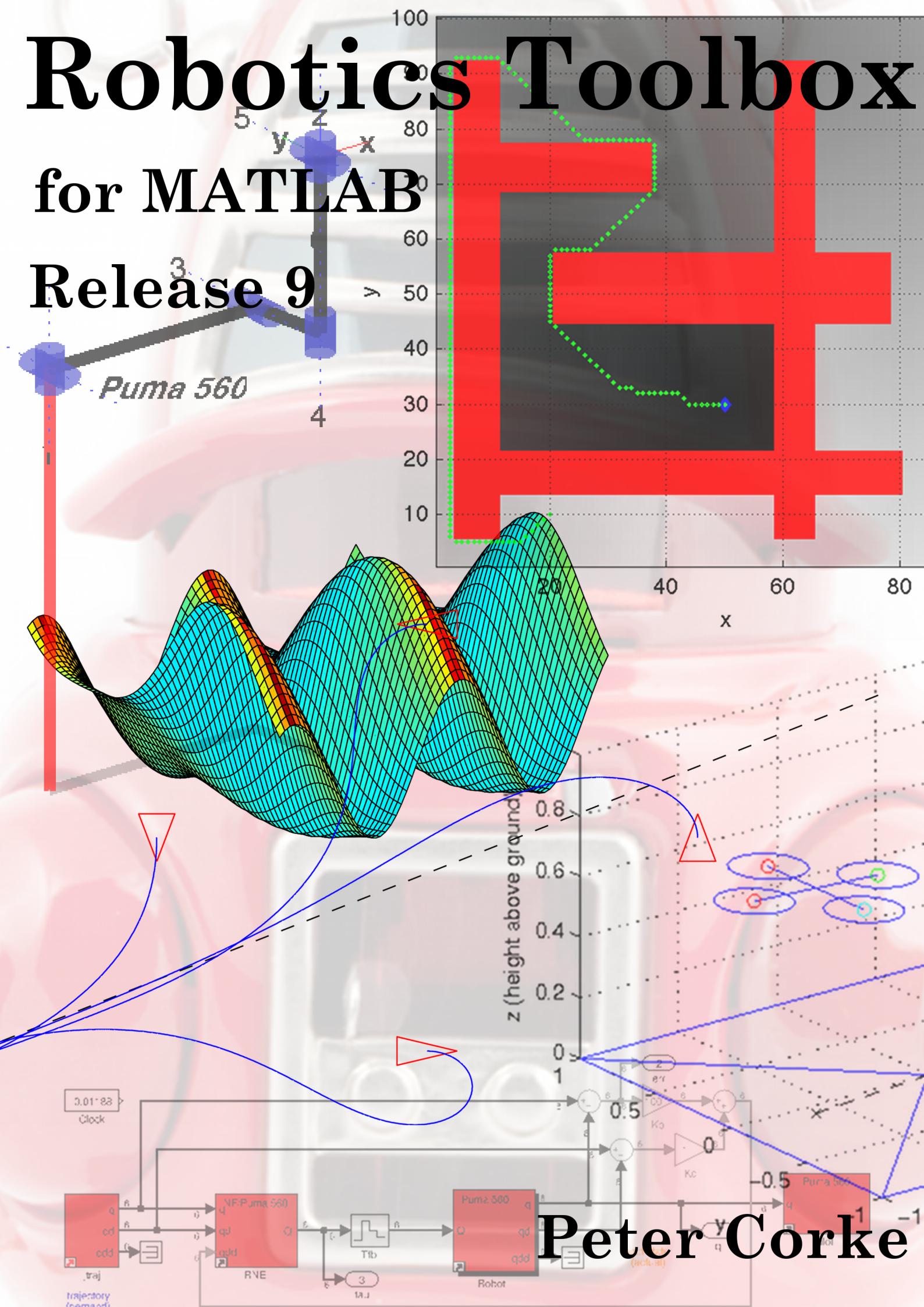


# Robotics Toolbox for MATLAB

Release 9



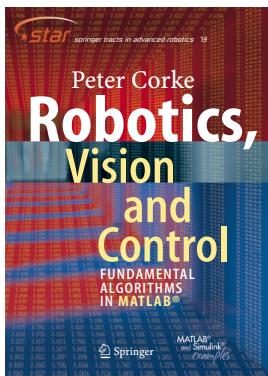
Peter Corke

Release 9.10  
Release date January 2015

Licence LGPL  
Toolbox home page <http://www.petercorke.com/robot>  
Discussion group <http://groups.google.com.au/group/robotics-tool-box>



# Preface



This, the ninth major release of the Toolbox, represents twenty years of development and a substantial level of maturity. This version captures a large number of changes and extensions generated over the last two years which support my new book “*Robotics, Vision & Control*” shown to the left.

