to the current figure.
- If the robot already exists then that graphical model will be found and moved.

Notes

- The `options` are processed when the figure is first drawn, to make different `options` come into effect it is neccessary to clear the figure.
- Delay betwen frames can be eliminated by setting option 'delay', 0 or 'fps', Inf.
- The size of the plot volume is determined by a heuristic for an all-revolute robot. If a prismatic joint is present the 'workspace'option is
- required. The 'zoom'option can reduce the size of this workspace.

See also

ETS2.teach, SerialLink.plot3d

ETS2.plus

Compound transforms

`E1 + E2` is a sequence of two elementary transform.

See also

ETS2.mtimes

ETS2.string

Convert to string with symbolic variables

`E.string` is a `string` representation of the transform sequence where non-joint parameters have symbolic names L1, L2, L3 etc.

See also

trchain

ETS2.structure

Show joint type structure

E.structure is a character array comprising the letters 'R'or 'P'that indicates the types of joints in the elementary transform sequence E.

Notes

- The string will be E.njoints long.

See also

SerialLink.config

ETS2.teach

Graphical teach pendant

Allow the user to "drive" a graphical robot using a graphical slider panel.

ETS.teach(OPTIONS) adds a slider panel to a current ETS plot. If no graphical robot exists one is created in a new window.

ETS.teach(Q, OPTIONS) as above but the robot joint angles are set to Q ($1 \times N$).

Options

'eul'	Display tool orientation in Euler angles (default)
'rpy'	Display tool orientation in roll/pitch/yaw angles
'approach'	Display tool orientation as approach vector (z-axis)
'[no]deg'	Display angles in degrees (default true)

GUI

- The Quit (red X) button removes the teach panel from the robot plot.

Notes

- The currently displayed robots move as the sliders are adjusted.
- The slider limits are derived from the joint limit properties. If not set then for
  - a revolute joint they are assumed to be [-π, +π]
  - a prismatic joint they are assumed unknown and an error occurs.

See also

ETS2.plot

ETS3

Elementary transform sequence in 3D

This class and package allows experimentation with sequences of spatial transformations in 3D.

```
import +ETS3.*
L1 = 0; L2 = -0.2337; L3 = 0.4318; L4 = 0.0203; L5 = 0.0837; L6 = 0.4318;
E3 = Tz(L1) * Rz('q1') * Ry('q2') * Ty(L2) * Tz(L3) * Ry('q3') * Tx(L4) * Ty(L5) * Tz(L6)
```

Operation methods

fkine

Information methods

isjoint	test if transform is a joint
njoints	the number of joint variables

structure a string listing the joint types

Display methods

display	display value as a string
plot	graphically display the sequence as a robot
teach	graphically display as robot and allow user control

Conversion methods

char	convert to string
string	convert to string with symbolic variables

Operators

*	compound two elementary transforms
+	compound two elementary transforms

Notes

- The sequence is an array of objects of superclass ETS3, but with distinct subclasses: Rx, Ry, Rz, Tx, Ty, Tz.
- Use the command 'clear imports'after using ETS2.

See also

ETS2

ETS3.ETS3

Create an ETS3 object

`E = ETS3(W, V)` is a new **ETS3** object that defines an elementary transform where `W` is 'Rx', 'Ry', 'Rz', 'Tx', 'Ty'or 'Tz'and `V` is the paramter for the transform. If `V` is a string of the form 'qN'where N is an integer then the transform is considered to be a joint and the parameter is ignored. Otherwise the transform is a constant.

`E = ETS3(E1)` is a new **ETS3** object that is a clone of the **ETS3** object `E1`.

See also

ETS2.Rz, ETS2.Tx, ETS2.Ty

ETS3.char

Convert to string

`E.char()` is a string showing transform parameters in a compact format. If E is a transform sequence ($1 \times N$) then the string describes each element in sequence in a single line format.

See also

ETS3.display

ETS3.display

Display parameters

`E.display()` displays the transform or transform sequence parameters in compact single line format.

Notes

- This method is invoked implicitly at the command line when the result of an expression is an ETS3 object and the command has no trailing
- semicolon.

See also

ETS3.char

ETS3.find

Find joints in transform sequence

`E.find(J)` is the index in the transform sequence ETS($1 \times N$) corresponding to the `J`'th joint.

ETS3.fkine

Forward kinematics

`ETS.fkine(Q, OPTIONS)` is the forward kinematics, the pose of the end of the sequence as an SE3 object. `Q` ($1 \times N$) is a vector of joint variables.

`ETS.fkine(Q, N, OPTIONS)` as above but process only the first `N` elements of the transform sequence.

Options

'deg'	Angles are given in degrees.

ETS3.isjoint

Test if transform is a joint

`E.isjoint` is true if the transform element is a joint, that is, its parameter is of the form 'qN'.

ETS3.isprismatic

Test if transform is prismatic joint

`E.isprismatic` is true if the transform element is a joint, that is, its parameter is of the form 'qN'and it controls a translation.

ETS3.mtimes

Compound transforms

`E1 * E2` is a sequence of two elementary transform.

See also

ETS3.plus

ETS3.n

Number of joints in transform sequence

`E.njoints` is the number of joints in the transform sequence.

Notes

- Is a wrapper on `njoints`, for compatibility with SerialLink object.

See also

ETS3.n

ETS3.njoints

Number of joints in transform sequence

`E.njoints` is the number of joints in the transform sequence.

See also

ETS2.n

ETS3.plot

Graphical display and animation

`ETS.plot(Q, options)` displays a graphical animation of a robot based on the transform sequence. Constant translations are represented as pipe segments, rotational joints as cylinder, and prismatic joints as boxes. The robot is displayed at the joint angle `Q` $(1 \times N)$, or if a matrix $(M \times N)$ it is animated as the robot moves along the M-point trajectory.

Options

'workspace', W	Size of robot 3D workspace, W = [xmn, xmx ymn ymx zm
'floorlevel',L	Z-coordinate of floor (default -1)
'delay',D	Delay betwen frames for animation (s)

'fps',fps	Number of frames per second for display, inve
'[no]loop'	Loop over the trajectory forever
'[no]raise'	Autoraise the figure
'movie',M	Save an animation to the movie M
'trail',L	Draw a line recording the tip path, with line st

'scale',S	Annotation scale factor
'zoom',Z robot look bigger	Reduce size of auto-computed workspace by Z
'ortho'	Orthographic view
'perspective'	Perspective view (default)
'view',V plan view, or general view by azimuth and elevation	Specify view V='x', 'y', 'top'or [az el] for side e angle.
'top'	View from the top.

'[no]shading'	Enable Gouraud shading (default true)
'lightpos',L	Position of the light source (default [0 0 20])
'[no]name'	Display the robot's name

'[no]wrist'	Enable display of wrist coordinate frame
'xyz'	Wrist axis label is XYZ
'noa'	Wrist axis label is NOA
'[no]arrow'	Display wrist frame with 3D arrows

'[no]tiles'	Enable tiled floor (default true)
'tilesize',S	Side length of square tiles on the floor (default

'tile1color',C	Color of even tiles [r g b] (default [0.5 1 0.5] light green)
'tile2color',C	Color of odd tiles [r g b] (default [1 1 1] white)

'[no]shadow'	Enable display of shadow (default true)
'shadowcolor',C	Colorspec of shadow, [r g b]
'shadowwidth',W	Width of shadow line (default 6)

'[no]jaxes'	Enable display of joint axes (default false)
'[no]jvec'	Enable display of joint axis vectors (default false)
'[no]joints'	Enable display of joints
'jointcolor',C	Colorspec for joint cylinders (default [0.7 0 0])
'jointcolor',C	Colorspec for joint cylinders (default [0.7 0 0])
'jointdiam',D	Diameter of joint cylinder in scale units (default 5)

'linkcolor',C	Colorspec of links (default 'b')

'[no]base'	Enable display of base 'pedestal'
'basecolor',C	Color of base (default 'k')
'basewidth',W	Width of base (default 3)

The `options` come from 3 sources and are processed in order:

- Cell array of `options` returned by the function PLOTBOTOPT (if it exists)
- Cell array of `options` given by the 'plotopt'option when creating the SerialLink object.
- List of arguments in the command line.

Many boolean `options` can be enabled or disabled with the 'no'prefix. The various option sources can toggle an option, the last value encountered is used.

Graphical annotations and `options`

The robot is displayed as a basic stick figure robot with annotations such as:

- shadow on the floor
- XYZ wrist axes and labels
- joint cylinders and axes

which are controlled by `options`.

The size of the annotations is determined using a simple heuristic from the workspace dimensions. This dimension can be changed by setting the multiplicative scale factor using the 'mag'option.

Figure behaviour

- If no figure exists one will be created and the robot drawn in it.
- If no robot of this name is currently displayed then a robot will be drawn in the current figure. If hold is enabled (hold on) then the
- robot will be added to the current figure.
- If the robot already exists then that graphical model will be found and moved.

Notes

- The `options` are processed when the figure is first drawn, to make different `options` come into effect it is neccessary to clear the figure.
- Delay betwen frames can be eliminated by setting option 'delay', 0 or 'fps', Inf.
- The size of the plot volume is determined by a heuristic for an all-revolute robot. If a prismatic joint is present the 'workspace'option is
- required. The 'zoom'option can reduce the size of this workspace.

See also

ETS3.teach, SerialLink.plot3d

ETS3.plus

Compound transforms

`E1 + E2` is a sequence of two elementary transform.

See also

ETS3.mtimes

ETS3.string

Convert to string with symbolic variables

`E.string` is a `string` representation of the transform sequence where non-joint parameters have symbolic names L1, L2, L3 etc.

See also

trchain

ETS3.structure

Show joint type structure

`E.structure` is a character array comprising the letters 'R'or 'P'that indicates the types of joints in the elementary transform sequence `E`.

Notes

- The string will be `E`.njoints long.

See also

SerialLink.config

ETS3.teach

Graphical teach pendant

Allow the user to “drive” a graphical robot using a graphical slider panel.

`ETS.teach(OPTIONS)` adds a slider panel to a current ETS plot. If no graphical robot exists one is created in a new window.

`ETS.teach(Q, OPTIONS)` as above but the robot joint angles are set to `Q` ($1 \times N$).

Options

'eul'	Display tool orientation in Euler angles (default)
'rpy'	Display tool orientation in roll/pitch/yaw angles
'approach'	Display tool orientation as approach vector (z-axis)
'[no]deg'	Display angles in degrees (default true)

GUI

- The Quit (red X) button removes the teach panel from the robot plot.

Notes

- The currently displayed robots move as the sliders are adjusted.
- The slider limits are derived from the joint limit properties. If not set then for
  - a revolute joint they are assumed to be [$-\pi$, $+\pi$]
  - a prismatic joint they are assumed unknown and an error occurs.

See also

ETS3.plot

Frame

Coordinate frame object

`F = Frame(P, OPTIONS)` creates an object that graphically renders a coordinate frame for SE(2), SO(2) or SE(3) represented by the pose `P` which can be:

- homogeneous transform (3×3) for $\mathrm{SE}(2)$
- Quaternion for $\mathrm{SO}(3)$
- orthonormal rotation matrix (3×3) for $\mathrm{SO}(3)$
- homogeneous transform (4×4) for $\mathrm{SE}(3)$

Methods

move	move the graphical coordinate frame to a new pose
animate	move the graphical coordinate frame to a new pose

char display delete

Options

'color',C	The color to draw the axes, MATLAB colorspec C
'noaxes'	Don't display axes on the plot
'axis',A	Set dimensions of the MATLAB axes to A=[xmin xmax
'frame',F	The frame is named {F} and the subscript on the axis la
'text_opts', opt	A cell array of MATLAB text properties
'handle',H	Draw in the MATLAB axes specified by the axis handle
'view',V for view toward origin of coordinate frame	Set plot view parameters V=[az el] angles, or 'auto'
'arrow'	Use arrows rather than line segments for the axes
'width', w	Width of arrow tips

Examples

f_a = Frame(TA, 'frame', 'A') f_b = Frame(TB, 'frame', 'B', 'color', 'b') f_c = Frame(TC, 'frame', 'C', 'text_opts', {'FontSize', 10, 'FontWeight', 'bold'})

```
f_a.move(T);
```

Notes

- The arrow option requires the third party package arrow3.

See also

trplot2, tranimate

Frame.animate

Animate a coordinate frame

`ANIMATE(P1, P2, OPTIONS)` animates a 3D coordinate frame moving from pose `P1` to pose `P2`. Poses `P1` and `P2` can be represented by:

- homogeneous transformation matrices (4×4)
- orthonormal rotation matrices (3×3)
- Quaternion

`ANIMATE(P, OPTIONS)` animates a coordinate frame moving from the identity pose to the pose `P` represented by any of the types listed above.

`ANIMATE(PSEQ, OPTIONS)` animates a trajectory, where `PSEQ` is any of

- homogeneous transformation matrix sequence ($4 \times 4 \times N$)
- orthonormal rotation matrix sequence ($3 \times 3 \times N$)
- Quaternion vector ($N \times 1$)

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]

See also

trplot

Frame.char

String representation of parameters

`s = L.char()` is a string showing link parameters in compact single line format. If L is a vector of Link objects return a string with one line per Link.

See also

Link.display

Frame.delete

Delete the coordinate frame

Frame.display

Display parameters

`F.display()` display link parameters in compact single line format. If L is a vector of Link objects display one line per element.

Notes

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

See also

Link.char, Link.dyn, SerialLink.showlink

---

getprofilefunctionstats

Summary of this function goes here

Detailed explanation goes here

Author

Joern Malzahn, (joern.malzahn@tu-dortmund.de)

---

joy2tr

Update transform from joystick

`T = JOY2TR(T, OPTIONS)` updates the SE(3) homogeneous transform (4×4) according to spatial velocities sourced from a connected joystick device.

Options

'delay',D	Pause for D seconds after reading (default 0.1)
'scale',S	A 2-vector which scales joystick translational and rotational to rates (default [0.5m/s, 0.25rad/s])
'world'	Joystick motion is in the world frame
'tool'	Joystick motion is in the tool frame (default)
'rotate',R	Index of the button used to enable rotation (default 7)

Notes

- Joystick axes 0,1,3 map to X,Y,Z or R,P,Y motion.
- A joystick button enables the mapping to translation OR rotation.
- A 'delay'of zero means no pause
- If 'delay'is non-zero 'scale'maps full scale to m/s or rad/s.
- If 'delay'is zero 'scale'maps full scale to m/sample or rad/sample.

See also

joystick

joystick

Input from joystick

`J = JOYSTICK()` returns a vector of joystick values in the range -1 to +1.

`[J,B] = JOYSTICK()` as above but also returns a vector of button values, either 0 (not pressed) or 1 (pressed).

Notes

- This is a MEX file that uses SDL (www.libsdl.org) to interface to a standard gaming joystick.
- The length of the vectors `J` and `B` depend on the capabilities of the joystick identified when it is first opened.

See also

joy2tr

---

jsingu

Show the linearly dependent joints in a Jacobian matrix

`JSINGU(J)` displays the linear dependency of joints in a Jacobian matrix. This dependency indicates joint axes that are aligned and causes singularity.

See also

SerialLink.jacobn

---

jtraj

Compute a joint space trajectory

`[Q,QD,QDD] = JTRAJ(Q0, QF, M)` is a joint space trajectory `Q` ($M \times N$) where the joint coordinates vary from `Q0` ($1 \times N$) to `QF` ($1 \times N$). A quintic (5th order) polynomial is used with default zero boundary conditions for velocity and acceleration. Time is assumed to vary from 0 to 1 in `M` steps. Joint velocity and acceleration can be optionally returned as `QD` ($M \times N$) and `QDD` ($M \times N$) respectively. The trajectory `Q`, `QD` and `QDD` are $M \times N$ matrices, with one row per time step, and one column per joint.

`[Q,QD,QDD] = JTRAJ(Q0, QF, M, QD0, QDF)` as above but also specifies initial `QD0` ($1 \times N$) and final `QDF` ($1 \times N$) joint velocity for the trajectory.

`[Q,QD,QDD] = JTRAJ(Q0, QF, T)` as above but the number of steps in the trajectory is defined by the length of the time vector `T` ($M \times 1$).

`[Q,QD,QDD] = JTRAJ(Q0, QF, T, QD0, QDF)` as above but specifies initial and final joint velocity for the trajectory and a time vector.

Notes

- When a time vector is provided the velocity and acceleration outputs are scaled assumign that the time vector starts at zero and increases
- linearly.

See also

qplot, ctraj, SerialLink.jtraj

LandmarkMap

Map of planar point landmarks

A LandmarkMap object represents a square 2D environment with a number of landmark landmark points.

Methods

plot	Plot the landmark map

landmark	Return a specified map landmark

display	Display map parameters in human readable form
char	Convert map parameters to human readable string

Properties

map	Matrix of map landmark coordinates $2 \times N$
dim	The dimensions of the map region x,y in [-dim,dim]

nlandmarks	The number of map landmarks N

Examples

To create a map for an area where X and Y are in the range -10 to +10 metres and with 50 random landmark points

```
map = LandmarkMap(50, 10);
```

which can be displayed by

```
map.plot();
```

Reference

Robotics, Vision & Control, Chap 6, Peter Corke, Springer 2011

See also

RangeBearingSensor, EKF

LandmarkMap.LandmarkMap

Create a map of point landmark landmarks

`M = LandmarkMap(N, DIM, OPTIONS)` is a **LandmarkMap** object that represents `N` random point landmarks in a planar region bounded by +/-`DIM` in the x- and y-directions.

Options

'verbose'	Be verbose

LandmarkMap.char

Convert map parameters to a string

`s = M.char()` is a string showing map parameters in a compact human readable format.

LandmarkMap.display

Display map parameters

`M.display()` displays map parameters in a compact human readable form.

Notes

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

See also

map.char

LandmarkMap.landmark

Get landmarks from map

`F = M.landmark(K)` is the coordinate (2×1) of the `K`'th landmark (landmark).

LandmarkMap.plot

Plot the map

`M.plot()` plots the landmark map in the current figure, as a square region with dimensions given by the M.dim property. Each landmark is marked by a black diamond.

`M.plot(LS)` as above, but the arguments `LS` are passed to plot and override the default marker style.

Notes

- The plot is left with HOLD ON.

LandmarkMap.show

Show the landmark map

Notes

- Deprecated, use plot method.