The Toolbox has always provided many functions that are useful for the study and simulation of classical arm-type robotics, for example such things as kinematics, dynamics, and trajectory generation. The Toolbox is based on a very general method of representing the kinematics and dynamics of serial-link manipulators.

These parameters are encapsulated in MATLAB<sup>®</sup> objects — robot objects can be created by the user for any serial-link manipulator and a number of examples are provided for well known robots such as the Puma 560 and the Stanford arm amongst others. The Toolbox also provides functions for manipulating and converting between datatypes such as vectors, homogeneous transformations and unit-quaternions which are necessary to represent 3-dimensional position and orientation.

This ninth release of the Toolbox has been significantly extended to support mobile robots. For ground robots the Toolbox includes standard path planning algorithms (bug, distance transform, D\*, PRM), kinodynamic planning (RRT), localization (EKF, particle filter), map building (EKF) and simultaneous localization and mapping (EKF), and a Simulink model of a non-holonomic vehicle. The Toolbox also includes a detailed Simulink model for a quadrotor flying robot.

The routines are generally 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<sup>®</sup> compiler, or create a MEX version.

This manual is now essentially auto-generated from the comments in the MATLAB<sup>®</sup> code itself which reduces the effort in maintaining code and a separate manual as I used to — the downside is that there are no worked examples and figures in the manual. However the book “*Robotics, Vision & Control*” provides a detailed discussion (600 pages, nearly 400 figures and 1000 code examples) of how to use the Toolbox functions to solve many types of problems in robotics.

# Contents

Preface . . . . .	4
Functions by category . . . . .	10
<b>1 Introduction</b>	<b>13</b>
1.1 What's changed . . . . .	13
1.1.1 New features and changes to RTB 9.10 . . . . .	13
1.1.2 Earlier changes to RTB 9 . . . . .	14
1.2 Migrating from RTB 8 and earlier . . . . .	16
1.2.1 New functions . . . . .	17
1.2.2 General improvements . . . . .	18
1.3 How to obtain the Toolbox . . . . .	18
1.3.1 Documentation . . . . .	19
1.4 MATLAB version issues . . . . .	19
1.5 Use in teaching . . . . .	19
1.6 Use in research . . . . .	19
1.7 Support . . . . .	20
1.8 Related software . . . . .	20
1.8.1 Octave . . . . .	20
1.8.2 Python version . . . . .	21
1.8.3 Machine Vision toolbox . . . . .	21
1.9 Contributing to the Toolboxes . . . . .	21
1.10 Acknowledgements . . . . .	21
<b>2 Functions and classes</b>	<b>22</b>
about . . . . .	22
angdiff . . . . .	22
angvec2r . . . . .	23
angvec2tr . . . . .	23
Animate . . . . .	24
Arbotix . . . . .	25
bresenham . . . . .	33
Bug2 . . . . .	34
ccodefunctionstring . . . . .	35
circle . . . . .	36
CodeGenerator . . . . .	37
colnorm . . . . .	68
colorname . . . . .	68
ctraj . . . . .	69

delta2tr . . . . .	69
DHFactor . . . . .	70
diff2 . . . . .	71
distanceform . . . . .	71
distributeblocks . . . . .	72
dockfigs . . . . .	73
doesblockexist . . . . .	73
Dstar . . . . .	73
DXform . . . . .	78
e2h . . . . .	81
edgelist . . . . .	81
EKF . . . . .	82
eul2jac . . . . .	90
eul2r . . . . .	91
eul2tr . . . . .	92
gauss2d . . . . .	92
h2e . . . . .	93
homline . . . . .	93
homtrans . . . . .	93
ishomog . . . . .	94
ishomog2 . . . . .	94
isrot . . . . .	95
isrot2 . . . . .	95
isvec . . . . .	96
joy2tr . . . . .	96
joystick . . . . .	97
jsingu . . . . .	98
jtraj . . . . .	98
Link . . . . .	99
lspb . . . . .	108
makemap . . . . .	108
Map . . . . .	109
mdl_3link3d . . . . .	112
mdl_ball . . . . .	112
mdl_baxter . . . . .	113
mdl_coil . . . . .	114
mdl_Fanuc10L . . . . .	114
mdl_hyper2d . . . . .	115
mdl_hyper3d . . . . .	116
mdl_irb140 . . . . .	116
mdl_irb140_mdh . . . . .	117
mdl_jaco . . . . .	118
mdl_KR5 . . . . .	119
mdl_m16 . . . . .	119
mdl_mico . . . . .	120
mdl_MotomanHP6 . . . . .	121
mdl_nao . . . . .	121
mdl_offset3 . . . . .	122
mdl_offset6 . . . . .	123
mdl_onelink . . . . .	123