LandmarkMap.verbosity

Set verbosity

`M.verbosity(V)` set verbosity to `V`, where 0 is silent and greater values display more information.

Lattice

Lattice planner navigation class

A concrete subclass of the abstract Navigation class that implements the lattice planner navigation algorithm over an occupancy grid. This performs goal independent planning of kinematically feasible paths.

Methods

Lattice	Constructor
plan	Compute the roadmap
query	Find a path
plot	Display the obstacle map
display	Display the parameters in human readable form
char	Convert to string

Properties (read only)

graph	A PGraph object describign the tree

Example

```
lp = Lattice();                      % create navigation object
lp.plan('iterations', 8)             % create roadmaps
lp.query( [1 2 pi/2], [2 -2 0] )     % find path
lp.plot();                           % plot the path
```

References

- Robotics, Vision & Control, Section 5.2.4, P. Corke, Springer 2016.

See also

Navigation, DXform, Dstar, PGraph

Lattice.Lattice

Create a Lattice navigation object

`P = Lattice(MAP, options)` is a probabilistic roadmap navigation object, and `MAP` is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied).

Options

'grid',G	Grid spacing in X and Y (default 1)
'root',R	Root coordinate of the lattice (2×1) (default [0,0])
'iterations',N	Number of sample points (default Inf)
'cost',C	Cost for straight, left, right (default [1,1,1])
'inflate',K	Inflate all obstacles by K cells.

Other `options` are supported by the Navigation superclass.

Notes

- Iterates until the area defined by the map is covered.

See also

Navigation.Navigation

Lattice.char

Convert to string

`P.char()` is a string representing the state of the **Lattice** object in human-readable form.

See also

Lattice.display

Lattice.plan

Create a lattice plan

`P.plan(OPTIONS)` creates the lattice by iteratively building a tree of possible paths. The resulting graph is kept within the object.

Options

'iterations',N	Number of sample points (default Inf)
'cost',C	Cost for straight, left, right (default [1,1,1])

Default parameter values come from the constructor

Lattice.plot

Visualize navigation environment

`P.plot()` displays the occupancy grid with an optional distance field.

Options

'goal'	Superimpose the goal position if set
'nooverlay'	Don't overlay the Lattice graph

Lattice.query

Find a path between two poses

`P.query(START, GOAL)` finds a path ($N \times 3$) from pose `START` (1×3) to pose `GOAL` (1×3). The pose is expressed as [X,Y,THETA].

---

bresenham

Generate a line

`P = BRESENHAM(X1, Y1, X2, Y2)` is a list of integer coordinates ($2 \times N$) for points lying on the line segment joining the integer coordinates (`X1`,`Y1`) and (`X2`,`Y2`).

`P = BRESENHAM(P1, P2)` as above but `P1`=[`X1`; `Y1`] and `P2`=[`X2`; `Y2`].

Notes

- Endpoint coordinates must be integer values.

Author

- Based on code by Aaron Wetzler

See also

icanvas

---

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 circumferance 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 $N \times 3$, but 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)

colorname

Map between color names and RGB values

`RGB = COLORNAME(NAME)` is the RGB-tristimulus value (1×3) corresponding to the color specified by the string NAME. If RGB is a cell-array ($1 \times N$) of names then RGB is a matrix ($N \times 3$) with each row being the corresponding tristimulus.

`XYZ = COLORNAME(NAME, 'xyz')` as above but the XYZ-tristimulus value corresponding to the color specified by the string NAME.

`XY = COLORNAME(NAME, 'xy')` as above but the xy-chromaticity coordinates corresponding to the color specified by the string NAME.

`NAME = COLORNAME(RGB)` is a string giving the name of the color that is closest (Euclidean) to the given RGB-tristimulus value (1×3). If RGB is a matrix ($N \times 3$) then return a cell-array ($1 \times N$) of color names.

`NAME = COLORNAME(XYZ, 'xyz')` as above but the color is the closest (Euclidean) to the given XYZ-tristimulus value.

`NAME = COLORNAME(XYZ, 'xy')` as above but the color is the closest (Euclidean) to the given xy-chromaticity value with assumed Y=1.

Notes

- Color name may contain a wildcard, eg. “?burnt”
- Based on the standard X11 color database rgb.txt.
- Tristimulus values are in the range 0 to 1

diff2

First-order difference

`D = DIFF2(V)` is the first-order difference ($1 \times N$) of the series data in vector `V` ($1 \times N$) and the first element is zero.

`D = DIFF2(A)` is the first-order difference ($M \times N$) of the series data in each row of the matrix `A` ($M \times N$) and the first element in each row is zero.

Notes

- Unlike the builtin function DIFF, the result of DIFF2 has the same number of columns as the input.

See also

diff

---

dockfigs

Control figure docking in the GUI

dockfigs causes all new figures to be docked into the GUI

dockfigs(1) as above.

dockfigs(0) causes all new figures to be undocked from the GUI

---

edgelist

Return list of edge pixels for region

`EG = EDGELIST(IM, SEED)` is a list of edge pixels ($2 \times N$) of a region in the image `IM` starting at edge coordinate `SEED`=[X,Y]. The edgelist has one column per edge point coordinate (x,y).

`EG = EDGELIST(IM, SEED, DIRECTION)` as above, but the direction of edge following is specified. `DIRECTION` == 0 (default) means clockwise, non zero is counter-clockwise. Note that direction is with respect to y-axis upward, in matrix coordinate frame, not image frame.

`[EG,D] = EDGELIST(IM, SEED, DIRECTION)` as above but also returns a vector of edge segment directions which have values 1 to 8 representing W SW S SE E NW N NW respectively.

Notes

- Coordinates are given assuming the matrix is an image, so the indices are always in the form (x,y) or (column,row).
- `IM` is a binary image where 0 is assumed to be background, non-zero is an object.
- `SEED` must be a point on the edge of the region.
- The seed point is always the first element of the returned edgelist.
- 8-direction chain coding can give incorrect results when used with blobs founds using 4-way connectivty.

Reference

- METHODS TO ESTIMATE AREAS AND PERIMETERS OF BLOB-LIKE OBJECTS: A COMPARISON Luren Yang, Fritz Albregtsen, Tor Lgnnestad and Per Grgttum
- IAPR Workshop on Machine Vision Applications Dec. 13-15, 1994, Kawasaki

See also

ilabel

---

filt1d

1-dimensional rank filter

`Y = FILT1D(X, OPTIONS)` is the minimum, maximum or median value ($1 \times N$) of the vector `X` ($1 \times N$) compute over an odd length sliding window.

Options

'max'	Compute maximum value over the window (default)
'min'	Compute minimum value over the window

'median'	Compute minimum value over the window
'width',W	Width of the window (default 5)

Notes

- If the window width is even, it is incremented by one.
- The first and last elements of X are replicated so the output vector is the same length as the input vector.

gaussfunc

kernel

`G = GAUSSFUNC(MEAN, VARIANCE, X)` is the value of the normal distribution (Gaussian) function with `MEAN` (1×1) and `VARIANCE` (1×1), at

the point `X`.

`G = GAUSSFUNC(MEAN, COVARIANCE, X, Y)` is the value of the bivariate normal distribution (Gaussian) function with `MEAN` (1×2) and `COVARIANCE` (2×2), at the point (`X`,`Y`).

`G = GAUSSFUNC(MEAN, COVARIANCE, X)` as above but `X` $(N \times M)$ and the result is also $(N \times M)$. `X` and `Y` values come from the column and row indices of `X`.

Notes

- `X` or `Y` can be row or column vectors, and the result will also be a vector.
- The area or volume under the curve is unity.

mmlabel

for mplot style graph

mmlabel({lab1 lab2 lab3})

Notes

- was previously (rev 9) named mlabel() but changed to avoid clash with the Mapping Toolbox.

mplot

Plot time-series data

A convenience function for plotting time-series data held in a matrix. Each row is a timestep and the first column is time.

`MPLOT(Y, OPTIONS)` plots the time series data `Y`($N \times M$) in multiple subplots. The first column is assumed to be time, so M-1 plots are produced.

`MPLOT(T, Y, OPTIONS)` plots the time series data `Y`($N \times M$) in multiple subplots. Time is provided explicitly as the first argument so M plots are produced.

`MPLOT(S, OPTIONS)` as above but `S` is a structure. Each field is assumed to be a time series which is plotted. Time is taken from the field called 't'. Plots are labelled according to the name of the corresponding field.

`MPLOT(W, OPTIONS)` as above but `W` is a structure created by the Simulink write to workspace block where the save format is set to "Structure with time". Each field in the signals substructure is plotted.

`MPLOT(R, OPTIONS)` as above but `R` is a Simulink.SimulationOutput object returned by the Simulink sim() function.

Options

'col',C column indices in the range 1 to M-1	Select columns to plot, a boolean of length M-1 or a list of
'label',L	Label the axes according to the cell array of strings L
'date'	Add a datestamp in the top right corner

Notes

- In all cases a simple GUI is created which is invoked by a right clicking on one of the plotted lines. The supported options are:
  - zoom in the x-direction
  - shift view to the left or right
  - unzoom
  - show data points

See also

plot2, plotp

mtools

simple/useful tools to all windows in figure

pickregion

Pick a rectangular region of a figure using mouse

`[p1,p2] = PICKREGION()` initiates a rubberband box at the current click point and animates it so long as the mouse button remains down. Returns the first and last coordinates in axis units.

Options

'axis',A	The axis to select from (default current axis)
'ls',LS	Line style for foreground line (default ':y');
'bg'LS,	Line style for background line (default '-k');
'width',W	Line width (default 2)
'pressed'	Don't wait for first button press, use current position

Notes

- Effectively a replacement for the builtin rbbox function which draws the box in the wrong location on my Mac's external monitor.

Author

Based on rubberband box from MATLAB Central written/Edited by Bob Hamans (B.C.Hamans@student.tue.nl) 02-04-2003, in turn based on an idea of Sandra Martinka's Rubberline.

plotp

Plot trajectory

Convenience function to plot points stored columnwise.

PLOTP(P) plots a set of points P, which by Toolbox convention are stored one per column. P can be $2 \times N$ or $3 \times N$. By default a linestyle of 'bx'is used.

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

See also

plot, plot2

---

polydiff

Differentiate a polynomial

PD = POLYDIFF(P) is a vector of coefficients of a polynomial $(1 \times N-1)$ which is the derivative of the polynomial P $(1 \times N)$.

```
p = [3 2 -1];
polydiff(p)
ans =
     6     2
```

See also

polyval

Polygon

Polygon class

A general class for manipulating polygons and vectors of polygons.

Methods

plot	Plot polygon
area	Area of polygon
moments	Moments of polygon
centroid	Centroid of polygon
perimeter	Perimter of polygon
transform	Transform polygon
inside	Test if points are inside polygon
intersection	Intersection of two polygons
difference	Difference of two polygons
union	Union of two polygons
xor	Exclusive or of two polygons
display	print the polygon in human readable form
char	convert the polgyon to human readable string

Properties

vertices	List of polygon vertices, one per column
extent	Bounding box [minx maxx; miny maxy]
n	Number of vertices

Notes

- This is reference class object
- Polygon objects can be used in vectors and arrays

Acknowledgement

The methods: inside, intersection, difference, union, and xor are based on code written by:

Kirill K. Pankratov, kirill@plume.mit.edu, `http://puddle.mit.edu/` glenn/kirill/saga.html

and require a licence. However the author does not respond to email regarding the licence, so use with care, and modify with acknowledgement.

Polygon.Polygon

Polygon class constructor

`P = Polygon(V)` is a polygon with vertices given by `V`, one column per vertex.

`P = Polygon(C, WH)` is a rectangle centred at `C` with dimensions `WH`=[WIDTH, HEIGHT].

Polygon.area

Area of polygon

`A = P.area()` is the area of the polygon.

See also

Polygon.moments

Polygon.centroid

Centroid of polygon

`X = P.centroid()` is the centroid of the polygon.

See also

Polygon.moments

Polygon.char

String representation

`S = P.char()` is a compact representation of the polgyon in human readable form.

Polygon.difference

Difference of polygons

`D = P.difference(Q)` is polygon P minus polygon `Q`.

Notes

- If polygons P and `Q` are not intersecting, returns coordinates of P.
- If the result `D` is not simply connected or consists of several polygons, resulting vertex list will contain NaNs.

Polygon.display

Display polygon

`P.display()` displays the polygon in a compact human readable form.

See also

Polygon.char

Polygon.inside

Test if points are inside polygon

`IN = P.inside(P)` tests if points given by columns of `P` ($2 \times N$) are inside the polygon. The corresponding elements of `IN` ($1 \times N$) are either true or false.

Polygon.intersect

Intersection of polygon with list of polygons

`I = P.intersect(PLIST)` indicates whether or not the **Polygon** P intersects with

`i(j)` = 1 if p intersects polylist(`j`), else 0.

Polygon.intersect_line

Intersection of polygon and line segment

`I = P.intersect_line(L)` is the intersection points of a polygon P with the line segment `L`=[x1 x2; y1 y2]. `I` ($2 \times N$) has one column per intersection, each column is [x y]'.

Polygon.intersection

Intersection of polygons

`I = P.intersection(Q)` is a **Polygon** representing the intersection of polygons P and `Q`.

Notes

- If these polygons are not intersecting, returns empty polygon.
- If intersection consist of several disjoint polygons (for non-convex P or `Q`) then vertices of `I` is the concatenation
- of the vertices of these polygons.

Polygon.moments

Moments of polygon

`A = P.moments(p, q)` is the pq'th moment of the polygon.

See also

Polygon.area, Polygon.centroid, mpq_poly

Polygon.perimeter

Perimeter of polygon

`L = P.perimeter()` is the perimeter of the polygon.

Polygon.plot

Draw polygon

`P.plot()` draws the polygon P in the current plot.

`P.plot(LS)` as above but pass the arguments `LS` to plot.

Notes

- The polygon is added to the current plot.

Polygon.transform

Transform polygon vertices

`P2 = P.transform(T)` is a new **Polygon** object whose vertices have been transformed by the SE(2) homgoeneous transformation `T` (3×3).

Polygon.union

Union of polygons

`I = P.union(Q)` is a polygon representing the union of polygons P and `Q`.

Notes

- If these polygons are not intersecting, returns a polygon with vertices of both polygons separated by NaNs.
- If the result P is not simply connected (such as a polygon with a "hole") the resulting contour consist of counter-
- clockwise "outer boundary" and one or more clock-wise
- "inner boundaries" around "holes".

Polygon.xor

Exclusive or of polygons

`I = P.union(Q)` is a polygon representing the exclusive-or of polygons P and `Q`.

Notes

- If these polygons are not intersecting, returns a polygon with vertices of both polygons separated by NaNs.
- If the result P is not simply connected (such as a polygon with a "hole") the resulting contour consist of counter-

- clockwise “outer boundary” and one or more clock-wise
- “inner boundaries” around “holes”.

randinit

Reset random number generator

RANDINIT resets the defaul random number stream.

See also

RandStream

runscript

Run an M-file in interactive fashion

`RUNSCRIPT(SCRIPT, OPTIONS)` runs the M-file `SCRIPT` and pauses after every executable line in the file until a key is pressed. Comment lines are shown without any delay between lines.

Options

'delay',D	Don't wait for keypress, just delay of D seconds (default 0)
'cdelay',D	Pause of D seconds after each comment line (default 0)
'begin'	Start executing the file after the comment line %%begin (default false)
'dock'	Cause the figures to be docked when created
'path',P	Look for the file `SCRIPT` in the folder P (default .)
'dock'	Dock figures within GUI
'nocolor'	Don't use cprintf to print lines in color (comments black, code blue)

Notes

- If no file extension is given in `SCRIPT`, .m is assumed.
- A copyright text block will be skipped and not displayed.

- If cprintf exists and 'nocolor'is not given then lines are displayed in color.
- Leading comment characters are not displayed.
- If the executable statement has comments immediately afterward (no blank lines) then the pause occurs after those comments are displayed.
- A simple '-'prompt indicates when the script is paused, hit enter.
- If the function cprintf() is in your path, the display is more colorful. You can get this file from MATLAB File Exchange.
- If the file has a lot of boilerplate, you can skip over and not display it by giving the 'begin'option which searchers for the first line
- starting with %%begin and commences execution at the line after that.

See also

eval

---

rvcpath

Install location of RVC tools

p = RVCPATH is the path of the top level folder for the installed RVC tools.

`p = RVCPATH(FOLDER)` is the full path of the specified `FOLDER` which is relative to the installed RVC tools.

---

stlRead

reads any STL file not depending on its format

`[v, f, n, name] = stlRead(fileName)` reads the STL format file (ASCII or binary) and returns vertices V, faces F, normals N and NAME 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

usefig

figure windows

`usefig('Foo')` makes figure 'Foo'the current figure, if it doesn't exist create it.

`h = usefig('Foo')` as above, but returns the figure handle

xaxis

Set X-axis scaling

`XAXIS(MAX)` set x-axis scaling from 0 to `MAX`.

`XAXIS(MIN, MAX)` set x-axis scaling from `MIN` to `MAX`.

`XAXIS([MIN MAX])` as above.

XAXIS restore automatic scaling for x-axis.

See also

yaxis

yaxis

Y-axis scaling

`YAXIS(MAX)` set y-axis scaling from 0 to `MAX`.

`YAXIS(MIN, MAX)` set y-axis scaling from `MIN` to `MAX`.

`YAXIS([MIN MAX])` as above.

YAXIS restore automatic scaling for y-axis.

See also

yaxis

---

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 submatrix.

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$\neq$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)$\neq$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 ($N \times 1$) 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 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 ($N \times 3$) 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 [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
~=	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 RTB-Pose 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 $\mathrm{SE}(2)$.

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 sub-matrix.

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
$\sim=$	^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 submatrix.

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 infinitesimal 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 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 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 $\mathrm{SO}(2)$ 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 $\mathrm{SO}(2)$.

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 $\mathrm{SO}(3)$.

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.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 $\mathrm{SO}(3)$ 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 homogenous 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 $(4 \times 4 \times \texttt{M})$ of homogeneous interpolating between identity matrix and `T1` in `M` steps.

Notes

- `T0` or `T1` can also be an $\mathrm{SO}(3)$ 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 $\mathrm{SE}(2)$ 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 $(4 \times 4 \times \texttt{M})$ of homogeneous transforms linearly interpolating between `T0` and `T1` in `M` steps.