mdl_p8 . . . . .	124
mdl_phantomx . . . . .	125
mdl_planar1 . . . . .	125
mdl_planar2 . . . . .	126
mdl_planar3 . . . . .	126
mdl_puma560 . . . . .	127
mdl_puma560akb . . . . .	128
mdl_S4ABB2p8 . . . . .	128
mdl_simple6 . . . . .	129
mdl_stanford . . . . .	129
mdl_stanford_mdh . . . . .	130
mdl_twolink . . . . .	131
mdl_twolink_mdh . . . . .	132
mstraj . . . . .	132
mtraj . . . . .	134
multidprintf . . . . .	134
Navigation . . . . .	135
numcols . . . . .	141
numrows . . . . .	142
oa2r . . . . .	142
oa2tr . . . . .	143
ParticleFilter . . . . .	143
peak . . . . .	148
peak2 . . . . .	149
PGraph . . . . .	150
plot2 . . . . .	164
plot_arrow . . . . .	165
plot_box . . . . .	165
plot_circle . . . . .	166
plot_ellipse . . . . .	167
plot_ellipse_inv . . . . .	167
plot_homline . . . . .	168
plot_point . . . . .	169
plot_poly . . . . .	170
plot_sphere . . . . .	170
plot_vehicle . . . . .	171
plotbotopt . . . . .	171
plotp . . . . .	172
polydiff . . . . .	172
Polygon . . . . .	172
Prismatic . . . . .	178
PrismaticMDH . . . . .	178
PRM . . . . .	179
qplot . . . . .	181
Quaternion . . . . .	182
r2t . . . . .	191
randinit . . . . .	192
RandomPath . . . . .	192
RangeBearingSensor . . . . .	195
Revolute . . . . .	199

RevoluteMDH . . . . .	200
RobotArm . . . . .	200
rot2 . . . . .	204
rotx . . . . .	204
roty . . . . .	205
rotz . . . . .	205
rpy2jac . . . . .	205
rpy2r . . . . .	206
rpy2tr . . . . .	207
RRT . . . . .	207
rt2tr . . . . .	211
rtbdemo . . . . .	211
runscript . . . . .	212
rvcpath . . . . .	213
se2 . . . . .	213
se3 . . . . .	214
Sensor . . . . .	214
SerialLink . . . . .	216
simulinkext . . . . .	255
skew . . . . .	256
startup_rtb . . . . .	256
symexpr2slblock . . . . .	256
t2r . . . . .	257
tb_optparse . . . . .	257
tpoly . . . . .	259
tr2angvec . . . . .	259
tr2delta . . . . .	260
tr2eul . . . . .	260
tr2jac . . . . .	261
tr2rpy . . . . .	261
tr2rt . . . . .	262
tranimate . . . . .	263
transl . . . . .	264
transl2 . . . . .	265
trchain . . . . .	265
trchain2 . . . . .	266
trinterp . . . . .	267
trnorm . . . . .	267
trot2 . . . . .	268
trotx . . . . .	268
troty . . . . .	269
trotz . . . . .	269
trplot . . . . .	270
trplot2 . . . . .	271
trprint . . . . .	272
trscale . . . . .	273
unit . . . . .	273
Vehicle . . . . .	274
vex . . . . .	282
VREP . . . . .	283

VREP_arm . . . . .	299
VREP_camera . . . . .	303
VREP_mirror . . . . .	308
VREP_obj . . . . .	311
wtrans . . . . .	315
xaxis . . . . .	315
xyzlabel . . . . .	316
yaxis . . . . .	316

# Functions by category

## 3D transforms

angvec2r . . . . .	23
angvec2tr . . . . .	23
eul2r . . . . .	91
eul2tr . . . . .	92
ishomog2 . . . . .	94
ishomog . . . . .	94
isrot2 . . . . .	95
isrot . . . . .	95
oa2r . . . . .	142
oa2tr . . . . .	143
r2t . . . . .	191
rotx . . . . .	204
roty . . . . .	205
rotz . . . . .	205
rpy2r . . . . .	206
rpy2tr . . . . .	207
rt2tr . . . . .	211
t2r . . . . .	257
tr2angvec . . . . .	259
tr2eul . . . . .	260
tr2rpy . . . . .	261
tr2rt . . . . .	262
tranimate . . . . .	263
transl2 . . . . .	265
transl . . . . .	264
trchain2 . . . . .	266
trchain . . . . .	265
trnorm . . . . .	267
trotx . . . . .	268
troty . . . . .	269
trotz . . . . .	269
trplot2 . . . . .	271
trplot . . . . .	270
trprint . . . . .	272
trscale . . . . .	273

## 2D transforms

ishomog2 . . . . .	94
isrot2 . . . . .	95
rot2 . . . . .	204
se2 . . . . .	213
se3 . . . . .	214
transl2 . . . . .	265
trchain2 . . . . .	266
trot2 . . . . .	268
trplot2 . . . . .	271

## Homogeneous points and lines

e2h . . . . .	81
h2e . . . . .	93
homline . . . . .	93
homtrans . . . . .	93
plot_homline . . . . .	168

## Differential motion

delta2tr . . . . .	69
eul2jac . . . . .	90
rpy2jac . . . . .	205
skew . . . . .	256
tr2delta . . . . .	260
tr2jac . . . . .	261
vex . . . . .	282
wtrans . . . . .	315

## Trajectory generation

ctraj . . . . .	69
jtraj . . . . .	98

lspb .....	108
mstraj .....	132
mtraj .....	134
tpoly .....	259
trinterp .....	267

**Quaternion**

Quaternion .....	182
------------------	-----

**Serial-link manipulator**

CodeGenerator .....	37
Link .....	99
PrismaticMDH .....	178
Prismatic .....	178
RevoluteMDH .....	200
Revolute .....	199
SerialLink .....	216

**Models****Kinematic**

DHFactor .....	70
jsingu .....	98

**Dynamics**

wtrans .....	315
--------------	-----

**Mobile robot**

Map .....	109
Navigation .....	135
RandomPath .....	192
RangeBearingSensor .....	195
Sensor .....	214
Vehicle .....	274
makemap .....	108

**Localization**

EKF .....	82
ParticleFilter .....	143

**Path planning**

Bug2 .....	34
DXform .....	78
Dstar .....	73
PRM .....	179
RRT .....	207

**Graphics**

Animate .....	24
plot2 .....	164
plotp .....	172
qplot .....	181
trplot2 .....	271
xaxis .....	315
xyzlabel .....	316
yaxis .....	316

**Utility**

PGraph .....	150
Polygon .....	172
about .....	22
angdiff .....	22
bresenham .....	33
circle .....	36
colnorm .....	68
colorname .....	68
diff2 .....	71
distancexfm .....	71
dockfigs .....	73
edgelist .....	81
gauss2d .....	92
isvec .....	96
multidfprintf .....	134
numcols .....	141
numrows .....	142
peak2 .....	149
peak .....	148
plot_circle .....	166
polydiff .....	172
randinit .....	192
runscript .....	212
rvcpath .....	213
unit .....	273

**Demonstrations**

rtbdemo ..... 211

joystick ..... 97

**Interfacing**

Arbotix ..... 25  
RobotArm ..... 200  
VREP\_arm ..... 299  
VREP\_camera ..... 303  
VREP\_mirror ..... 308  
VREP\_obj ..... 311  
VREP ..... 283  
joy2tr ..... 96

**Code generation**

ccodefunctionstring ..... 35  
distributeblocks ..... 72  
doesblockexist ..... 73  
simulinkext ..... 255  
symexpr2slblock ..... 256

**Examples**

plotbotopt ..... 171

# Chapter 1

## Introduction

### 1.1 What's changed

#### 1.1.1 New features and changes to RTB 9.10

Features of this dot point release include:

- A major pass over all the documentation working on clarity and consistency.
- `startup_rvc` now checks whether an update to RTB is available.
- Fixed the bug in `SerialLink.plot` for prismatic joints and modified DH parameters where the links were missing or in the wrong place.
- New methods for `SerialLink`
  - `edit` which presents all the parameters as a table in a figure window and allows interactive editing.
  - `fellipse` and `vellipse` which respectively plot the force and velocity ellipsoid of the robot.
  - `trchain` which describes forward kinematics as a minimal series of elementary transforms.
  - numerical inverse kinematic: `ikcon` and `ikunc` for constrained and unconstrained joint angles, based on the MATLAB Optimisation Toolbox. These provide a good alternative to the existing method `ikine`. Contributed by Bryan Moutrie.
  - `pay` and `paycap` to determine the effect of payload and maximum payload capability. Contributed by Bryan Moutrie.
  - `collisions` which interfaces to the public domain package pHRIWARE (by Bryan Moutrie) to perform collision checking between a robot arm and static and moving objects which are described by simple 3D shape primitives.

- A new function called `models` which lists all the robot models and their keywords. Allows searching by keywords. Becoming useful as the number of model files increases.
- Prototype models for Baxter, NAO and Kuka KR5 robots.
- New version of `rtbdemo` that uses a proper GUI.
- Update of the V-REP interface to support version 3.1.x and a demo created, see `rtbdemo`.
- An increasing number of functions now have a '`deg`' option which allows it to accept angle input in units of degrees rather than the default of radians.
- New Simulink blocks to support N-rotor flyers, eg. hexa- and octo-rotors. This includes a new control mixer block and a generalized N-rotor dynamics block. Graphics has been updated to render the appropriate number of rotors. The vehicle model structure must now include an element `nrotors` to specify the number of rotors.
- `robblocks`, the Toolbox Simulink block library is now a `.slx` file, rather than a `.mdl` file and all models have been updated to suit. Some older Simulink models had atrophied and have been updated.