`TRINTERP2(T1, M)` as above but return a sequence $(4 \times 4 \times \texttt{M})$ of homogeneous interpolating between identity matrix and `T1` in `M` steps.

Notes

- `T0` or `T1` can also be an $\mathrm{SO}(2)$ 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 $[0,\pi]$.
- 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(H, T)` 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 r
'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 'au
'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 homogeneous 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.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 $\mathrm{SO}(3)$ 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

Link

manipulator Link class

A Link object holds all information related to a robot joint and link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

Constructors

Link	general constructor
Prismatic	construct a prismatic joint+link using standard DH
PrismaticMDH	construct a prismatic joint+link using modified DH
Revolute	construct a revolute joint+link using standard DH
RevoluteMDH	construct a revolute joint+link using modified DH

Information/display methods

display	print the link parameters in human readable form
dyn	display link dynamic parameters
type	joint type: 'R'or 'P'

Conversion methods

char	convert to string

Operation methods

A	link transform matrix
friction	friction force
nofriction	Link object with friction parameters set to zero%

Testing methods

islimit	test if joint exceeds soft limit
isrevolute	test if joint is revolute
isprismatic	test if joint is prismatic
issym	test if joint+link has symbolic parameters

Overloaded operators

+	concatenate links, result is a SerialLink object

Properties (read/write)

theta	kinematic: joint angle
d	kinematic: link offset
a	kinematic: link length
alpha	kinematic: link twist
jointtype	kinematic: 'R'if revolute, 'P'if prismatic
mdh	kinematic: 0 if standard D&H, else 1
offset	kinematic: joint variable offset
qlim	kinematic: joint variable limits [min max]
m	dynamic: link mass
r	dynamic: link COG wrt link coordinate frame 3×1
I	dynamic: link inertia matrix, symmetric 3×3, about link COG.
B	dynamic: link viscous friction (motor referred)
Tc	dynamic: link Coulomb friction
G	actuator: gear ratio
Jm	actuator: motor inertia (motor referred)

Examples

```
L = Link([0 1.2 0.3 pi/2]);
L = Link('revolute', 'd', 1.2, 'a', 0.3, 'alpha', pi/2);
L = Revolute('d', 1.2, 'a', 0.3, 'alpha', pi/2);
```


Notes

- This is a reference class object.
- Link objects can be used in vectors and arrays.
- Convenience subclasses are Revolute, Prismatic, RevoluteMDH and PrismaticMDH.

References

- Robotics, Vision & Control, P. Corke, Springer 2011, Chap 7.

See also

Link, Revolute, Prismatic, SerialLink, RevoluteMDH, PrismaticMDH

Link.Link

Create robot link object

This the class constructor which has several call signatures.

`L = Link()` is a **Link** object with default parameters.

`L = Link(LNK)` is a **Link** object that is a deep copy of the link object `LNK` and has type Link, even if `LNK` is a subclass.

`L = Link(OPTIONS)` is a link object with the kinematic and dynamic parameters specified by the key/value pairs.

Options

'theta',TH	joint angle, if not specified joint is revolute
'd',D	joint extension, if not specified joint is prismatic
'a',A	joint offset (default 0)
'alpha',A	joint twist (default 0)
'standard'	defined using standard D&H parameters (default).
'modified'	defined using modified D&H parameters.
'offset',O	joint variable offset (default 0)
'qlim',L	joint limit (default [])
'I',I	link inertia matrix (3×1, 6×1 or 3×3)
'r',R	link centre of gravity (3×1)
'm',M	link mass (1×1)
'G',G	motor gear ratio (default 1)
'B',B	joint friction, motor referenced (default 0)

'Jm',J	motor inertia, motor referenced (default 0)
'Tc',T	Coulomb friction, motor referenced (1×1 or 2×1), (default [0 0])
'revolute'	for a revolute joint (default)
'prismatic'	for a prismatic joint 'p'
'standard'	for standard D&H parameters (default).
'modified'	for modified D&H parameters.
'sym'	consider all parameter values as symbolic not numeric

Notes

- It is an error to specify both 'theta'and 'd'
- The joint variable, either theta or d, is provided as an argument to the A() method.
- The link inertia matrix (3×3) is symmetric and can be specified by giving a 3×3 matrix, the diagonal elements [Ixx Iyy Izz], or the moments and products
- of inertia [Ixx Iyy Izz Ixy Iyz Ixz].
- All friction quantities are referenced to the motor not the load.
- Gear ratio is used only to convert motor referenced quantities such as friction and interia to the link frame.

Old syntax

`L = Link(DH, OPTIONS)` is a link object using the specified kinematic convention and with parameters:

- `DH` = [THETA D A ALPHA SIGMA OFFSET] where SIGMA=0 for a revolute and 1 for a prismatic joint; and OFFSET is a constant displacement between the
- user joint variable and the value used by the kinematic model.
- `DH` = [THETA D A ALPHA SIGMA] where OFFSET is zero.
- `DH` = [THETA D A ALPHA], joint is assumed revolute and OFFSET is zero.

Options

'standard'	for standard D&H parameters (default).
'modified'	for modified D&H parameters.
'revolute'	for a revolute joint, can be abbreviated to 'r'(default)
'prismatic'	for a prismatic joint, can be abbreviated to 'p'

Notes

- The parameter D is unused in a revolute joint, it is simply a placeholder in the vector and the value given is ignored.

- The parameter THETA is unused in a prismatic joint, it is simply a placeholder in the vector and the value given is ignored.

Examples

A standard Denavit-Hartenberg link

```
L3 = Link('d', 0.15005, 'a', 0.0203, 'alpha', -pi/2);
```

since 'theta'is not specified the joint is assumed to be revolute, and since the kinematic convention is not specified it is assumed 'standard'.

Using the old syntax

```
L3 = Link([ 0, 0.15005, 0.0203, -pi/2], 'standard');
```

the flag 'standard'is not strictly necessary but adds clarity. Only 4 parameters are specified so sigma is assumed to be zero, ie. the joint is revolute.

```
L3 = Link([ 0, 0.15005, 0.0203, -pi/2, 0], 'standard');
```

the flag 'standard'is not strictly necessary but adds clarity. 5 parameters are specified and sigma is set to zero, ie. the joint is revolute.

```
L3 = Link([ 0, 0.15005, 0.0203, -pi/2, 1], 'standard');
```

the flag 'standard'is not strictly necessary but adds clarity. 5 parameters are specified and sigma is set to one, ie. the joint is prismatic.

For a modified Denavit-Hartenberg revolute joint

```
L3 = Link([ 0, 0.15005, 0.0203, -pi/2, 0], 'modified');
```

Notes

- Link object is a reference object, a subclass of Handle object.
- Link objects can be used in vectors and arrays.
- The joint offset is a constant added to the joint angle variable before forward kinematics and subtracted after inverse kinematics. It is useful
- if you want the robot to adopt a 'sensible'pose for zero joint angle
- configuration.
- The link dynamic (inertial and motor) parameters are all set to zero. These must be set by explicitly assigning the object
- properties: m, r, I, Jm, B, Tc.
- The gear ratio is set to 1 by default, meaning that motor friction and inertia will be considered if they are non-zero.

See also

Revolute, Prismatic, RevoluteMDH, PrismaticMDH

Link.A

Link transform matrix

`T = L.A(Q)` is an SE3 object representing the transformation between link frames when the link variable `Q` which is either the Denavit-Hartenberg parameter THETA (revolute) or D (prismatic). For:

- standard DH parameters, this is from the previous frame to the current.
- modified DH parameters, this is from the current frame to the next.

Notes

- For a revolute joint the THETA parameter of the link is ignored, and `Q` used instead.
- For a prismatic joint the D parameter of the link is ignored, and `Q` used instead.
- The link offset parameter is added to `Q` before computation of the transformation matrix.

See also

SerialLink.fkine

Link.char

Convert to string

`s = L.char()` is a string showing link parameters in a compact single line format. If L is a vector of Link objects return a string with one line per Link.

See also

Link.display

Link.display

Display parameters

`L.display()` displays the link parameters in compact single line format. If L is a vector of Link objects displays one line per element.

Notes

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

See also

Link.char, Link.dyn, SerialLink.showlink

Link.dyn

Show inertial properties of link

`L.dyn()` displays the inertial properties of the link object in a multi-line format. The properties shown are mass, centre of mass, inertia, friction, gear ratio and motor properties.

If L is a vector of **Link** objects show properties for each link.

See also

SerialLink.dyn

Link.friction

Joint friction force

`F = L.friction(QD)` is the joint friction force/torque ($1 \times N$) for joint velocity `QD` ($1 \times N$). The friction model includes:

- Viscous friction which is a linear function of velocity.
- Coulomb friction which is proportional to sign(`QD`).

Notes

- The friction value should be added to the motor output torque, it has a negative value when `QD`>0.
- The returned friction value is referred to the output of the gearbox.
- The friction parameters in the Link object are referred to the motor.
- Motor viscous friction is scaled up by G^2.
- Motor Coulomb friction is scaled up by G.
- The appropriate Coulomb friction value to use in the non-symmetric case depends on the sign of the joint velocity, not the motor velocity.
- The absolute value of the gear ratio is used. Negative gear ratios are tricky: the Puma560 has negative gear ratio for joints 1 and 3.

See also

Link.nofriction

Link.horzcat

Concatenate link objects

[L1 L2] is a vector that contains deep copies of the **Link** class objects L1 and L2.

Notes

- The elements of the vector are all of type Link.
- If the elements were of a subclass type they are convered to type Link.
- Extends to arbitrary number of objects in list.

See also

Link.plus

Link.islimit

Test joint limits

`L.islimit(Q)` is true (1) if `Q` is outside the soft limits set for this joint.

Note

- The limits are not currently used by any Toolbox functions.

Link.isprismatic

Test if joint is prismatic

`L.isprismatic()` is true (1) if joint is prismatic.

See also

Link.isrevolute

Link.isrevolute

Test if joint is revolute

`L.isrevolute()` is true (1) if joint is revolute.

See also

Link.isprismatic

Link.issym

Check if link is a symbolic model

`res = L.issym()` is true if the **Link** L has any symbolic parameters.

See also

Link.sym

Link.nofriction

Remove friction

`LN = L.nofriction()` is a link object with the same parameters as L except non-linear (Coulomb) friction parameter is zero.

`LN = L.nofriction('all')` as above except that viscous and Coulomb friction are set to zero.

`LN = L.nofriction('coulomb')` as above except that Coulomb friction is set to zero.

`LN = L.nofriction('viscous')` as above except that viscous friction is set to zero.

Notes

- Forward dynamic simulation can be very slow with finite Coulomb friction.

See also

Link.friction, SerialLink.nofriction, SerialLink.fdyn

Link.plus

Concatenate link objects into a robot

`L1+L2` is a SerialLink object formed from deep copies of the **Link** class objects `L1` and `L2`.

Notes

- The elements can belong to any of the Link subclasses.
- Extends to arbitrary number of objects, eg. `L1+L2`+L3+L4.

See also

SerialLink, SerialLink.plus, Link.horzcat

Link.set.I

Set link inertia

`L.I` = [Ixx Iyy Izz] sets link inertia to a diagonal matrix.

`L.I` = [Ixx Iyy Izz Ixy Iyz Ixz] sets link inertia to a symmetric matrix with specified inertia and product of intertia elements.

`L.I` = M set **Link** inertia matrix to M (3×3) which must be symmetric.

Link.set.r

Set centre of gravity

`L.r` = R sets the link centre of gravity (COG) to R (3-vector).

Link.set.Tc

Set Coulomb friction

`L.Tc` = F sets Coulomb friction parameters to [F -F], for a symmetric Coulomb friction model.

`L.Tc` = [FP FM] sets Coulomb friction to [FP FM], for an asymmetric Coulomb friction model. FP>0 and FM<0. FP is applied for a positive joint velocity and FM for a negative joint velocity.

Notes

- The friction parameters are defined as being positive for a positive joint velocity, the friction force computed by Link.friction uses the
- negative of the friction parameter, that is, the force opposing motion of
- the joint.

See also

Link.friction

Link.sym

Convert link parameters to symbolic type

`LS = L.sym` is a **Link** object in which all the parameters are symbolic ('`sym`') type.

See also

Link.issym

Link.type

Joint type

`c = L.type()` is a character 'R'or 'P'depending on whether joint is revolute or prismatic respectively. If L is a vector of Link objects return an array of characters in joint order.

See also

SerialLink.config

lspb

Linear segment with parabolic blend

`[S,SD,SDD] = LSPB(S0, SF, M)` is a scalar trajectory ($M \times 1$) that varies smoothly from `S0` to `SF` in `M` steps using a constant velocity segment and parabolic blends (a trapezoidal velocity profile). Velocity and acceleration can be optionally returned as `SD` ($M \times 1$) and `SDD` ($M \times 1$) respectively.

`[S,SD,SDD] = LSPB(S0, SF, M, V)` as above but specifies the velocity of the linear segment which is normally computed automatically.

`[S,SD,SDD] = LSPB(S0, SF, T)` as above but specifies the trajectory in terms of the length of the time vector `T` ($M \times 1$).

`[S,SD,SDD] = LSPB(S0, SF, T, V)` as above but specifies the velocity of the linear segment which is normally computed automatically and a time vector.

`LSPB(S0, SF, M, V)` as above but plots `S`, `SD` and `SDD` versus time in a single figure.

Notes

- If M is given
  - Velocity is in units of distance per trajectory step, not per second.
  - Acceleration is in units of distance per trajectory step squared, not per second squared.
- If T is given then results are scaled to units of time.
- The time vector T is assumed to be monotonically increasing, and time scaling is based on the first and last element.
- For some values of V no solution is possible and an error is flagged.

References

- Robotics, Vision & Control, Chap 3, P. Corke, Springer 2011.

See also

tpoly, jtraj

---

makemap

Make an occupancy map

`map = makemap(N)` is an occupancy grid map (N × N) created by a simple interactive editor. The map is initially unoccupied and obstacles can be added using geometric primitives.

`map = makemap()` as above but N=128.

`map = makemap(map0)` as above but the map is initialized from the occupancy grid map0, allowing obstacles to be added.

With focus in the displayed figure window the following commands can be entered:

left button	click and drag to create a rectangle
p	draw polygon
c	draw circle
u	undo last action
e	erase map
q	leave editing mode and return map

See also

DXForm, PRM, RRT

---

models

Summarise and search available robot models

`MODELS()` lists keywords associated with each of the models in Robotics Toolbox.

`MODELS(QUERY)` lists those models that match the keyword `QUERY`. Case is ignored in the comparison.

`M = MODELS(QUERY)` as above but returns a cell array ($N \times 1$) of the names of the `M`-files that define the models.

Examples

```
models
models('modified_DH')   % all models using modified DH notation
models('kinova')        % all Kinova robot models
models('6dof')          % all 6dof robot models
models('redundant')     % all redundant robot models, >6 DOF
models('prismatic')     % all robots with a prismatic joint
```

Notes

- A model is a file mdl_*.m in the models folder of the RTB directory.
- The keywords are indicated by a line '% MODEL: 'after the main comment block.

---

mdl_ball

Create model of a ball manipulator

MDL_BALL creates the workspace variable ball which describes the kinematic characteristics of a serial link manipulator with 50 joints that folds into a ball shape.

`MDL_BALL(N)` as above but creates a manipulator with `N` joints.

Also define the workspace vectors:

q joint angle vector for default ball configuration

Reference

- "A divide and conquer articulated-body algorithm for parallel O(log(n)) calculation of rigid body dynamics, Part 2",
- Int. J. Robotics Research, 18(9), pp 876-892.

Notes

- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_coil, SerialLink

mdl_baxter

Kinematic model of Baxter dual-arm robot

MDL_BAXTER is a script that creates the workspace variables left and right which describes the kinematic characteristics of the two 7-joint arms of a Rethink Robotics Baxter robot using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY'configuration
qd	lower arm horizontal as per data sheet

Notes

- SI units of metres are used.

References

"Kinematics Modeling and Experimental Verification of Baxter Robot" Z. Ju, C. Yang, H. Ma, Chinese Control Conf, 2015.

See also

mdl_nao, SerialLink

---

mdl_cobra600

Create model of Adept Cobra 600 manipulator

MDL_COBRA600 is a script that creates the workspace variable c600 which describes the kinematic characteristics of the 4-axis Adept Cobra 600 SCARA manipulator using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration

Notes

- SI units are used.

See also

SerialRevolute, mdl_puma560akb, mdl_stanford

---

mdl_coil

Create model of a coil manipulator

MDL_COIL creates the workspace variable coil which describes the kinematic characteristics of a serial link manipulator with 50 joints that folds into a helix shape.

`MDL_BALL(N)` as above but creates a manipulator with `N` joints.

Also defines the workspace vectors:

q joint angle vector for default helical configuration

Reference

- "A divide and conquer articulated-body algorithm for parallel O(log(n)) calculation of rigid body dynamics, Part 2",
- Int. J. Robotics Research, 18(9), pp 876-892.

Notes

- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_ball, SerialLink

---

mdl_fanuc10L

Create kinematic model of Fanuc AM120iB/10L robot

MDL_FANUC10L is a script that creates the workspace variable R which describes the kinematic characteristics of a Fanuc AM120iB/10L robot using standard DH conventions.

Also defines the workspace vector:

q0 mastering position.

Notes

- SI units of metres are used.

Author

Wynand Swart, Mega Robots CC, P/O Box 8412, Pretoria, 0001, South Africa, wynand.swart@gmail.com

See also

mdl_irb140, mdl_m16, mdl_motomanHP6, mdl_puma560, SerialLink

---

mdl_hyper2d

Create model of a hyper redundant planar manipulator

MDL_HYPER2D creates the workspace variable h2d which describes the kinematic characteristics of a serial link manipulator with 10 joints which at zero angles is a straight line in the XY plane.

`MDL_HYPER2D(N)` as above but creates a manipulator with `N` joints.

Also define the workspace vectors:

qz joint angle vector for zero angle configuration

`R = MDL_HYPER2D(N)` functional form of the above, returns the SerialLink object.

`[R,QZ] = MDL_HYPER2D(N)` as above but also returns a vector of zero joint angles.