### 1.1.2 Earlier changes to RTB 9

- A major rewrite of `CodeGenerator`
- A major rewrite of `ikine6s` to handle a number of specific cases: robot with no shoulder offset, robot with shoulder offset (can have left/right configuration), Stanford arm (prismatic third joint), Puma 560 arm. The previous code made lots of assumptions applicable to the Puma, which caused errors for other 6-axis robots with spherical wrists.
- Symbolic inverse kinematics (developmental) can be found for robots with 2, 3 or 6 DOF. See `SerialLink.ikine_sym`.

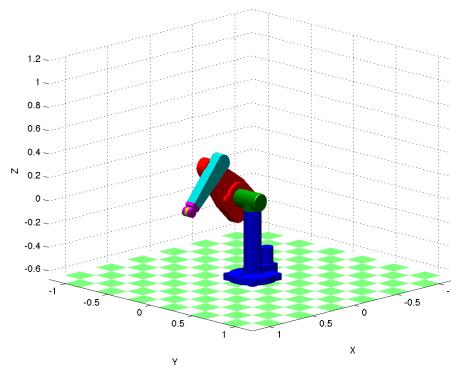


Figure 1.1: New rendered robot model using the `plot3d()` method.

- Aesthetic updates to `plot()` and `teach()` methods of the `SerialLink` object
- A new method `plot3d()` which uses STL format solid models to render realistic looking robots as shown in Figure 1.1. This requires STL models, such as those shipped with the package ARTE by Arturo Gil (<https://arvc.umh.es/arte>).
- Subclasses of `Link` called `Revolute`, `Prismatic`, `RevoluteMDH` and `PrismaticMDH` that can be used to make your code clearer and more concise.
- The behaviour of Fast RNE is a bit different. The MATLAB version `@SerialLink/rne.m` is always executed, and it makes the decision whether or not to invoke the MEX file. The MEX file is executed if:
  1. the robot is not symbolic, and
  2. the `SerialLink` property `fast` is true (ie. the ‘`nofast`’ option is not given), and
  3. the MEX file exists.
- A significant number of new robot models.
- A major renovation of `DHFactor` to bring it up to spec with the lastest version of Java.
- A ‘`flip`’ option added to `tr2eul`.
- A new method `A` for `SerialLink` object that computes a sequence of Link A matrices.
- A new method `trchain` for `SerialLink` object that emits the kinematic model as a sequence of elementary transforms.
- A new method `trchain` that expresses kinematics as a chain of elementary transforms.
- A bunch of functions with suffix 2 that deal with SE(2) and SO(2) transforms.
- An improved version of the demo `rtbdemo`, with more functions and an improved interface. It uses the common function `runscript` to step through the individual demo scripts. It works even better with `cprintf` from MATLAB Central.
- A set of classes (experimental) to interface with the V-REP robotics simulation engine by Coppelia Robotics. See `robot/interfaces/VREP`.
- Since 9.8 the Toolbox now contains the Robotic Symbolic Toolbox by Jörn Malzahn. There are additional functions, as well as symbolic support throughout the `SerialLink` class.
- Many bug fixes

## 1.2 Migrating from RTB 8 and earlier

If you're upgrading from RTB 8 or earlier note a significant number of changes summarised below.

- The command `startup_rvc` should be executed before using the Toolbox. This sets up the MATLAB search paths correctly.
- The Robot class is now named SerialLink to be more specific.
- Almost all functions that operate on a SerialLink object are now methods rather than functions, for example `plot()` or `fkine()`. In practice this makes little difference to the user but operations can now be expressed as `robot.plot(q)` or `plot(robot, q)`. Toolbox documentation now prefers the former convention which is more aligned with object-oriented practice.
- The parameters to the Link object constructor are now in the order: theta, d, a, alpha. Why this order? It's the order in which the link transform is created: RZ(theta) TZ(d) TX(a) RX(alpha).
- All robot models now begin with the prefix `mdl_`, so `puma560` is now `mdl_puma560`.
- The function `drivebot` is now the SerialLink method `teach`.
- The function `ikine560` is now the SerialLink method `ikine6s` to indicate that it works for any 6-axis robot with a spherical wrist.
- The link class is now named Link to adhere to the convention that all classes begin with a capital letter.
- The `robot` class is now called `SerialLink`. It is created from a vector of `Link` objects, not a cell array.
- The quaternion class is now named `Quaternion` to adhere to the convention that all classes begin with a capital letter.
- A number of utility functions have been moved into the a directory common since they are not robot specific.
- `skew` no longer accepts a skew symmetric matrix as an argument and returns a 3-vector, this functionality is provided by the new function `vex`.
- `tr2diff` and `diff2tr` are now called `tr2delta` and `delta2tr`
- `ctraj` with a scalar argument now spaces the points according to a trapezoidal velocity profile (see `lspb`). To obtain even spacing provide a uniformly spaced vector as the third argument, eg. `linspace(0, 1, N)`.
- The RPY functions `tr2rpy` and `rpy2tr` assume that the roll, pitch, yaw rotations are about the X, Y, Z axes which is consistent with common conventions for vehicles (planes, ships, ground vehicles). For some applications (eg. cameras) it useful to consider the rotations about the Z, Y, X axes, and this behaviour can be obtained by using the option '`'zyx'`' with these functions (note this is the pre release 8 behaviour).
- Many functions now accept MATLAB style arguments given as trailing strings, or string-value pairs. These are parsed by the internal function `tb_optparse`.