Notes

- All joint axes are parallel to z-axis.
- The manipulator in default pose is a straight line 1m long.
- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_hyper3d, mdl_coil, mdl_ball, mdl_twolink, SerialLink

---

mdl_hyper3d

Create model of a hyper redundant 3D manipulator

MDL_HYPER3D is a script that creates the workspace variable h3d which describes the kinematic characteristics of a serial link manipulator with 10 joints which at zero angles is a straight line in the XY plane.

`MDL_HYPER3D(N)` as above but creates a manipulator with `N` joints.

Also define the workspace vectors:

qz joint angle vector for zero angle configuration

`R = MDL_HYPER3D(N)` functional form of the above, returns the SerialLink object.

`[R,QZ] = MDL_HYPER3D(N)` as above but also returns a vector of zero joint angles.

Notes

- In the zero configuration joint axes alternate between being parallel to the z- and y-axes.
- A crude snake or elephant trunk robot.
- The manipulator in default pose is a straight line 1m long.
- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_hyper2d, mdl_ball, mdl_coil, SerialLink

---

mdl_irb140

Create model of ABB IRB 140 manipulator

MDL_IRB140 is a script that creates the workspace variable irb140 which describes the kinematic characteristics of an ABB IRB 140 manipulator using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY'configuration
qd	lower arm horizontal as per data sheet

Reference

- "IRB 140 data sheet", ABB Robotics.

- “Utilizing the Functional Work Space Evaluation Tool for Assessing a System Design and Reconfiguration Alternatives”
- A. Djuric and R. J. Urbanic

Notes

- SI units of metres are used.
- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_fanuc10l, mdl_m16, mdl_motormanHP6, mdl_S4ABB2p8, mdl_puma560, SerialLink

---

mdl_irb140_mdh

Create model of the ABB IRB 140 manipulator

MDL_IRB140_MOD is a script that creates the workspace variable irb140 which describes the kinematic characteristics of an ABB IRB 140 manipulator using modified DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration

Reference

- ABB IRB 140 data sheet
- “The modeling of a six degree-of-freedom industrial robot for the purpose of efficient path planning”,
- Master of Science Thesis, Penn State U, May 2009,
- Tyler Carter

See also

mdl_irb140, mdl_puma560, mdl_stanford, mdl_twolink, SerialLink

Notes

- SI units of metres are used.
- The tool frame is in the centre of the tool flange.
- Zero angle configuration has the upper arm vertical and lower arm horizontal.

mdl_jaco

Create model of Kinova Jaco manipulator

MDL_JACO is a script that creates the workspace variable jaco which describes the kinematic characteristics of a Kinova Jaco manipulator using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY'configuration

Reference

- "DH Parameters of Jaco" Version 1.0.8, July 25, 2013.

Notes

- SI units of metres are used.
- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_mico, mdl_puma560, SerialLink

mdl_KR5

Create model of Kuka KR5 manipulator

MDL_KR5 is a script that creates the workspace variable KR5 which describes the kinematic characteristics of a Kuka KR5 manipulator using standard DH conventions.

Also define the workspace vectors:

qk1	nominal working position 1
qk2	nominal working position 2
qk3	nominal working position 3

Notes

- SI units of metres are used.
- Includes an 11.5cm tool in the z-direction

Author

- Gautam Sinha, Indian Institute of Technology, Kanpur.

See also

mdl_irb140, mdl_fanuc10l, mdl_motomanHP6, mdl_S4ABB2p8, mdl_puma560, SerialLink

---

mdl_LWR

Create model of Kuka LWR manipulator

MDL_LWR is a script that creates the workspace variable KR5 which describes the kinematic characteristics of a Kuka KR5 manipulator using standard DH conventions.

Also define the workspace vectors:

qz	all zero angles

Notes

- SI units of metres are used.

Reference

- Identifying the Dynamic Model Used by the KUKA LWR: A Reverse Engineering Approach Claudio Gaz Fabrizio Flacco Alessandro De Luca
- ICRA 2014

See also

mdl_kr5, mdl_irb140, mdl_puma560, SerialLink

---

mdl_M16

Create model of Fanuc M16 manipulator

MDL_M16 is a script that creates the workspace variable m16 which describes the kinematic characteristics of a Fanuc M16 manipulator using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY'configuration
qd	lower arm horizontal as per data sheet

References

- "Fanuc M-16iB data sheet", http://www.robots.com/fanuc/m-16ib.
- "Utilizing the Functional Work Space Evaluation Tool for Assessing a System Design and Reconfiguration Alternatives",
- A. Djuric and R. J. Urbanic

Notes

- SI units of metres are used.
- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_irb140, mdl_fanuc10l, mdl_motomanHP6, mdl_S4ABB2p8, mdl_puma560, SerialLink

---

mdl_mico

Create model of Kinova Mico manipulator

MDL_MICO is a script that creates the workspace variable mico which describes the kinematic characteristics of a Kinova Mico manipulator using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY'configuration

Reference

- "DH Parameters of Mico" Version 1.0.1, August 05, 2013. Kinova

Notes

- SI units of metres are used.
- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

Revolute, mdl_jaco, mdl_puma560, mdl_twolink, SerialLink

---

mdl_motomanHP6

Create kinematic data of a Motoman HP6 manipulator

MDL_MotomanHP6 is a script that creates the workspace variable hp6 which describes the kinematic characteristics of a Motoman HP6 manipulator using standard DH conventions.

Also defines the workspace vector:

q0	mastering position.

Author

Wynand Swart, Mega Robots CC, P/O Box 8412, Pretoria, 0001, South Africa, wynand.swart@gmail.com

Notes

- SI units of metres are used.

See also

mdl_irb140, mdl_m16, mdl_fanuc10l, mdl_S4ABB2p8, mdl_puma560, SerialLink

---

mdl_nao

Create model of Aldebaran NAO humanoid robot

MDL_NAO is a script that creates several workspace variables

leftarm	left-arm kinematics (4DOF)
rightarm	right-arm kinematics (4DOF)
leftleg	left-leg kinematics (6DOF)
rightleg	right-leg kinematics (6DOF)

which are each SerialLink objects that describe the kinematic characteristics of the arms and legs of the NAO humanoid.

Reference

- “Forward and Inverse Kinematics for the NAO Humanoid Robot”, Nikolaos Kofinas,
- Thesis, Technical University of Crete
- July 2012.
- “Mechatronic design of NAO humanoid” David Gouaillier etal.
- IROS 2009, pp. 769-774.

Notes

- SI units of metres are used.
- The base transform of arms and legs are constant with respect to the torso frame, which is assumed to be the constant value when the robot
- is upright. Clearly if the robot is walking these base transforms
- will be dynamic.
- The first reference uses Modified DH notation, but doesn't explicitly mention this, and the parameter tables have the wrong column headings
- for Modified DH parameters.
- TODO; add joint limits
- TODO; add dynamic parameters

See also

mdl_baxter, SerialLink

mdl_offset6

A minimalistic 6DOF robot arm with shoulder offset

MDL_OFFSET6 is a script that creates the workspace variable off6 which describes the kinematic characteristics of a simple arm manipulator with a spherical wrist and a shoulder offset, using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration

Notes

- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_simple6, mdl_puma560, mdl_twolink, SerialLink

mdl_onelink

Create model of a simple 1-link mechanism

MDL_ONELINK is a script that creates the workspace variable tl which describes the kinematic and dynamic characteristics of a simple planar 1-link mechanism.

Also defines the vector:

qz	corresponds to the zero joint angle configuration.

Notes

- SI units are used.
- It is a planar mechanism operating in the XY (horizontal) plane and is therefore not affected by gravity.
- Assume unit length links with all mass (unity) concentrated at the joints.

References

- Based on Fig 3-6 (p73) of Spong and Vidyasagar (1st edition).

See also

mdl_twolink, mdl_planar1, SerialLink

mdl_p8

Create model of Puma robot on an XY base

MDL_P8 is a script that creates the workspace variable p8 which is an 8-axis robot comprising a Puma 560 robot on an XY base. Joints 1 and 2 are the base, joints 3-8 are the robot arm.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY'configuration
qstretch	arm is stretched out in the X direction
qn	arm is at a nominal non-singular configuration

Notes

- SI units of metres are used.

References

- Robotics, Vision & Control, 1st edn, P. Corke, Springer 2011. Sec 7.3.4.

See also

mdl_puma560, SerialLink

---

mdl_panda

Create model of Franka-Emika PANDA robot

MDL_PANDA is a script that creates the workspace variable panda which describes the kinematic characteristics of a Franka-Emika PANDA manipulator using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	arm along +ve x-axis configuration

Reference

- http://www.diag.uniroma1.it/ deluca/rob1_en/WrittenExamsRob1/Robotics1_18.01.11.pdf
- "Dynamic Identification of the Franka Emika Panda Robot With Retrieval of Feasible Parameters Using Penalty-Based Optimization" C. Gaz, M. Cognetti, A. Oliva, P. Robuffo Giordano and A. De Luca
- IEEE Robotics and Automation Letters 4(4), pp. 4147-4154, Oct. 2019, doi: 10.1109/LRA.2019.2931248

Notes

- SI units of metres are used.
- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_sawyer, SerialLink

---

mdl_phantomx

Create model of PhantomX pincher manipulator

MDL_PHANTOMX is a script that creates the workspace variable px which describes the kinematic characteristics of a PhantomX Pincher Robot, a 4 joint hobby class manipulator by Trossen Robotics.

Also define the workspace vectors:

qz	zero joint angle configuration

Notes

- Uses standard DH conventions.
- Tool centrepoint is middle of the fingertips.
- All translational units in mm.

Reference

- `http://www.trossenrobotics.com/productdocs/assemblyguides/phantomx-basic-robot-arm.html`

mdl_planar1

Create model of a simple planar 1-link mechanism

MDL_PLANAR1 is a script that creates the workspace variable p1 which describes the kinematic characteristics of a simple planar 1-link mechanism.

Also defines the vector:

qz corresponds to the zero joint angle configuration.

Notes

- Moves in the XY plane.
- No dynamics in this model.

See also

mdl_planar2, mdl_planar3, SerialLink

mdl_planar2

Create model of a simple planar 2-link mechanism

MDL_PLANAR2 is a script that creates the workspace variable p2 which describes the kinematic characteristics of a simple planar 2-link mechanism.

Also defines the vector:

qz corresponds to the zero joint angle configuration.

Notes

- Moves in the XY plane.
- No dynamics in this model.

See also

mdl_twolink, mdl_planar1, mdl_planar3, SerialLink

---

mdl_planar2_sym

Create model of a simple planar 2-link mechanism

MDL_PLANAR2 is a script that creates the workspace variable p2 which describes the kinematic characteristics of a simple planar 2-link mechanism.

Also defines the vector:

qz corresponds to the zero joint angle configuration.

Also defines the vector:

qz corresponds to the zero joint angle configuration.

Notes

- Moves in the XY plane.
- No dynamics in this model.

See also

mdl_twolink, mdl_planar1, mdl_planar3, SerialLink

---

mdl_planar3

Create model of a simple planar 3-link mechanism

MDL_PLANAR2 is a script that creates the workspace variable p3 which describes the kinematic characteristics of a simple redundant planar 3-link mechanism.

Also defines the vector:

qz	corresponds to the zero joint angle configuration.

Notes

- Moves in the XY plane.
- No dynamics in this model.

See also

mdl_twolink, mdl_planar1, mdl_planar2, SerialLink

---

mdl_puma560

Create model of Puma 560 manipulator

MDL_PUMA560 is a script that creates the workspace variable p560 which describes the kinematic and dynamic characteristics of a Unimation Puma 560 manipulator using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY'configuration
qstretch	arm is stretched out in the X direction
qn	arm is at a nominal non-singular configuration

Notes

- SI units are used.
- The model includes armature inertia and gear ratios.

Reference

- "A search for consensus among model parameters reported for the PUMA 560 robot", P. Corke and B. Armstrong-Helouvry,
- Proc. IEEE Int. Conf. Robotics and Automation, (San Diego),
- pp. 1608-1613, May 1994.

See also

SerialRevolute, mdl_puma560akb, mdl_stanford

mdl_puma560akb

Create model of Puma 560 manipulator

MDL_PUMA560AKB is a script that creates the workspace variable p560m which describes the kinematic and dynamic characterstics of a Unimation Puma 560 manipulator modified DH conventions.

Also defines the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY'configuration
qstretch	arm is stretched out in the X direction

Notes

- SI units are used.

References

- "The Explicit Dynamic Model and Inertial Parameters of the Puma 560 Arm" Armstrong, Khatib and Burdick
- 1986

See also

mdl_puma560, mdl_stanford_mdh, SerialLink

mdl_quadrotor

Dynamic parameters for a quadrotor.

MDL_QUADCOPTER is a script creates the workspace variable quad which describes the dynamic characterstics of a quadrotor flying robot.

Properties

This is a structure with the following elements:

nrotors	Number of rotors (1×1)
J	Flyer rotational inertia matrix (3×3)
h	Height of rotors above CoG (1×1)
d	Length of flyer arms (1×1)
nb	Number of blades per rotor (1×1)
r	Rotor radius (1×1)
c	Blade chord (1×1)
e	Flapping hinge offset (1×1)
Mb	Rotor blade mass (1×1)
Mc	Estimated hub clamp mass (1×1)
ec	Blade root clamp displacement (1×1)
Ib	Rotor blade rotational inertia (1×1)
Ic	Estimated root clamp inertia (1×1)
mb	Static blade moment (1×1)
Ir	Total rotor inertia (1×1)
Ct	Non-dim. thrust coefficient (1×1)
Cq	Non-dim. torque coefficient (1×1)
sigma	Rotor solidity ratio (1×1)
thetat	Blade tip angle (1×1)
theta0	Blade root angle (1×1)
theta1	Blade twist angle (1×1)
theta75	3/4 blade angle (1×1)
thetai	Blade ideal root approximation (1×1)
a	Lift slope gradient (1×1)
A	Rotor disc area (1×1)
gamma	Lock number (1×1)

Notes

- SI units are used.

References

- Design, Construction and Control of a Large Quadrotor micro air vehicle. P.Pounds, PhD thesis,
- Australian National University, 2007.
- http://www.eng.yale.edu/pep5/P_Pounds_Thesis_2008.pdf
- This is a heavy lift quadrotor

See also

sl_quadrotor

---

mdl_S4ABB2p8

Create kinematic model of ABB S4 2.8robot

MDL_S4ABB2p8 is a script that creates the workspace variable s4 which describes the kinematic characteristics of an ABB S4 2.8 robot using standard DH conventions.

Also defines the workspace vector:

q0	mastering position.

Author

Wynand Swart, Mega Robots CC, P/O Box 8412, Pretoria, 0001, South Africa, wynand.swart@gmail.com

See also

mdl_fanuc10l, mdl_m16, mdl_motormanHP6, mdl_irb140, mdl_puma560, SerialLink

---

mdl_sawyer

Create model of Rethink Robotics Sawyer robot

MDL_SAYWER is a script that creates the workspace variable sawyer which describes the kinematic characteristics of a Rethink Robotics Sawyer manipulator using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	arm along +ve x-axis configuration

Reference

- https://sites.google.com/site/daniellayeghi/daily-work-and-writing/major-project-2

Notes

- SI units of metres are used.
- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_baxter, SerialLink

mdl_simple6

A minimalistic 6DOF robot arm

MDL_SIMPLE6 is a script creates the workspace variable s6 which describes the kinematic characteristics of a simple arm manipulator with a spherical wrist and no shoulder offset, using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration

Notes

- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_offset6, mdl_puma560, SerialLink

---

mdl_stanford

Create model of Stanford arm

MDL_STANFORD is a script that creates the workspace variable stanf which describes the kinematic and dynamic characteristics of the Stanford (Scheinman) arm.

Also defines the vectors:

qz	zero joint angle configuration.

Note

- SI units are used.
- Gear ratios not currently known, though reflected armature inertia is known, so gear ratios are set to 1.

References

- Kinematic data from “Modelling, Trajectory calculation and Servoing of a computer controlled arm”. Stanford AIM-177. Figure 2.3
- Dynamic data from “Robot manipulators: mathematics, programming and control” Paul 1981, Tables 6.5, 6.6
- Dobrotin & Scheinman, “Design of a computer controlled manipulator for robot research”, IJCAI, 1973.

See also

mdl_puma560, mdl_puma560akb, SerialLink

mdl_stanford_mdh

Create model of Stanford arm using MDH conventions

MDL_STANFORD is a script that creates the workspace variable stanf which describes the kinematic and dynamic characteristics of the Stanford (Scheinman) arm using modified Denavit-Hartenberg parameters.

Also defines the vectors:

qz zero joint angle configuration.

Notes

- SI units are used.

References

- Kinematic data from "Modelling, Trajectory calculation and Servoing of a computer controlled arm". Stanford AIM-177. Figure 2.3
- Dynamic data from "Robot manipulators: mathematics, programming and control" Paul 1981, Tables 6.5, 6.6

See also

mdl_puma560, mdl_puma560akb, SerialLink

mdl_twolink

Create model of a 2-link mechanism

MDL_TWOLINK is a script that creates the workspace variable twolink which describes the kinematic and dynamic characteristics of a simple planar 2-link mechanism moving in the xz-plane, it experiences gravity loading.

Also defines the vector:

qz corresponds to the zero joint angle configuration.

Notes

- SI units are used.
- It is a planar mechanism operating in the vertical plane and is therefore affected by gravity (unlike mdl_planar2 in the horizontal
- plane).
- Assume unit length links with all mass (unity) concentrated at the joints.

References

- Based on Fig 3-6 (p73) of Spong and Vidyasagar (1st edition).

See also

mdl_twolink_sym, mdl_planar2, SerialLink

mdl_twolink_mdh

Create model of a 2-link mechanism using modified DH convention

MDL_TWOLINK_MDH is a script that the workspace variable twolink which describes the kinematic and dynamic characteristics of a simple planar 2-link mechanism using modified Denavit-Hartenberg conventions.

Also defines the vector:

qz	corresponds to the zero joint angle configuration.

Notes

- SI units of metres are used.
- It is a planar mechanism operating in the xz-plane (vertical) and is therefore not affected by gravity.

References

- Based on Fig 3.8 (p71) of Craig (3rd edition).

See also

mdl_twolink, mdl_onelink, mdl_planar2, SerialLink

mdl_twolink_sym

Create symbolic model of a simple 2-link mechanism

MDL_TWOLINK_SYM is a script that creates the workspace variable twolink which describes in symbolic form the kinematic and dynamic characteristics of a simple planar 2-link mechanism moving in the xz-plane, it experiences gravity loading. The symbolic parameters are:

- link lengths: a1, a2
- link masses: m1, m2
- link CoMs in the link frame x-direction: c1, c2
- gravitational acceleration: g
- joint angles: q1, q2
- joint angle velocities: qd1, qd2
- joint angle accelerations: qdd1, qdd2

Notes

- It is a planar mechanism operating in the vertical plane and is therefore affected by gravity (unlike mdl_planar2 in the horizontal
- plane).
- Gear ratio is 1 and motor inertia is 0.
- Link inertias Iyy1, Iyy2 are 0.
- Viscous and Coulomb friction is 0.

References

- Based on Fig 3-6 (p73) of Spong and Vidyasagar (1st edition).

See also

mdl_puma560, mdl_stanford, SerialLink

mdl_ur10

Create model of Universal Robotics UR10 manipulator

MDL_UR5 is a script that creates the workspace variable ur10 which describes the kinematic characteristics of a Universal Robotics UR10 manipulator using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	arm along +ve x-axis configuration

Reference

- https://www.universal-robots.com/how-tos-and-faqs/faq/ur-faq/actual-center-of-mass-for-robot-17264/

Notes

- SI units of metres are used.
- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_ur3, mdl_ur5, mdl_puma560, SerialLink

mdl_ur3

Create model of Universal Robotics UR3 manipulator

MDL_UR5 is a script that creates the workspace variable ur3 which describes the kinematic characteristics of a Universal Robotics UR3 manipulator using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	arm along +ve x-axis configuration

Reference

- https://www.universal-robots.com/how-tos-and-faqs/faq/ur-faq/actual-center-of-mass-for-robot-17264/

Notes

- SI units of metres are used.
- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_ur5, mdl_ur10, mdl_puma560, SerialLink

---

mdl_ur5

Create model of Universal Robotics UR5 manipulator

MDL_UR5 is a script that creates the workspace variable ur5 which describes the kinematic characteristics of a Universal Robotics UR5 manipulator using standard DH conventions.

Also define the workspace vectors:

qz	zero joint angle configuration
qr	arm along +ve x-axis configuration

Reference

- https://www.universal-robots.com/how-tos-and-faqs/faq/ur-faq/actual-center-of-mass-for-robot-17264/

Notes

- SI units of metres are used.
- Unlike most other mdl_xxx scripts this one is actually a function that behaves like a script and writes to the global workspace.

See also

mdl_ur3, mdl_ur10, mdl_puma560, SerialLink

---

mstraj

Multi-segment multi-axis trajectory

`TRAJ = MSTRAJ(WP, QDMAX, TSEG, Q0, DT, TACC, OPTIONS)` is a trajectory $(K \times N)$ for N axes moving simultaneously through M segment. Each segment is linear motion and polynomial blends connect the segments. The axes start at `Q0` $(1 \times N)$ and pass through M-1 via points defined by the rows of the matrix `WP` $(M \times N)$, and finish at the point defined by the last row of `WP`. The trajectory matrix has one row per time step, and one column per axis. The number of steps in the trajectory K is a function of the number of via points and the time or velocity limits that apply.

- `WP` $(M \times N)$ is a matrix of via points, 1 row per via point, one column per axis. The last via point is the destination.
- `QDMAX` $(1 \times N)$ are axis speed limits which cannot be exceeded,
- `TSEG` $(1 \times M)$ are the durations for each of the K segments
- `Q0` $(1 \times N)$ are the initial axis coordinates
- `DT` is the time step
- `TACC` (1×1) is the acceleration time used for all segment transitions
- `TACC` $(1 \times M)$ is the acceleration time per segment, `TACC`(i) is the acceleration time for the transition from segment i to segment i+1. `TACC`(1) is also
- the acceleration time at the start of segment 1.

TRAJ = MSTRAJ(WP, QDMAX, TSEG, [], DT, TACC, OPTIONS) as above but the initial coordinates are taken from the first row of WP.

TRAJ = MSTRAJ(WP, QDMAX, Q0, DT, TACC, QD0, QDF, OPTIONS) as above but additionally specifies the initial and final axis velocities ($1 \times N$).

Options

'verbose'	Show details.

Notes

- Only one of QDMAX or TSEG can be specified, the other is set to [].
- If no output arguments are specified the trajectory is plotted.
- The path length K is a function of the number of via points, Q0, DT and TACC.
- The final via point P(end,:) is the destination.
- The motion has M segments from Q0 to P(1,:) to P(2,:) ... to P(end,:).
- All axes reach their via points at the same time.
- Can be used to create joint space trajectories where each axis is a joint coordinate.
- Can be used to create Cartesian trajectories where the “axes” correspond to translation and orientation in RPY or Euler angle form.
- If qdmax is a scalar then all axes are assumed to have the same maximum speed.

See also

mtraj, lspb, ctraj

mtraj

Multi-axis trajectory between two points

[Q,QD,QDD] = MTRAJ(TFUNC, Q0, QF, M) is a multi-axis trajectory ($M \times N$) varying from configuration Q0 ($1 \times N$) to QF ($1 \times N$) according to the scalar trajectory function TFUNC in M steps. Joint velocity and acceleration can be optionally returned as QD ($M \times N$) and QDD ($M \times N$) respectively. The trajectory outputs have one row per time step, and one column per axis.

The shape of the trajectory is given by the scalar trajectory function `TFUNC` which is applied to each axis:

`[S,SD,SDD] = TFUNC(S0, SF, M);`

and possible values of `TFUNC` include @lspb for a trapezoidal trajectory, or @tpoly for a polynomial trajectory.

`[Q,QD,QDD] = MTRAJ(TFUNC, Q0, QF, T)` as above but `T` (`M` $\times$ 1) is a time vector which dictates the number of points on the trajectory.

Notes

- If no output arguments are specified `Q`, `QD`, and `QDD` are plotted.
- When `TFUNC` is @tpoly the result is functionally equivalent to JTRAJ except that no initial velocities can be specified. JTRAJ is computationally a little
- more efficient.

See also

jtraj, mstraj, lspb, tpoly

---

multidfprintf

Print formatted text to multiple streams

COUNT = MULTIDFPRINTF(IDVEC, FORMAT, A, ...) performs formatted output to multiple streams such as console and files. FORMAT is the format string as used by sprintf and fprintf. A is the array of elements, to which the format will be applied similar to sprintf and fprint.

IDVEC is a vector ($1 \times N$) of file descriptors and COUNT is a vector ($1 \times N$) of the number of bytes written to each file.

Notes

- To write to the consolde use the file identifier 1.

Example

```
% Create and open a new example file:
fid = fopen('exampleFile.txt','w+');
% Write something to the file and the console simultaneously:
```



```
multidfprintf([1 FID],'% s % d % d % d   % Close the file:
fclose(FID);
```

Authors

Joern Malzahn, (joern.malzahn@tu-dortmund.de)

See also

fprintf, sprintf

Navigation

Navigation superclass

An abstract superclass for implementing planar grid-based navigation classes.

Methods

Navigation	Superclass constructor
plan	Find a path to goal
query	Return/animate a path from start to goal
plot	Display the occupancy grid
display	Display the parameters in human readable form
char	Convert to string
isoccupied	Test if cell is occupied
rand	Uniformly distributed random number
randn	Normally distributed random number
randi	Uniformly distributed random integer

progress_init	Create a progress bar
progress	Update progress bar
progress_delete	Remove progress bar

Properties (read only)

occgrid	Occupancy grid representing the navigation environment
goal	Goal coordinate
start	Start coordinate
seed0	Random number state

Methods that must be provided in subclass

plan	Generate a plan for motion to goal
next	Returns coordinate of next point along path

Methods that may be overriden in a subclass

goal_set	The goal has been changed by nav.goal = (a,b)
navigate_init	Start of path planning.

Notes

- Subclasses the MATLAB handle class which means that pass by reference semantics apply.
- A grid world is assumed and vehicle position is quantized to grid cells.
- Vehicle orientation is not considered.
- The initial random number state is captured as seed0 to allow rerunning an experiment with an interesting outcome.

See also

Bug2, Dstar, Dxform, PRM, Lattice, RRT

Navigation.Navigation

Create a Navigation object

`N = Navigation(OCCGRID, OPTIONS)` is a **Navigation** object that holds an occupancy grid `OCCGRID`. A number of options can be be passed.

Options

'goal',G	Specify the goal point (2×1)
'inflate',K	Inflate all obstacles by K cells.
'private'	Use private random number stream.
'reset'	Reset random number stream.
'verbose'	Display debugging information
'seed',S be a proper random number generator state such as saved in	Set the initial state of the random number stream. S the seed0 property of an earlier run.

Notes

- In the occupancy grid a value of zero means free space and non-zero means occupied (not driveable).
- Obstacle inflation is performed with a round structuring element (kcircle) with radius given by the 'inflate'option.
- Inflation requires either MVTB or IPT installed.
- The 'private'option creates a private random number stream for the methods rand, randn and randi. If not given the global stream is used.

See also

randstream

Navigation.char

Convert to string

`N.char()` is a string representing the state of the navigation object in human-readable form.

Navigation.display

Display status of navigation object

`N.display()` displays the state of the navigation object in human-readable form.

Notes

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

See also

Navigation.char

Navigation.goal_change

Notify change of goal

Invoked when the goal property of the object is changed. Typically this is overriden in a subclass to take particular action such as invalidating a costmap.

Navigation.isoccupied

Test if grid cell is occupied

`N.isoccupied(POS)` is true if there is a valid grid map and the coordinates given by the columns of `POS` ($2 \times N$) are occupied.

`N.isoccupied(X,Y)` as above but the coordinates given separately.

Notes:

- `X` and `Y` are Cartesian rather than MATLAB row-column coordinates.

Navigation.message

Print debug message

`N.message(S)` displays the string `S` if the verbose property is true.

`N.message(FMT, ARGS)` as above but accepts printf() like semantics.

Navigation.navigate_init

Notify start of path

`N.navigate_init(START)` is called when the query() method is invoked. Typically overriden in a subclass to take particular action such as computing some path parameters. `START` (2×1) is the initial position for this path, and nav.goal (2×1) is the final position.

See also

Navigate.query

Navigation.plot

Visualize navigation environment

`N.plot(OPTIONS)` displays the occupancy grid in a new figure.

`N.plot(P, OPTIONS)` as above but overlays the points along the path ($2 \times M$) matrix.

Options

'distance',D a matrix of the same size as the occupancy grid.	Display a distance field D behind the obstacle r
'colormap',@f	Specify a colormap for the distance field as a fu
'beta',B	Brighten the distance field by factor B.
'inflated'	Show the inflated occupancy grid rather than or

Notes

- The distance field at a point encodes its distance from the goal, small distance is dark, a large distance is bright. Obstacles are encoded as
- red.
- Beta value -1<B<0 to darken, 0<B<+1 to lighten.

See also

Navigation.plot_fg, Navigation.plot_bg

Navigation.plot_bg

Visualization background

`N.plot_bg(OPTIONS)` displays the occupancy grid with occupied cells shown as red and an optional distance field.

`N.plot_bg(P,OPTIONS)` as above but overlays the points along the path ($2 \times M$) matrix.

Options

'distance',D a matrix of the same size as the occupancy grid.	Display a distance field D behind the obstacle r
'colormap',@f	Specify a colormap for the distance field as a fu

'beta',B	Brighten the distance field by factor B.
'inflated'	Show the inflated occupancy grid rather than original
'pathmarker',M	Options to draw a path point
'startmarker',M	Options to draw the start marker
'goalmarker',M	Options to draw the goal marker

Notes

- The distance field at a point encodes its distance from the goal, small distance is dark, a large distance is bright. Obstacles are encoded as
- red.
- Beta value -1<B<0 to darken, 0<B<+1 to lighten.

See also

Navigation.plot, Navigation.plot_fg, brighten

Navigation.plot_fg

Visualization foreground

`N.plot_fg(OPTIONS)` displays the start and goal locations if specified. By default the goal is a pentagram and start is a circle.

`N.plot_fg(P, OPTIONS)` as above but overlays the points along the path ($2 \times M$) matrix.

Options

'pathmarker',M	Options to draw a path point
'startmarker',M	Options to draw the start marker
'goalmarker',M	Options to draw the goal marker

Notes

- In all cases M is a single string eg. 'r*'or a cell array of MATLAB LineSpec options.
- Typically used after a call to plot_bg().

See also

Navigation.plot_bg

Navigation.query

Find a path from start to goal using plan

`N.query(START, OPTIONS)` animates the robot moving from `START` (2×1) to the goal (which is a property of the object) using a previously computed plan.

`X = N.query(START, OPTIONS)` returns the path ($M \times 2$) from `START` to the goal (which is a property of the object).

The method performs the following steps:

- Initialize navigation, invoke method N.navigate_init()
- Visualize the environment, invoke method N.plot()
- Iterate on the next() method of the subclass until the goal is achieved.

Options

'animate'	Show the computed path as a series of green dots.

Notes

- If `START` given as [] then the user is prompted to click a point on the map.

See also

Navigation.navigate_init, Navigation.plot, Navigation.goal

Navigation.rand

Uniformly distributed random number

`R = N.rand()` return a uniformly distributed random number from a private random number stream.

`R = N.rand(M)` as above but return a matrix (`M` × `M`) of random numbers.

`R = N.rand(L,M)` as above but return a matrix (`L` × `M`) of random numbers.

Notes

- Accepts the same arguments as rand().
- Seed is provided to Navigation constructor.
- Provides an independent sequence of random numbers that does not interfere with any other randomised algorithms that might be used.

See also

Navigation.randi, Navigation.randn, rand, RandStream

Navigation.randi

Integer random number

`I = N.randi(RM)` returns a uniformly distributed random integer in the range 1 to `RM` from a private random number stream.

`I = N.randi(RM, M)` as above but returns a matrix (`M` $\times$ `M`) of random integers.

`I = N.randn(RM, L,M)` as above but returns a matrix (`L` $\times$ `M`) of random integers.

Notes

- Accepts the same arguments as randi().
- Seed is provided to Navigation constructor.
- Provides an independent sequence of random numbers that does not interfere with any other randomised algorithms that might be used.

See also

Navigation.rand, Navigation.randn, randi, RandStream

Navigation.randn

Normally distributed random number

`R = N.randn()` returns a normally distributed random number from a private random number stream.

`R = N.randn(M)` as above but returns a matrix (`M` $\times$ `M`) of random numbers.

`R = N.randn(L,M)` as above but returns a matrix (L × M) of random numbers.

Notes

- Accepts the same arguments as randn().
- Seed is provided to Navigation constructor.
- Provides an independent sequence of random numbers that does not interfere with any other randomised algorithms that might be used.

See also

Navigation.rand, Navigation.randi, randn, RandStream

Navigation.spinner

Update progress spinner

`N.spinner()` displays a simple ASCII progress spinner, a rotating bar.

Navigation.verbosity

Set verbosity

`N.verbosity(V)` set verbosity to `V`, where 0 is silent and greater values display more information.

ParticleFilter

Particle filter class

`Monte-carlo` based localisation for estimating vehicle pose based on odometry and observations of known landmarks.

Methods

run	run the particle filter
plot_xy	display estimated vehicle path
plot_pdf	display particle distribution

Properties

robot	reference to the robot object
sensor	reference to the sensor object
history each time step	vector of structs that hold the detailed information from
nparticles	number of particles used
x	particle states; nparticles x 3
weight	particle weights; nparticles x 1
x_est	mean of the particle population
std	standard deviation of the particle population
Q	covariance of noise added to state at each step
L	covariance of likelihood model
w0	offset in likelihood model
dim	maximum xy dimension

Example

Create a landmark map

map = PointMap(20);

and a vehicle with odometry covariance and a driver

W = diag([0.1, 1*π/180].^{2}); veh = Vehicle(W); veh.add_driver(RandomPath(10));

and create a range bearing sensor

R = diag([0.005, 0.5*π/180].^{2}); sensor = RangeBearingSensor(veh, map, R);

For the particle filter we need to define two covariance matrices. The first is is the covariance of the random noise added to the particle states at each iteration to represent uncertainty in configuration.

`Q = diag([0.1, 0.1, 1*pi/180]).`^{2};

and the covariance of the likelihood function applied to innovation

`L = diag([0.1 0.1]);`

Now construct the particle filter

pf = ParticleFilter(veh, sensor, `Q`, `L`, 1000);

which is configured with 1000 particles. The particles are initially uniformly distributed over the 3-dimensional configuration space.

We run the simulation for 1000 time steps `pf.run`(1000);

then plot the map and the true vehicle path

`map.plot()`; veh.plot_xy('b');

and overlay the mean of the particle cloud

`pf.plot_xy('r');`

We can plot the standard deviation against time

plot(`pf`.std(1:100,:))

The particles are a sampled approximation to the PDF and we can display this as

`pf.plot_pdf()`

Acknowledgement

Based on code by Paul Newman, Oxford University, `http://www.robots.ox.ac.uk/` pnewman

Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

See also

Vehicle, RandomPath, RangeBearingSensor, PointMap, EKF

ParticleFilter.ParticleFilter

Particle filter constructor

`PF = ParticleFilter(VEHICLE, SENSOR, Q, L, NP, OPTIONS)` is a particle filter that estimates the state of the `VEHICLE` with a landmark sensor `SENSOR`. `Q` is the covariance of the noise added to the particles at each step (diffusion), `L` is the covariance used in the sensor likelihood model, and `NP` is the number of particles.

Options

'verbose'	Be verbose.
'private'	Use private random number stream.
'reset'	Reset random number stream.
'seed',S be a proper random number generator state such as saved in	Set the initial state of the random numb the seed0 property of an earlier run.

'nohistory'	Don't save history.
'x0'	Initial particle states ($N \times 3$)

Notes

- ParticleFilter subclasses Handle, so it is a reference object.
- If initial particle states not given they are set to a uniform distribution over the map, essentially the kidnapped robot problem
- which is quite unrealistic.
- Initial particle weights are always set to unity.
- The 'private'option creates a private random number stream for the methods rand, randn and randi. If not given the global stream is used.

See also

Vehicle, Sensor, RangeBearingSensor, PointMap

ParticleFilter.char

Convert to string

`PF.char()` is a string representing the state of the **ParticleFilter** object in human-readable form.