### 1.2.1 New functions

Release 9 introduces considerable new functionality, in particular for mobile robot control, navigation and localization:

- Mobile robotics:

**Vehicle** Model of a mobile robot that has the “bicycle” kinematic model (car-like). For given inputs it updates the robot state and returns noise corrupted odometry measurements. This can be used in conjunction with a “driver” class such as RandomPath which drives the vehicle between random waypoints within a specified rectangular region.

**Sensor**

**RangeBearingSensor** Model of a laser scanner RangeBearingSensor, subclass of Sensor, that works in conjunction with a Map object to return range and bearing to invariant point features in the environment.

**EKF** Extended Kalman filter EKF can be used to perform localization by dead reckoning or map features, map buildings and simultaneous localization and mapping.

**DXForm** Path planning classes: distance transform DXform, D\* lattice planner Dstar, probabilistic roadmap planner PRM, and rapidly exploring random tree RRT.

Monte Carlo estimator ParticleFilter.

- Arm robotics: jsingu, qplot, DHFactor (a simple means to generate the Denavit-Hartenberg kinematic model of a robot from a sequence of elementary transforms)

- Trajectory related: lspb, tpoly, mtraj, mstraj

- General transformation: homtrans, se2, se3, wtrans, vex (performs the inverse function to skew, it converts a skew-symmetric matrix to a 3-vector)

- Data structures:

Pgraph represents a non-directed embedded graph, supports plotting and minimum cost path finding.

Polygon a generic 2D polygon class that supports plotting, intersection/union/difference of polygons, line/polygon intersection, point/polygon containment.

- Graphical functions:

**trprint** compact display of a transform in various formats.

**trplot** display a coordinate frame in SE(3)

**trplot2** as above but for SE(2)

**tranimate** animate the motion of a coordinate frame

**plot\_box** plot a box given TL/BR corners or center+WH, with options for edge color, fill color and transparency.

**plot\_circle** plot one or more circles, with options for edge color, fill color and transparency.

**plot\_sphere** plot a sphere, with options for edge color, fill color and transparency.

**plot\_ellipse** plot an ellipse, with options for edge color, fill color and transparency.

**plot\_ellipsoid** plot an ellipsoid, with options for edge color, fill color and transparency.

**plot\_poly** plot a polygon, with options for edge color, fill color and transparency.

- Utility:

**about** display a one line summary of a matrix or class, a compact version of whos

**tb\_optparse** general argument handler and options parser, used internally in many functions.

- Lots of Simulink models are provided in the subdirectory `simulink`. These models all have the prefix `sl_`.

## 1.2.2 General improvements

- Many functions now accept MATLAB style arguments given as trailing strings, or string-value pairs. These are parsed by the internal function `tb_optparse`.
- Many functions now handle sequences of rotation matrices or homogeneous transformations.
- Improved error messages in many functions
- Removed trailing commas from if and for statements

## 1.3 How to obtain the Toolbox

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

<http://www.petercorke.com>

The web page requests some information from you regarding such as your country, type of organization and application. This is just a means for me to gauge interest and to remind myself that this is a worthwhile activity.

The file is available in zip format (.zip). Download it and unzip it. Files all unpack to the correct parts of a hierarchy of directories (folders) headed by `rvctools`.

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

Ensure that the folder `rvctools` is on your MATLAB® search path. You can do this by issuing the `addpath` command at the MATLAB® prompt. Then issue the command `startup_rvc` and it will add a number of paths to your MATLAB® search path. 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, or by pressing the “Update Toolbox Path Cache” button under MATLAB® General preferences.

A menu-driven demonstration can be invoked by the function `rtbdemo`.

### 1.3.1 Documentation

This document `robot.pdf` is a 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.

The same documentation is available online in alphabetical order at [http://www.petercorke.com/RTB/r9/html/index\\_alpha.html](http://www.petercorke.com/RTB/r9/html/index_alpha.html) or by category at <http://www.petercorke.com/RTB/r9/html/index.html>. Documentation is also available via the MATLAB® help browser, “Robotics Toolbox” appears under the Contents.

## 1.4 MATLAB version issues

The Toolbox has been tested under R2014b. Compatibility problems are increasingly likely the older your version of MATLAB® is.