See also

ParticleFilter.display

ParticleFilter.display

Display status of particle filter object

`PF.display()` displays the state of the **ParticleFilter** object in human-readable form.

Notes

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

See also

ParticleFilter.char

ParticleFilter.init

Initialize the particle filter

`PF.init()` initializes the particle distribution and clears the history.

Notes

- If initial particle states were given to the constructor the states are set to this value, else a random distribution over the map is used.
- Invoked by the run() method.

ParticleFilter.plot_pdf

Plot particles as a PDF

`PF.plot_pdf()` plots a sparse PDF as a series of vertical line segments of height equal to particle weight.

ParticleFilter.plot_xy

Plot vehicle position

`PF.plot_xy()` plots the estimated vehicle path in the xy-plane.

`PF.plot_xy(LS)` as above but the optional line style arguments `LS` are passed to plot.

ParticleFilter.run

Run the particle filter

`PF.run(N, OPTIONS)` runs the filter for `N` time steps.

Options

'noplot'	Do not show animation.
'movie',M	Create an animation movie file M

Notes

- All previously estimated states and estimation history is cleared.

---

plot_vehicle

Draw mobile robot pose

`PLOT_VEHICLE(X,OPTIONS)` draws an oriented triangle to represent the pose of a mobile robot moving in a planar world. The pose `X` (1×3) = [x,y,theta]. If `X` is a matrix $(N \times 3)$ then an animation of the robot motion is shown and animated at the specified frame rate.

Image mode

Create a structure with the following elements and pass it with the 'model'option.

image	an RGB image $(H \times W \times 3)$
alpha	an alpha plane $(H \times W)$ with pixels 0 if transparent
rotation	image rotation in degrees required for front to pointing to the right
centre	image coordinate (U,V) of the centre of the back axle
length	length of the car in real-world units

Animation mode

`H = PLOT_VEHICLE(X,OPTIONS)` as above draws the robot and returns a handle.

PLOT_VEHICLE(X, 'handle', H) updates the pose X (1×3) of the previously drawn robot.

Options

'scale',S 1/60)	draw vehicle with length S x max
'size',S	draw vehicle with length S
'fillcolor',F	the color of the circle's interior, M
'alpha',A	transparency of the filled circle: (
'box'	draw a box shape (default is trian
'fps',F	animate at F frames per second (c
'image',I	use an image to represent the robo
'retain'	when X ($N \times 3$) then retain the ro
'model',M elements: image, alpha, rotation (deg), centre (pix), length (m).	animate an image of the vehicle.
'axis',h	handle of axis or UIAxis to draw
'movie',M	create a movie file in file M

Example

```
[car.image,~,car.alpha] = imread('car2.png');  % image and alpha layer
car.rotation = 180;  % image rotation to align front with world x-axis
car.centre = [648; 173]; % image coordinates of centre of the back wheels
car.length = 4.2; % real world length for scaling (guess)
h = plot_vehicle(x, 'model', car)  % draw car at configuration x
plot_vehicle(x, 'handle', h)  % animate car to configuration x
```

Notes

- The vehicle is drawn relative to the size of the axes, so set them first using axis().
- For backward compatibility, 'fill', is a synonym for 'fillcolor'
- For the 'model'option, you provide a monochrome or color image of the vehicle. Optionally you can provide an alpha mask (0=transparent).
- Specify the reference point on the vehicle as the (x,y) pixel
- coordinate within the image. Specify the rotation, in degrees, so that
- the car's front points to the right. Finally specify a length of the
- car, the image is scaled to be that length in the plot.
- Set 'fps'to Inf to have zero pause

See also Vehicle.plot, Animate, plot_poly, demos/car_animation

plotbotopt

Define default options for robot plotting

A user provided function that returns a cell array of default plot options for the SerialLink.plot method.

See also

SerialLink.plot

PoseGraph

Pose graph

PoseGraph.PoseGraph

the file data

we assume g2o format

`VERTEX* vertex`_id X Y THETA EDGE* startvertex_id endvertex_id X Y THETA IXX IXY IYY IXT IYT ITT

`vertex` numbers start at 0

PoseGraph.linear_factors

the ids of the vertices connected by the kth edge

id_i=eids(1,k); id_j=eids(2,k);

extract the poses of the vertices and the mean of the edge

v_i=vmeans(:,id_i); v_j=vmeans(:,id_j); z_ij=emeans(:,k);

Prismatic

Robot manipulator prismatic link class

A subclass of the Link class for a prismatic joint defined using standard Denavit-Hartenberg parameters: holds all information related to a robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

Constructors

Prismatic	construct a prismatic joint+link using standard DH

Information/display methods

display	print the link parameters in human readable form
dyn	display link dynamic parameters
type	joint type: 'R'or 'P'

Conversion methods

char	convert to string

Operation methods

A	link transform matrix
friction	friction force
nofriction	Link object with friction parameters set to zero%

Testing methods

islimit	test if joint exceeds soft limit
isrevolute	test if joint is revolute
isprismatic	test if joint is prismatic
issym	test if joint+link has symbolic parameters

Overloaded operators

+	concatenate links, result is a SerialLink object

Properties (read/write)

theta	kinematic: joint angle
d	kinematic: link offset
a	kinematic: link length
alpha	kinematic: link twist
jointtype	kinematic: 'R'if revolute, 'P'if prismatic
mdh	kinematic: 0 if standard D&H, else 1
offset	kinematic: joint variable offset
qlim	kinematic: joint variable limits [min max]
m	dynamic: link mass
r	dynamic: link COG wrt link coordinate frame 3×1
I	dynamic: link inertia matrix, symmetric 3×3, about link COG.
B	dynamic: link viscous friction (motor referred)
Tc	dynamic: link Coulomb friction
G	actuator: gear ratio
Jm	actuator: motor inertia (motor referred)

Notes

- Methods inherited from the Link superclass.
- This is reference class object
- Link class objects can be used in vectors and arrays

References

- Robotics, Vision & Control, P. Corke, Springer 2011, Chap 7.

See also

Link, Revolute, SerialLink

Prismatic.Prismatic

Create prismatic robot link object

`L = Prismatic(OPTIONS)` is a prismatic link object with the kinematic and dynamic parameters specified by the key/value pairs using the standard Denavit-Hartenberg conventions.

Options

'theta',TH	joint angle
'a',A	joint offset (default 0)
'alpha',A	joint twist (default 0)
'standard'	defined using standard D&H parameters (default).
'modified'	defined using modified D&H parameters.
'offset',O	joint variable offset (default 0)
'qlim',L	joint limit (default [])
'I',I	link inertia matrix (3×1, 6×1 or 3×3)
'r',R	link centre of gravity (3×1)
'm',M	link mass (1×1)
'G',G	motor gear ratio (default 1)
'B',B	joint friction, motor referenced (default 0)
'Jm',J	motor inertia, motor referenced (default 0)
'Tc',T	Coulomb friction, motor referenced (1×1 or 2×1), (default [0 0])
'sym'	consider all parameter values as symbolic not numeric

Notes

- The joint extension, d, is provided as an argument to the A() method.
- The link inertia matrix (3×3) is symmetric and can be specified by giving a 3×3 matrix, the diagonal elements [Ixx Iyy Izz], or the moments and products
- of inertia [Ixx Iyy Izz Ixy Iyz Ixz].
- All friction quantities are referenced to the motor not the load.
- Gear ratio is used only to convert motor referenced quantities such as friction and interia to the link frame.

See also

Link, Prismatic, RevoluteMDH

PrismaticMDH

Robot manipulator prismatic link class for MDH convention

A subclass of the Link class for a prismatic joint defined using modified Denavit-Hartenberg parameters: holds all information related to a robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

Constructors

PrismaticMDH	construct a prismatic joint+link using modified DH

Information/display methods

display	print the link parameters in human readable form
dyn	display link dynamic parameters
type	joint type: 'R'or 'P'

Conversion methods

char	convert to string

Operation methods

A	link transform matrix
friction	friction force
nofriction	Link object with friction parameters set to zero%

Testing methods

islimit	test if joint exceeds soft limit
isrevolute	test if joint is revolute
isprismatic	test if joint is prismatic
issym	test if joint+link has symbolic parameters

Overloaded operators

+	concatenate links, result is a SerialLink object

Properties (read/write)

theta	kinematic: joint angle
d	kinematic: link offset
a	kinematic: link length
alpha	kinematic: link twist
jointtype	kinematic: 'R'if revolute, 'P'if prismatic
mdh	kinematic: 0 if standard D&H, else 1
offset	kinematic: joint variable offset

qlim	kinematic: joint variable limits [min max]
m	dynamic: link mass
r	dynamic: link COG wrt link coordinate frame 3×1
I	dynamic: link inertia matrix, symmetric 3×3, about link COG.
B	dynamic: link viscous friction (motor referred)
Tc	dynamic: link Coulomb friction
G	actuator: gear ratio
Jm	actuator: motor inertia (motor referred)

Notes

- Methods inherited from the Link superclass.
- This is reference class object
- Link class objects can be used in vectors and arrays
- Modified Denavit-Hartenberg parameters are used

References

- Robotics, Vision & Control, P. Corke, Springer 2011, Chap 7.

See also

Link, Prismatic, RevoluteMDH, SerialLink

PrismaticMDH.PrismaticMDH

Create prismatic robot link object using MDH notaton

`L = PrismaticMDH(OPTIONS)` is a prismatic link object with the kinematic and dynamic parameters specified by the key/value pairs using the modified Denavit-Hartenberg conventions.

Options

'theta',TH	joint angle
'a',A	joint offset (default 0)
'alpha',A	joint twist (default 0)
'standard'	defined using standard D&H parameters (default).
'modified'	defined using modified D&H parameters.

'offset',O	joint variable offset (default 0)
'qlim',L	joint limit (default [])
'I',I	link inertia matrix (3×1, 6×1 or 3×3)
'r',R	link centre of gravity (3×1)
'm',M	link mass (1×1)
'G',G	motor gear ratio (default 1)
'B',B	joint friction, motor referenced (default 0)
'Jm',J	motor inertia, motor referenced (default 0)
'Tc',T	Coulomb friction, motor referenced (1×1 or 2×1), (default [0 0])
'sym'	consider all parameter values as symbolic not numeric

Notes

- The joint extension, d, is provided as an argument to the A() method.
- The link inertia matrix (3×3) is symmetric and can be specified by giving a 3×3 matrix, the diagonal elements [Ixx Iyy Izz], or the moments and products
- of inertia [Ixx Iyy Izz Ixy Iyz Ixz].
- All friction quantities are referenced to the motor not the load.
- Gear ratio is used only to convert motor referenced quantities such as friction and interia to the link frame.

See also

Link, Prismatic, RevoluteMDH

PRM

Probabilistic RoadMap navigation class

A concrete subclass of the abstract Navigation class that implements the probabilistic roadmap navigation algorithm over an occupancy grid. This performs goal independent planning of roadmaps, and at the query stage finds paths between specific start and goal points.

Methods

PRM	Constructor
plan	Compute the roadmap
query	Find a path
plot	Display the obstacle map

display	Display the parameters in human readable form
char	Convert to string

Example

```
load map1               % load map
goal = [50,30];         % goal point
start = [20, 10];       % start point
prm = PRM(map);         % create navigation object
prm.plan()              % create roadmaps
prm.query(start, goal)  % animate path from this start location
```

References

- Probabilistic roadmaps for path planning in high dimensional configuration spaces, L. Kavraki, P. Svestka, J. Latombe, and M. Overmars,
- IEEE Transactions on Robotics and Automation, vol. 12, pp. 566-580, Aug 1996.
- Robotics, Vision & Control, Section 5.2.4, P. Corke, Springer 2011.

See also

Navigation, DXform, Dstar, PGraph

PRM.PRM

Create a PRM navigation object

`P = PRM(MAP, options)` is a probabilistic roadmap navigation object, and `MAP` is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied).

Options

'npoints',N	Number of sample points (default 100)
'distthresh',D than D (default 0.3 max(size(occgrid)))	Distance threshold, edges only connect vertices closer

Other `options` are supported by the Navigation superclass.

See also

Navigation.Navigation

PRM.char

Convert to string

`P.char()` is a string representing the state of the **PRM** object in human-readable form.

See also

PRM.display

PRM.plan

Create a probabilistic roadmap

`P.plan(OPTIONS)` creates the probabilistic roadmap by randomly sampling the free space in the map and building a graph with edges connecting close points. The resulting graph is kept within the object.

Options

'npoints',N	Number of sample points (default is set by constructor)
'distthresh',D than D (default set by constructor)	Distance threshold, edges only connect vertices closer
'movie',M	make a movie of the PRM planning

PRM.plot

Visualize navigation environment

`P.plot()` displays the roadmap and the occupancy grid.

Options

'goal'	Superimpose the goal position if set
'nooverlay'	Don't overlay the PRM graph

Notes

- If a query has been made then the path will be shown.
- Goal and start locations are kept within the object.

PRM.query

Find a path between two points

`P.query(START, GOAL)` finds a path ($M \times 2$) from `START` to `GOAL`.

purepursuit

Find pure pursuit goal

`P = PUREPURSUIT(CP, R, PATH)` is the current pursuit point (2×1) for a robot at location `CP` (2×1) following a `PATH` ($N \times 2$). The pursuit point is the closest point along the path that is a distance $>=$ `R` from the current point `CP`.

Reference

- A review of some pure-pursuit based tracking techniques for control of autonomous vehicle, Samuel etal., Int. J. Computer Applications, Feb 2016
- Steering Control of an Autonomous Ground Vehicle with Application to the DARPA Urban Challenge, Stefan F. Campbell, Masters thesis, MIT, 2007.

See also

Navigation

qplot

Plot robot joint angles

`QPLOT(Q)` is a convenience function to plot joint angle trajectories ($M \times 6$) for a 6-axis robot, where each row represents one time step.

The first three joints are shown as solid lines, the last three joints (wrist) are shown as dashed lines. A legend is also displayed.

`QPLOT(T, Q)` as above but displays the joint angle trajectory versus time given the time vector `T` ($M \times 1$).

See also

jtraj, plotp, plot

RandomPath

Vehicle driver class

Create a “driver” object capable of steering a Vehicle subclass object through random waypoints within a rectangular region and at constant speed.

The driver object is connected to a Vehicle object by the latter's add_driver() method. The driver's demand() method is invoked on every call to the Vehicle's step() method.

Methods

init	reset the random number generator
demand	speed and steer angle to next waypoint
display	display the state and parameters in human readable form
char	convert to string

plot

Properties

goal	current goal/waypoint coordinate
veh	the Vehicle object being controlled
dim	dimensions of the work space (2×1) [m]

speed	speed of travel [m/s]
dthresh	proximity to waypoint at which next is chosen [m]

Example

`veh = Bicycle(V);` `veh`.add_driver(RandomPath(20, 2));

Notes

- It is possible in some cases for the vehicle to move outside the desired region, for instance if moving to a waypoint near the edge, the limited
- turning circle may cause the vehicle to temporarily move outside.
- The vehicle chooses a new waypoint when it is closer than property closeenough to the current waypoint.
- Uses its own random number stream so as to not influence the performance of other randomized algorithms such as path planning.

Reference

Robotics, Vision & Control, Chap 6, Peter Corke, Springer 2011

See also

Vehicle, Bicycle, Unicycle

RandomPath.RandomPath

Create a driver object

`D = RandomPath(D, OPTIONS)` returns a "driver" object capable of driving a Vehicle subclass object through random waypoints. The waypoints are positioned inside a rectangular region of dimension `D` interpreted as:

- `D` scalar; X: -`D` to +`D`, Y: -`D` to +`D`
- `D` (1×2); X: -`D`(1) to +`D`(1), Y: -`D`(2) to +`D`(2)
- `D` (1×4); X: `D`(1) to `D`(2), Y: `D`(3) to `D`(4)

Options

'speed',S	Speed along path (default 1m/s).
'dthresh',D	Distance from goal at which next goal is chosen.

See also

Vehicle

RandomPath.char

Convert to string

`s = R.char()` is a string showing driver parameters and state in in a compact human readable format.

RandomPath.demand

Compute speed and heading to waypoint

`[SPEED,STEER] = R.demand()` is the speed and steer angle to drive the vehicle toward the next waypoint. When the vehicle is within R.dtresh a new waypoint is chosen.

See also

Vehicle

RandomPath.display

Display driver parameters and state

`R.display()` displays driver parameters and state in compact human readable form.

Notes

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

See also

RandomPath.char

RandomPath.init

Reset random number generator

`R.init()` resets the random number generator used to create the waypoints. This enables the sequence of random waypoints to be repeated.

Notes

- Called by Vehicle.run.

See also

randstream

RangeBearingSensor

Range and bearing sensor class

A concrete subclass of the Sensor class that implements a range and bearing angle sensor that provides robot-centric measurements of landmark points in the world. To enable this it holds a references to a map of the world (LandmarkMap object) and a robot (Vehicle subclass object) that moves in $SE(2)$.

The sensor observes landmarks within its angular field of view between the minimum and maximum range.

Methods

reading	range/bearing observation of random landmark
h	range/bearing observation of specific landmark
Hx	Jacobian matrix with respect to vehicle pose dh/dx
Hp	Jacobian matrix with respect to landmark position dh/dp
Hw	Jacobian matrix with respect to noise dh/dw
g	feature position given vehicle pose and observation
Gx	Jacobian matrix with respect to vehicle pose dg/dx
Gz	Jacobian matrix with respect to observation dg/dz

Properties (read/write)

W	measurement covariance matrix (2×2)
interval	valid measurements returned every interval'th call to reading()

landmarklog time history of observed landmarks

Reference

Robotics, Vision & Control, Chap 6, Peter Corke, Springer 2011

See also

Sensor, Vehicle, LandmarkMap, EKF

RangeBearingSensor.RangeBearingSensor

Range and bearing sensor constructor

`S = RangeBearingSensor(VEHICLE, MAP, OPTIONS)` is an object representing a range and bearing angle sensor mounted on the Vehicle subclass object `VEHICLE` and observing an environment of known landmarks represented by the LandmarkMap object `MAP`. The sensor covariance is W (2×2) representing range and bearing covariance.

The sensor has specified angular field of view and minimum and maximum range.