## 1.5 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).

## 1.6 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{Corkella,
  Author = {Peter I. Corke},
  Date-Added = {2011-01-12 08:19:32 +1000},
  Date-Modified = {2012-07-29 20:07:27 +1000},
  Note = {ISBN 978-3-642-20143-1},
```

```
Publisher = {Springer},  
Title = {Robotics, Vision \& Control: Fundamental Algorithms in {MATLAB}},  
Year = {2011}}
```

or

P.I. Corke, Robotics, Vision & Control: Fundamental Algorithms in MATLAB. Springer, 2011. ISBN 978-3-642-20143-1.

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

## 1.7 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://groups.google.com.au/group/robotics-tool-box>

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.

## 1.8 Related software

### 1.8.1 Octave

Octave is an open-source mathematical environment that is very similar to MATLAB<sup>®</sup>, but it has some important differences particularly with respect to graphics and classes. Many Toolbox functions work just fine under Octave. Three important classes (Quaternion, Link and SerialLink) will not work so modified versions of these classes is provided in the subdirectory called Octave. Copy all the directories from Octave to the main Robotics Toolbox directory.

The Octave port is a second priority for support and upgrades and is offered in the hope that you find it useful.

### 1.8.2 Python version

A python implementation of the Toolbox at <http://code.google.com/p/robotics-toolbox-python>. All core functionality of the release 8 Toolbox is present including kinematics, dynamics, Jacobians, quaternions etc. It is based on the python numpy class. The main current limitation is the lack of good 3D graphics support but people are working on this. Nevertheless this version of the toolbox is very usable and of course you don't need a MATLAB<sup>®</sup> licence to use it. Watch this space.

### 1.8.3 Machine Vision toolbox

Machine Vision toolbox (MVTB) for MATLAB<sup>®</sup>. 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 beyond the Math-work's Image Processing Toolbox. You can obtain this from <http://www.petercorke.com/vision>.

## 1.9 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).

## 1.10 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.

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 mentions to Gautam Sinha, Wynand Smart for models of industrial robot arm, Paul Pounds for the quadrotor and related models, and Paul Newman (Oxford) for inspiring the mobile robot code.

## Chapter 2

# Functions and classes

## about

### Compact display of variable type

**about(x)** displays a compact line that describes the class and dimensions of **x**.

**about x** as above but this is the command rather than functional form

### Examples

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

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

### See also

[whos](#)

---

## angdiff

### Difference of two angles

**d = angdiff(th1, th2)** returns the difference between angles **th1** and **th2** on the circle. The result is in the interval [-pi pi]. If **th1** is a column vector, and **th2** a scalar then re-

turn a column vector where **th2** is modulo subtracted from the corresponding elements of **th1**.

**d = angdiff(th)** returns the equivalent angle to **th** in the interval [-pi pi).

---

## angvec2r

### Convert angle and vector orientation to a rotation matrix

**R = angvec2r(theta, v)** is an orthonormal rotation matrix ( $3 \times 3$ ) equivalent to a rotation of **theta** about the vector **v**.

#### See also

[eul2r](#), [rpy2r](#), [tr2angvec](#)

---

## angvec2tr

### Convert angle and vector orientation to a homogeneous transform

**T = angvec2tr(theta, v)** is a homogeneous transform matrix ( $4 \times 4$ ) equivalent to a rotation of **theta** about the vector **v**.

#### Note

- The translational part is zero.

#### See also

[eul2tr](#), [rpy2tr](#), [angvec2r](#), [tr2angvec](#)

---

## Animate

### Create an animation

Helper class for creating animations. Saves snapshots of a figure as a folder of individual PNG format frames numbered 0000.png, 0001.png and so on.

### Example

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

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

```
% ffmpeg -r 10 -i movie/*.png out.mp4
```

---

## Animate.Animate

### Create an animation class

**a = ANIMATE(name, options)** initializes an animation, and creates a folder called **name** to hold the individual frames.

### Options

‘resolution’, R Set the resolution of the saved image to R pixels per inch.

---

## Animate.add

### Adds current plot to the animation

**A.ADD()** adds the current figure in PNG format to the animation folder with a unique sequential filename.

**A.ADD(fig)** as above but captures the figure **fig**.

**See also**[print](#)

# Arbotix

## Interface to Arbotix robot-arm controller

A concrete subclass of the abstract Machine class that implements a connection over a serial port to an Arbotix robot-arm controller.

### Methods

Arbotix	Constructor, establishes serial communications
delete	Destructor, closes serial connection
getpos	Get joint angles
setpos	Set joint angles and optionally speed
setpath	Load a trajectory into Arbotix RAM
relax	Control relax (zero torque) state
setled	Control LEDs on servos
gettemp	Temperature of motors
writedata1	Write byte data to servo control table
writedata2	Write word data to servo control table
readdata	Read servo control table
command	Execute command on servo
flush	Flushes serial data buffer
receive	Receive data

### Example