Options

'covar',W	covariance matrix (2×2)
'range',xmax	maximum range of sensor
'range',[xmin xmax]	minimum and maximum range of sensor
'angle',TH	angular field of view, from -TH to +TH
'angle',[THMIN THMAX] and THMAX	detection for angles betwen THMIN
'skip',K	return a valid reading on every K'th call
'fail',[TMIN TMAX] timesteps TMIN and TMAX	sensor simulates failure between
'animate'	animate sensor readings

See also

options for Sensor constructor

See also

RangeBearingSensor.reading, Sensor.Sensor, Vehicle, LandmarkMap, EKF

RangeBearingSensor.g

Compute landmark location

`P = S.g(X, Z)` is the world coordinate (2×1) of a feature given the observation `Z` (1×2) from a vehicle state with `X` (3×1).

See also

RangeBearingSensor.Gx, RangeBearingSensor.Gz

RangeBearingSensor.Gx

Jacobian dg/dx

`J = S.Gx(X, Z)` is the Jacobian dg/dx (2×3) at the vehicle state `X` (3×1) for sensor observation `Z` (2×1).

See also

RangeBearingSensor.g

RangeBearingSensor.Gz

Jacobian dg/dz

`J = S.Gz(X, Z)` is the Jacobian dg/dz (2×2) at the vehicle state `X` (3×1) for sensor observation `Z` (2×1).

See also

RangeBearingSensor.g

RangeBearingSensor.h

Landmark range and bearing

`Z = S.h(X, K)` is a sensor observation (1×2), range and bearing, from vehicle at pose `X` (1×3) to the `K`'th landmark.

`Z = S.h(X, P)` as above but compute range and bearing to a landmark at coordinate `P`.

`Z = s.h(X)` as above but computes range and bearing to all map features. `Z` has one row per landmark.

Notes

- Noise with covariance W (propertyW) is added to each row of `Z`.
- Supports vectorized operation where XV ($N \times 3$) and `Z` ($N \times 2$).
- The landmark is assumed visible, field of view and range liits are not applied.

See also

RangeBearingSensor.reading, RangeBearingSensor.Hx, RangeBearingSensor.Hw, RangeBearingSensor.Hp

---

RangeBearingSensor.Hp

Jacobian dh/dp

`J = S.Hp(X, K)` is the Jacobian dh/dp (2×2) at the vehicle state `X` (3×1) for map landmark `K`.

`J = S.Hp(X, P)` as above but for a landmark at coordinate `P` (1×2).

See also

RangeBearingSensor.h

RangeBearingSensor.Hw

Jacobian dh/dw

`J = S.Hw(X, K)` is the Jacobian dh/dw (2×2) at the vehicle state `X` (3×1) for map landmark `K`.

See also

RangeBearingSensor.h

RangeBearingSensor.Hx

Jacobian dh/dx

`J = S.Hx(X, K)` returns the Jacobian dh/dx (2×3) at the vehicle state `X` (3×1) for map landmark `K`.

`J = S.Hx(X, P)` as above but for a landmark at coordinate `P`.

See also

RangeBearingSensor.h

RangeBearingSensor.reading

Choose landmark and return observation

`[Z,K] = S.reading()` is an observation of a random visible landmark where `Z`=[R,THETA] is the range and bearing with additive Gaussian noise of covariance W (property W). `K` is the index of the map feature that was observed.

The landmark is chosen randomly from the set of all visible landmarks, those within the angular field of view and range limits. If no valid measurement, ie. no features within range, interval subsampling enabled or simulated failure the return is `Z`=[] and `K`=0.

Notes

- Noise with covariance W (property W) is added to each row of `Z`.
- If 'animate'option set then show a line from the vehicle to the landmark

- If 'animate'option set and the angular and distance limits are set then display that region as a shaded polygon.
- Implements sensor failure and subsampling if specified to constructor.

See also

RangeBearingSensor.h

ReedsShepp

Shepp path planner sample code

based on python code from Python Robotics by Atsushi Sakai(@Atsushi_twi)

Peter 3/18

Finds the shortest path between 2 configurations:

- robot can move forward or backward
- the robot turns at zero or maximum curvature
- there are discontinuities in velocity and steering commands (cusps)

to see what it does run

>> ReedsShepp.test

References

- Reeds, J. A.; Shepp, L. A. Optimal paths for a car that goes both forwards and backwards.
- Pacific J. Math. 145 (1990), no. 2, 367–393.
- `https://projecteuclid.org/euclid.pjm/1102645450`

ReedsShepp.generate_path

a list of all possible words

Revolute

Robot manipulator Revolute link class

A subclass of the Link class for a revolute joint defined using standard Denavit-Hartenberg parameters: holds all information related to a revolute robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

Constructors

Revolute	construct a revolute joint+link using standard DH

Information/display methods

display	print the link parameters in human readable form
dyn	display link dynamic parameters
type	joint type: 'R'or 'P'

Conversion methods

char	convert to string

Operation methods

A	link transform matrix
friction	friction force
nofriction	Link object with friction parameters set to zero%

Testing methods

islimit	test if joint exceeds soft limit
isrevolute	test if joint is revolute
isprismatic	test if joint is prismatic
issym	test if joint+link has symbolic parameters

Overloaded operators

+	concatenate links, result is a SerialLink object

Properties (read/write)

theta	kinematic: joint angle
d	kinematic: link offset
a	kinematic: link length
alpha	kinematic: link twist
jointtype	kinematic: 'R'if revolute, 'P'if prismatic
mdh	kinematic: 0 if standard D&H, else 1
offset	kinematic: joint variable offset
qlim	kinematic: joint variable limits [min max]
m	dynamic: link mass
r	dynamic: link COG wrt link coordinate frame 3×1
I	dynamic: link inertia matrix, symmetric 3×3, about link COG.
B	dynamic: link viscous friction (motor referred)
Tc	dynamic: link Coulomb friction
G	actuator: gear ratio
Jm	actuator: motor inertia (motor referred)

Notes

- Methods inherited from the Link superclass.
- This is reference class object
- Link class objects can be used in vectors and arrays

References

- Robotics, Vision & Control, P. Corke, Springer 2011, Chap 7.

See also

Link, Prismatic, RevoluteMDH, SerialLink

Revolute.Revolute

Create revolute robot link object

`L = Revolute(OPTIONS)` is a revolute link object with the kinematic and dynamic parameters specified by the key/value pairs using the standard Denavit-Hartenberg conventions.

Options

'd',D	joint extension
'a',A	joint offset (default 0)
'alpha',A	joint twist (default 0)
'standard'	defined using standard D&H parameters (default).
'modified'	defined using modified D&H parameters.
'offset',O	joint variable offset (default 0)
'qlim',L	joint limit (default [])
'I',I	link inertia matrix (3×1, 6×1 or 3×3)
'r',R	link centre of gravity (3×1)
'm',M	link mass (1×1)
'G',G	motor gear ratio (default 1)
'B',B	joint friction, motor referenced (default 0)
'Jm',J	motor inertia, motor referenced (default 0)
'Tc',T	Coulomb friction, motor referenced (1×1 or 2×1), (default [0 0])
'sym'	consider all parameter values as symbolic not numeric

Notes

- The joint angle, theta, is provided as an argument to the A() method.
- The link inertia matrix (3×3) is symmetric and can be specified by giving a 3×3 matrix, the diagonal elements [Ixx Iyy Izz], or the moments and products
- of inertia [Ixx Iyy Izz Ixy Iyz Ixz].
- All friction quantities are referenced to the motor not the load.
- Gear ratio is used only to convert motor referenced quantities such as friction and interia to the link frame.

See also

Link, Prismatic, RevoluteMDH

RevoluteMDH

Robot manipulator Revolute link class for MDH convention

A subclass of the Link class for a revolute joint defined using modified Denavit-Hartenberg parameters: holds all information related to a revolute robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

Constructors

RevoluteMDH	construct a revolute joint+link using modified DH

Information/display methods

display	print the link parameters in human readable form
dyn	display link dynamic parameters
type	joint type: 'R'or 'P'

Conversion methods

char	convert to string

Operation methods

A	link transform matrix
friction	friction force
nofriction	Link object with friction parameters set to zero%

Testing methods

islimit	test if joint exceeds soft limit
isrevolute	test if joint is revolute
isprismatic	test if joint is prismatic
issym	test if joint+link has symbolic parameters

Overloaded operators

+	concatenate links, result is a SerialLink object

Properties (read/write)

theta	kinematic: joint angle
d	kinematic: link offset
a	kinematic: link length
alpha	kinematic: link twist
jointtype	kinematic: 'R'if revolute, 'P'if prismatic
mdh	kinematic: 0 if standard D&H, else 1
offset	kinematic: joint variable offset

qlim	kinematic: joint variable limits [min max]
m	dynamic: link mass
r	dynamic: link COG wrt link coordinate frame 3×1
I	dynamic: link inertia matrix, symmetric 3×3, about link COG.
B	dynamic: link viscous friction (motor referred)
Tc	dynamic: link Coulomb friction
G	actuator: gear ratio
Jm	actuator: motor inertia (motor referred)

Notes

- Methods inherited from the Link superclass.
- This is reference class object
- Link class objects can be used in vectors and arrays
- Modified Denavit-Hartenberg parameters are used

References

- Robotics, Vision & Control, P. Corke, Springer 2011, Chap 7.

See also

Link, PrismaticMDH, Revolute, SerialLink

RevoluteMDH.RevoluteMDH

Create revolute robot link object using MDH notation

`L = RevoluteMDH(OPTIONS)` is a revolute link object with the kinematic and dynamic parameters specified by the key/value pairs using the modified Denavit-Hartenberg conventions.

Options

'd',D	joint extension
'a',A	joint offset (default 0)
'alpha',A	joint twist (default 0)
'standard'	defined using standard D&H parameters (default).
'modified'	defined using modified D&H parameters.

'offset',O	joint variable offset (default 0)
'qlim',L	joint limit (default [])
'I',I	link inertia matrix (3×1, 6×1 or 3×3)
'r',R	link centre of gravity (3×1)
'm',M	link mass (1×1)
'G',G	motor gear ratio (default 1)
'B',B	joint friction, motor referenced (default 0)
'Jm',J	motor inertia, motor referenced (default 0)
'Tc',T	Coulomb friction, motor referenced (1×1 or 2×1), (default [0 0])
'sym'	consider all parameter values as symbolic not numeric

Notes

- The joint angle, theta, is provided as an argument to the A() method.
- The link inertia matrix (3×3) is symmetric and can be specified by giving a 3×3 matrix, the diagonal elements [Ixx Iyy Izz], or the moments and products
- of inertia [Ixx Iyy Izz Ixy Iyz Ixz].
- All friction quantities are referenced to the motor not the load.
- Gear ratio is used only to convert motor referenced quantities such as friction and interia to the link frame.

See also

Link, Prismatic, RevoluteMDH

RobotArm

Serial-link robot arm class

A subclass of SerialLink than includes an interface to a physical robot.

Methods

plot	display graphical representation of robot
teach	drive the physical and graphical robots
mirror	use the robot as a slave to drive graphics
jmove	joint space motion of the physical robot
cmove	Cartesian space motion of the physical robot

plus all other methods of SerialLink

Properties

as per SerialLink class

Note

- the interface to a physical robot, the machine, should be an abstract

superclass but right now it isn't

- RobotArm is a subclass of SerialLink.
- RobotArm is a reference (handle subclass) object.
- RobotArm objects can be used in vectors and arrays

Reference

- `http://www.petercorke.com/doc/robotarm.pdf`
- Robotics, Vision & Control, Chaps 7-9, P. Corke, Springer 2011.
- Robot, Modeling & Control, M.Spong, S. Hutchinson & M. Vidyasagar, Wiley 2006.

See also

Machine, SerialLink, Link, DHFactor

RobotArm.RobotArm

Construct a RobotArm object

`RA = RobotArm(L, M, OPTIONS)` is a robot object defined by a vector of Link objects `L` with a physical robot interface `M` represented by an object of class Machine.

Options

'name', name	set robot name property
'comment', comment	set robot comment property
'manufacturer', manuf	set robot manufacturer property
'base', base	set base transformation matrix property
'tool', tool	set tool transformation matrix property

'gravity', g	set gravity vector property
'plotopt', po	set plotting options property

See also

SerialLink.SerialLink, Arbotix.Arbotix

RobotArm.cmove

Cartesian space move

`RA.cmove(T)` moves the robot arm to the pose specified by the homogeneous transformation (4×4).

Notes

- A joint-space trajectory is computed from the current configuration to QD using the jmove() method.
- If the robot is 6-axis with a spherical wrist inverse kinematics are computed using ikine6s() otherwise numerically using ikine().

See also

RobotArm.jmove, Arbotix.setpath

RobotArm.delete

Destroy the RobotArm object

`RA.delete()` closes and destroys the machine interface object and the **RobotArm** object.

RobotArm.getq

Get the robot joint angles

`Q = RA.getq()` is a vector ($1 \times N$) of robot joint angles.

Notes

- If the robot has a gripper, its value is not included in this vector.

RobotArm.gripper

Control the robot gripper

`RA.gripper(C)` sets the robot gripper according to `C` which is 0 for closed and 1 for open.

Notes

- Not all robots have a gripper.
- The gripper is assumed to be the last servo motor in the chain.

RobotArm.jmove

Joint space move

`RA.jmove(QD)` moves the robot arm to the configuration specified by the joint angle vector `QD` ($1 \times N$).

`RA.jmove(QD, T)` as above but the total move takes `T` seconds.

Notes

- A joint-space trajectory is computed from the current configuration to `QD`.

See also

RobotArm.cmove, Arbotix.setpath

RobotArm.mirror

Mirror the robot pose to graphics

`RA.mirror()` places the robot arm in relaxed mode, and as it is moved by hand the graphical animation follows.

See also

SerialLink.teach, SerialLink.plot

RobotArm.teach

Teach the robot

`RA.teach()` invokes a simple GUI to allow joint space motion, as well as showing an animation of the robot on screen.

See also

SerialLink.teach, SerialLink.plot

RRT

Class for rapidly-exploring random tree navigation

A concrete subclass of the abstract Navigation class that implements the rapidly exploring random tree (RRT) algorithm. This is a kinodynamic planner that takes into account the motion constraints of the vehicle.

Methods

RRT	Constructor
plan	Compute the tree
query	Compute a path
plot	Display the tree
display	Display the parameters in human readable form
char	Convert to string

Properties (read only)

graph	A PGraph object describign the tree

Example


```
goal = [0,0,0];
start = [0,2,0];
veh = Bicycle('steermax', 1.2);
rrt = RRT(veh, 'goal', goal, 'range', 5);
rrt.plan()              % create navigation tree
rrt.query(start, goal)  % animate path from this start location
```

References

- Randomized kinodynamic planning, S. LaValle and J. Kuffner,
- International Journal of Robotics Research vol. 20, pp. 378-400, May 2001.
- Probabilistic roadmaps for path planning in high dimensional configuration spaces, L. Kavraki, P. Svestka, J. Latombe, and M. Overmars,
- IEEE Transactions on Robotics and Automation, vol. 12, pp. 566-580, Aug 1996.
- Robotics, Vision & Control, Section 5.2.5, P. Corke, Springer 2011.

See also

Navigation, PRM, DXform, Dstar, PGraph

RRT.RRT

Create an RRT navigation object

`R = RRT.RRT(VEH, OPTIONS)` is a rapidly exploring tree navigation object for a vehicle kinematic model given by a Vehicle subclass object `VEH`.

`R = RRT.RRT(VEH, MAP, OPTIONS)` as above but for a region with obstacles defined by the occupancy grid `MAP`.

Options

'npoints',N	Number of nodes in the tree (default 500)
'simtime',T random point (default 0.5s)	Interval over which to simulate kinematic model toward
'goal',P	Goal position (1×2) or pose (1×3) in workspace
'speed',S	Speed of vehicle [m/s] (default 1)
'root',R	Configuration of tree root (3×1) (default [0,0,0])
'revcost',C	Cost penalty for going backwards (default 1)
'range',R	Specify rectangular bounds of robot's workspace:

- `R` scalar; X: -`R` to +`R`, Y: -`R` to +`R`

- R (1×2); X: -R(1) to +R(1), Y: -R(2) to +R(2)
- R (1×4); X: R(1) to R(2), Y: R(3) to R(4)

Other options are provided by the Navigation superclass.

Notes

- 'range'option is ignored if an occupacy grid is provided.

Reference

- Robotics, Vision & Control Peter Corke, Springer 2011. p102.

See also

Vehicle, Bicycle, Unicycle

RRT.char

Convert to string

`R.char()` is a string representing the state of the **RRT** object in human-readable form.

RRT.plan

Create a rapidly exploring tree

`R.plan(OPTIONS)` creates the tree roadmap by driving the vehicle model toward random goal points. The resulting graph is kept within the object.

Options

'goal',P	Goal pose (1×3)
'ntrials',N	Number of path trials (default 50)
'noprogress'	Don't show the progress bar
'samples'	Show progress in a plot of the workspace

- '.'for each random point x_rand
- 'o'for the nearest point which is added to the tree

- red line for the best path

Notes

- At each iteration we need to find a vehicle path/control that moves it from a random point towards a point on the graph. We sample ntrials of
- random steer angles and velocities and choose the one that gets us
- closest (computationally slow, since each path has to be integrated
- over time).

RRT.plot

Visualize navigation environment

`R.plot()` displays the navigation tree in 3D, where the vertical axis is vehicle heading angle. If an occupancy grid was provided this is also displayed.

RRT.query

Find a path between two points

`X = R.path(START, GOAL)` finds a path ($N \times 3$) from pose `START` (1×3) to pose `GOAL` (1×3). The pose is expressed as [X,Y,THETA].

`R.path(START, GOAL)` as above but plots the path in 3D, where the vertical axis is vehicle heading angle. The nodes are shown as circles and the line segments are blue for forward motion and red for backward motion.

Notes

- The path starts at the vertex closest to the `START` state, and ends at the vertex closest to the `GOAL` state. If the tree is sparse this
- might be a poor approximation to the desired start and end.

See also

RRT.plot

rtbdemo

Robot toolbox demonstrations

rtbdemo displays a menu of toolbox demonstration scripts that illustrate:

- fundamental datatypes
  - rotation and homogeneous transformation matrices
  - quaternions
  - trajectories
- serial link manipulator arms
  - forward and inverse kinematics
  - robot animation
  - forward and inverse dynamics
- mobile robots
  - kinematic models and control
  - path planning (D*, PRM, Lattice, RRT)
  - localization (EKF, particle filter)
  - SLAM (EKF, pose graph)
  - quadrotor control

`rtbdemo(T)` as above but waits for `T` seconds after every statement, no need to push the enter key periodically.

Notes

- By default the scripts require the user to periodically hit <Enter> in order to move through the explanation.
- Some demos require Simulink
- To quit, close the rtbdemo window

RTBPlot

Plot utilities for Robotics Toolbox

RTBPlot.box

Draw a box

BPX(AX, R, EXTENT, COLOR, OFFSET, OPTIONS) draws a cylinder parallel to axis AX ('x', 'y'or 'z') of side length R between EXTENT(1) and EXTENT(2).

RTBPlot.cyl

Draw a cylinder

CYL(AX, R, EXTENT, COLOR, OFFSET, OPTIONS) draws a cylinder parallel to axis AX ('x', 'y'or 'z') of radius R between EXTENT(1) and EXTENT(2).

OPTIONS are passed through to surf.

See also

surf, RTBPlot.box

Sensor

Sensor superclass

An abstract superclass to represent robot navigation sensors.

Methods

plot	plot a line from robot to map feature
display	print the parameters in human readable form
char	convert to string

Properties

robot	The Vehicle object on which the sensor is mounted
map	The PointMap object representing the landmarks around the robot

Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

See also

RangeBearingSensor, EKF, Vehicle, LandmarkMap

Sensor.Sensor

Sensor object constructor

`S = Sensor(VEHICLE, MAP, OPTIONS)` is a sensor mounted on a vehicle described by the Vehicle subclass object `VEHICLE` and observing landmarks in a map described by the LandmarkMap class object `MAP`.

Options

'animate'	animate the action of the laser scanner
'ls',LS	laser scan lines drawn with style ls (default 'r-')
'skip', I	return a valid reading on every I'th call
'fail',T	sensor simulates failure between timesteps T=[TMIN,TMAX]

Notes

- Animation shows a ray from the vehicle position to the selected landmark.

Sensor.char

Convert sensor parameters to a string

`s = S.char()` is a string showing sensor parameters in a compact human readable format.

Sensor.display

Display status of sensor object

`S.display()` displays the state of the sensor object in human-readable form.

Notes

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

See also

Sensor.char

Sensor.plot

Plot sensor reading

`S.plot(J)` draws a line from the robot to the `J`'th map feature.

Notes

- The line is drawn using the linestyle given by the property ls
- There is a delay given by the property delay

simulinkext

Return file extension of Simulink block diagrams.

`str = simulinkext()` is either

- '.mdl'if Simulink version number is less than 8
- '.slx'if Simulink version numberis larger or equal to 8

Notes

The file extension for Simulink block diagrams has changed from Matlab 2011b to Matlab 2012a. This function is used for backwards compatibility.

Author

Joern Malzahn, (joern.malzahn@tu-dortmund.de)

See also

symexpr2slblock, doesblockexist, distributeblocks

---

startup_rtb

Initialize MATLAB paths for Robotics Toolbox

Adds demos, data, contributed code and examples to the MATLAB path, and adds also to Java class path.

Notes

- This sets the paths for the current session only.
- To make the settings persistent across sessions you can:
  - Add this script to your MATLAB startup.m script.
  - After running this script run PATHTOOL and save the path.

See also

path, addpath, pathtool, javaaddpath

---

sym2

Subclass of sym class

This is ugly. The provided sym class can only generate MATLAB functions, not expressions. It can generate expressions in C and Fortran however.

The only way to access this capability is direct to the MuPad engine, and since we can't change the sym class we use a subclass and add a matgen method

sym2.matgen

MATLAB representation of a symbolic expression.

`MATGEN(S)` is a fragment of MATLAB that evaluates symbolic expression `S`.

See also

sym/pretty, sym/latex, sym/ccode

Based on sym.fortran().

---

symexpr2slblock

Create symbolic embedded MATLAB Function block

`symexpr2slblock(VARARGIN)` creates an Embedded MATLAB Function block from a symbolic expression. The input arguments are just as used with the functions emlBlock or matlabFunctionBlock.

Notes

- In Symbolic Toolbox versions prior to V5.7 (2011b) the function to create Embedded Matlab Function blocks from symbolic expressions is
- 'emlBlock'.
- Since V5.7 (2011b) there is another function named 'matlabFunctionBlock'which replaces the old function.
- symexpr2slblock is a wrapper around both functions, which checks for the installed Symbolic Toolbox version and calls the
- required function accordingly.

Authors

Joern Malzahn, (joern.malzahn@tu-dortmund.de)

See also

emlBlock, matlabFunctionBlock

test_jacob_dot

harness for jacob_dot

tpoly

Generate scalar polynomial trajectory

`[S,SD,SDD] = TPOLY(S0, SF, M)` is a scalar trajectory ($M \times 1$) that varies smoothly from `S0` to `SF` in `M` steps using a quintic (5th order) polynomial. Velocity and acceleration can be optionally returned as `SD` ($M \times 1$) and `SDD` ($M \times 1$) respectively.

`TPOLY(S0, SF, M)` as above but plots `S`, `SD` and `SDD` versus time in a single figure.

`[S,SD,SDD] = TPOLY(S0, SF, T)` as above but the trajectory is computed at each point in the time vector `T` ($M \times 1$).

`[S,SD,SDD] = TPOLY(S0, SF, T, QD0, QD1)` as above but also specifies the initial and final velocity of the trajectory.

Notes

- If `M` is given
  - Velocity is in units of distance per trajectory step, not per second.
  - Acceleration is in units of distance per trajectory step squared, not per second squared.
- If `T` is given then results are scaled to units of time.
- The time vector `T` is assumed to be monotonically increasing, and time scaling is based on the first and last element.

Reference:

Robotics, Vision & Control Chap 3 Springer 2011

See also

lspb, jtraj

Unicycle

vehicle class

This concrete class models the kinematics of a differential steer vehicle (unicycle model) on a plane. For given steering and velocity inputs it updates the true vehicle state and returns noise-corrupted odometry readings.

Methods

init	initialize vehicle state
f	predict next state based on odometry
step	move one time step and return noisy odometry
control	generate the control inputs for the vehicle
update	update the vehicle state
run	run for multiple time steps
Fx	Jacobian of f wrt x
Fv	Jacobian of f wrt odometry noise
gstep	like step() but displays vehicle
plot	plot/animate vehicle on current figure
plot_xy	plot the true path of the vehicle
add_driver	attach a driver object to this vehicle
display	display state/parameters in human readable form
char	convert to string

Class methods

plotv	plot/animate a pose on current figure

Properties (read/write)

x	true vehicle state: x, y, theta (3×1)
V	odometry covariance (2×2)
odometry	distance moved in the last interval (2×1)

rdim	dimension of the robot (for drawing)

L	length of the vehicle (wheelbase)
alphalim	steering wheel limit
maxspeed	maximum vehicle speed
T	sample interval
verbose	verbosity
x_hist	history of true vehicle state ($N \times 3$)
driver	reference to the driver object
x0	initial state, restored on init()

Examples

Odometry covariance (per timstep) is

V = diag([0.02, 0.5*π/180].^{2});

Create a vehicle with this noisy odometry

v = Unicycle('covar', diag([0.1 0.01].^{2}));

and display its initial state

v

now apply a speed (0.2m/s) and steer angle (0.1rad) for 1 time step

`odo = v.step`(0.2, 0.1)

where `odo` is the noisy odometry estimate, and the new true vehicle state

`v`

We can add a driver object

`v.add`_driver(RandomPath(10))

which will move the vehicle within the region -10<x<10, -10<y<10 which we can see by

`v.run`(1000)

which shows an animation of the vehicle moving for 1000 time steps between randomly selected wayoints.

Notes

- Subclasses the MATLAB handle class which means that pass by reference semantics apply.

Reference

Robotics, Vision & Control, Chap 6 Peter Corke, Springer 2011

See also

RandomPath, EKF

Unicycle.Unicycle

Unicycle object constructor

`V = Unicycle(VA, OPTIONS)` creates a **Unicycle** object with actual odometry covariance `VA` (2×2) matrix corresponding to the odometry vector [dx dtheta].

Options

'W',W	Wheel separation [m] (default 1)

'vmax',S	Maximum speed (default 5m/s)
'x0',x0	Initial state (default (0,0,0))
'dt',T	Time interval
'rdim',R	Robot size as fraction of plot window (default 0.2)
'verbose'	Be verbose

Notes

- Subclasses the MATLAB handle class which means that pass by reference semantics apply.

Unicycle.char

Convert to a string

`s = V.char()` is a string showing vehicle parameters and state in a compact human readable format.

See also

Unicycle.display

Unicycle.deriv

be called from a continuous time integrator such as ode45 or Simulink

---

Unicycle.f

Predict next state based on odometry

`XN = V.f(X, ODO)` is the predicted next state `XN` (1×3) based on current state `X` (1×3) and odometry `ODO` (1×2) = [distance, heading_change].

`XN = V.f(X, ODO, W)` as above but with odometry noise `W`.

Notes

- Supports vectorized operation where `X` and `XN` ($N \times 3$).

Unicycle.Fv

Jacobian df/dv

`J = V.Fv(X, ODO)` is the Jacobian df/dv (3×2) at the state `X`, for odometry input `ODO` (1×2) = [distance, heading_change].

See also

Unicycle.F, Vehicle.Fx

Unicycle.Fx

Jacobian df/dx

`J = V.Fx(X, ODO)` is the Jacobian df/dx (3×3) at the state `X`, for odometry input `ODO` (1×2) = [distance, heading_change].

See also

Unicycle.f, Vehicle.Fv

Unicycle.update

Update the vehicle state

`ODO = V.update(U)` is the true odometry value for motion with `U`=[speed,steer].

Notes

- Appends new state to state history property x_hist.
- Odometry is also saved as property odometry.

Vehicle

Abstract vehicle class

This abstract class models the kinematics of a mobile robot moving on a plane and with a pose in $SE(2)$. For given steering and velocity inputs it updates the true vehicle state and returns noise-corrupted odometry readings.

Methods

Vehicle	constructor
add_driver	attach a driver object to this vehicle
control	generate the control inputs for the vehicle
f	predict next state based on odometry
init	initialize vehicle state
run	run for multiple time steps
run2	run with control inputs
step	move one time step and return noisy odometry
update	update the vehicle state

Plotting/display methods

char	convert to string
display	display state/parameters in human readable form

plot	plot/animate vehicle on current figure
plot_xy	plot the true path of the vehicle
Vehicle.plotv	plot/animate a pose on current figure

Properties (read/write)

x	true vehicle state: x, y, theta (3×1)
V	odometry covariance (2×2)
odometry	distance moved in the last interval (2×1)

rdim	dimension of the robot (for drawing)

L	length of the vehicle (wheelbase)
alphalim	steering wheel limit
speedmax	maximum vehicle speed
T	sample interval
verbose	verbosity
x_hist	history of true vehicle state ($N \times 3$)
driver	reference to the driver object
x0	initial state, restored on init()

Examples

If veh is an instance of a Vehicle class then we can add a driver object

`veh.add`_driver(RandomPath(10))

which will move the vehicle within the region -10<x<10, -10<y<10 which we can see by

`veh.run`(1000)

which shows an animation of the vehicle moving for 1000 time steps between randomly selected wayoints.

Notes

- Subclass of the MATLAB handle class which means that pass by reference semantics apply.

Reference

Robotics, Vision & Control, Chap 6 Peter Corke, Springer 2011

See also

Bicycle, Unicycle, RandomPath, EKF

Vehicle.Vehicle

Vehicle object constructor

`V = Vehicle(OPTIONS)` creates a **Vehicle** object that implements the kinematic model of a wheeled vehicle.

Options

'covar',C	specify odometry covariance (2×2) (default 0)
'speedmax',S	Maximum speed (default 1m/s)
'L',L	Wheel base (default 1m)
'x0',x0	Initial state (default (0,0,0))
'dt',T	Time interval (default 0.1)
'rdim',R	Robot size as fraction of plot window (default 0.2)
'verbose'	Be verbose

Notes

- The covariance is used by a “hidden” random number generator within the class.
- Subclasses the MATLAB handle class which means that pass by reference semantics apply.

Vehicle.add_driver

Add a driver for the vehicle

`V.add_driver(D)` connects a driver object `D` to the vehicle. The driver object has one public method:

```
[speed, steer] = D.demand();
```

that returns a speed and steer angle.

Notes

- The Vehicle.step() method invokes the driver if one is attached.

See also

Vehicle.step, RandomPath

Vehicle.char

Convert to string

`s = V.char()` is a string showing vehicle parameters and state in a compact human readable format.

See also

Vehicle.display

Vehicle.control

Compute the control input to vehicle

`U = V.control(SPEED, STEER)` is a control input (1×2) = [speed,steer] based on provided controls `SPEED,STEER` to which speed and steering angle limits have been applied.

`U = V.control()` as above but demand originates with a "driver" object if one is attached, the driver's DEMAND() method is invoked. If no driver is attached then speed and steer angle are assumed to be zero.

See also

Vehicle.step, RandomPath

Vehicle.display

Display vehicle parameters and state

`V.display()` displays vehicle parameters and state in compact human readable form.

Notes

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

See also

Vehicle.char

Vehicle.init

Reset state

`V.init()` sets the state V.x := V.x0, initializes the driver object (if attached) and clears the history.

`V.init(X0)` as above but the state is initialized to `X0`.

Vehicle.path

Compute path for constant inputs

`XF = V.path(TF, U)` is the final state of the vehicle (3×1) from the initial state (0,0,0) with the control inputs `U` (vehicle specific). `TF` is a scalar to specify the total integration time.

`XP = V.path(TV, U)` is the trajectory of the vehicle ($N \times 3$) from the initial state (0,0,0) with the control inputs `U` (vehicle specific). T is a vector (N) of times for which elements of the trajectory will be computed.

`XP = V.path(T, U, X0)` as above but specify the initial state.

Notes

- Integration is performed using ODE45.
- The ODE being integrated is given by the deriv method of the vehicle object.

See also

ode45

Vehicle.plot

Plot vehicle

The vehicle is depicted graphically as a narrow triangle that travels "point first" and has a length V.rdim.

`V.plot(OPTIONS)` plots the vehicle on the current axes at a pose given by the current robot state. If the vehicle has been previously plotted its pose is updated.

`V.plot(X, OPTIONS)` as above but the robot pose is given by `X` (1×3).

`H = V.plotv(X, OPTIONS)` draws a representation of a ground robot as an oriented triangle with pose `X` (1×3) [x,y,theta]. `H` is a graphics handle.

`V.plotv(H, X)` as above but updates the pose of the graphic represented by the handle `H` to pose `X`.

Options

'scale',S	Draw vehicle with length S x maximum axis dimension
'size',S	Draw vehicle with length S
'color',C	Color of vehicle.
'fill'	Filled
'trail',S	Trail with line style S, use line() name-value pairs

Example

```
veh.plot('trail', {'Color', 'r', 'Marker', 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r', '
```

Vehicle.plot_xy

Plots true path followed by vehicle

`V.plot_xy()` plots the true xy-plane path followed by the vehicle.

`V.plot_xy(LS)` as above but the line style arguments `LS` are passed to plot.

Notes

- The path is extracted from the x_hist property.

Vehicle.plotv

Plot ground vehicle pose

`H = Vehicle.plotv(X, OPTIONS)` draws a representation of a ground robot as an oriented triangle with pose `X` (1×3) [x,y,theta]. `H` is a graphics handle. If `X` ($N \times 3$) is a matrix it is considered to represent a trajectory in which case the vehicle graphic is animated.

`Vehicle.plotv(H, X)` as above but updates the pose of the graphic represented by the handle `H` to pose `X`.

Options

'scale',S	Draw vehicle with length S x maximum axis dimension
'size',S	Draw vehicle with length S
'fillcolor',C	Color of vehicle.
'fps',F	Frames per second in animation mode (default 10)

Example

Generate some path $3 \times N$

```
p = PRM.plan(start, goal);
```

Set the axis dimensions to stop them rescaling for every point on the path

```
axis([-5 5 -5 5]);
```

Now invoke the static method

```
Vehicle.plotv(p);
```

Notes

- This is a class method.

See also

Vehicle.plot

Vehicle.run

Run the vehicle simulation

`V.run(N)` runs the vehicle model for `N` timesteps and plots the vehicle pose at each step.

`P = V.run(N)` runs the vehicle simulation for `N` timesteps and return the state history (`N` $\times$ 3) without plotting. Each row is (x,y,theta).

See also

Vehicle.step, Vehicle.run2

Vehicle.run2

Run the vehicle simulation with control inputs

`P = V.run2(T, X0, SPEED, STEER)` runs the vehicle model for a time `T` with speed `SPEED` and steering angle `STEER`. `P` ($N \times 3$) is the path followed and each row is (x,y,theta).

Notes

- Faster and more specific version of run() method.
- Used by the RRT planner.

See also

Vehicle.run, Vehicle.step, RRT

Vehicle.step

Advance one timestep

`ODO = V.step(SPEED, STEER)` updates the vehicle state for one timestep of motion at specified `SPEED` and `STEER` angle, and returns noisy odometry.

`ODO = V.step()` updates the vehicle state for one timestep of motion and returns noisy odometry. If a "driver" is attached then its DEMAND() method is invoked to compute speed and steer angle. If no driver is attached then speed and steer angle are assumed to be zero.

Notes

- Noise covariance is the property V.

See also

Vehicle.control, Vehicle.update, Vehicle.add_driver

Vehicle.update

Update the vehicle state

`ODO = V.update(U)` is the true odometry value for motion with `U`=[speed,steer].

Notes

- Appends new state to state history property x_hist.
- Odometry is also saved as property odometry.

Vehicle.verbosity

Set verbosity

`V.verbosity(A)` set verbosity to `A`. `A`=0 means silent.

---

wtrans

Transform a wrench between coordinate frames

`WT = WTRANS(T, W)` is a wrench (6×1) in the frame represented by the homogeneous transform `T` (4×4) corresponding to the world frame wrench `W` (6×1).

The wrenches `W` and `WT` are 6-vectors of the form [Fx Fy Fz Mx My Mz]'.

See also

tr2delta, tr2jac

Bibliography