```
arb=Arbotix('port', '/dev/tty.usbserial-A800JDPN', 'nservos', 5);
q = arb.getpos();
arb.setpos(q + 0.1);
```

### Notes

- This is experimental code.
- Considers the robot as a string of motors, and the last joint is assumed to be the gripper. This should be abstracted, at the moment this is done in RobotArm.
- Connects via serial port to an Arbotix controller running the pypose sketch.

**See also**

[Machine](#), [RobotArm](#)

---

## Arbotix.Arbotix

### Create Arbotix interface object

`arb = Arbotix(options)` is an object that represents a connection to a chain of **Arbotix** servos connected via an **Arbotix** controller and serial link to the host computer.

### Options

‘port’, P	Name of the serial port device, eg. /dev/tty.USB0
‘baud’, B	Set baud rate (default 38400)
‘debug’, D	Debug level, show communications packets (default 0)
‘nservos’, N	Number of servos in the chain

## Arbotix.a2e

### Convert angle to encoder

`E = ARB.A2E(a)` is a vector of encoder values `E` corresponding to the vector of joint angles `a`. TODO:

- Scale factor is constant, should be a parameter to constructor.

## Arbotix.char

### Convert Arbotix status to string

`C = ARB.char()` is a string that succinctly describes the status of the **Arbotix** controller link.

## Arbotix.command

### Execute command on servo

`R = ARB.COMMAND(id, instruc)` executes the instruction `instruc` on servo `id`.

**R = ARB.COMMAND(id, instruc, data)** as above but the vector **data** forms the payload of the command message, and all numeric values in **data** must be in the range 0 to 255.

The optional output argument **R** is a structure holding the return status.

### Notes

- **id** is in the range 0 to N-1, where N is the number of servos in the system.
- Values for **instruc** are defined as class properties **INS\_\***.
- If ‘debug’ was enabled in the constructor then the hex values are echoed to the screen as well as being sent to the Arbotix.
- If an output argument is requested the serial channel is flushed first.

### See also

[Arbotix.receive](#), [Arbotix.flush](#)

---

## Arbotix.connect

### Connect to the physical robot controller

ARB.**connect()** establish a serial connection to the physical robot controller.

### See also

[Arbotix.disconnect](#)

---

## Arbotix.disconnect

### Disconnect from the physical robot controller

ARB.**disconnect()** closes the serial connection.

### See also

[Arbotix.connect](#)

---

## Arbotix.display

### Display parameters

ARB.**display()** displays the servo parameters in compact single line format.

### Notes

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

### See also

[Arbotix.char](#)

---

## Arbotix.e2a

### Convert encoder to angle

**a** = ARB.**E2A(E)** is a vector of joint angles **a** corresponding to the vector of encoder values **E**.

TODO:

- Scale factor is constant, should be a parameter to constructor.
- 

## Arbotix.flush

### Flush the receive buffer

ARB.**FLUSH()** flushes the serial input buffer, data is discarded.

**s** = ARB.**FLUSH()** as above but returns a vector of all bytes flushed from the channel.

### Notes

- Every command sent to the Arbotix elicits a reply.
- The method receive() should be called after every command.
- Some Arbotix commands also return diagnostic text information.

### See also

[Arbotix.receive](#), [Arbotix.parse](#)

---

## Arbotix.getpos

### Get position

**p** = ARB.**GETPOS**(**id**) is the position (0-1023) of servo **id**.

**p** = ARB.**GETPOS**([]) is a vector ( $1 \times N$ ) of positions of servos 1 to N.

### Notes

- N is defined at construction time by the ‘nservos’ option.

### See also

[Arbotix.e2a](#)

---

## Arbotix.gettemp

### Get temperature

**T** = ARB.**GETTEMP**(**id**) is the tempeature (deg C) of servo **id**.

**T** = ARB.**GETTEMP**() is a vector ( $1 \times N$ ) of the temperature of servos 1 to N.

### Notes

- N is defined at construction time by the ‘nservos’ option.

## Arbotix.parse

### Parse Arbotix return strings

ARB.**PARSE**(**s**) parses the string returned from the **Arbotix** controller and prints diagnostic text. The string **s** contains a mixture of Dynamixel style return packets and diagnostic text.

## Notes

- Every command sent to the Arbotix elicits a reply.
- The method receive() should be called after every command.
- Some Arbotix commands also return diagnostic text information.

## See also

[Arbotix.flush](#), [Arbotix.command](#)

---

# Arbotix.readdata

## Read byte data from servo control table

**R = ARB.READDATA(id, addr)** reads the successive elements of the servo control table for servo **id**, starting at address **addr**. The complete return status in the structure **R**, and the byte data is a vector in the field ‘params’.

## Notes

- **id** is in the range 0 to N-1, where N is the number of servos in the system.
- If ‘debug’ was enabled in the constructor then the hex values are echoed to the screen as well as being sent to the Arbotix.

## See also

[Arbotix.receive](#), [Arbotix.command](#)

---

# Arbotix.receive

## Decode Arbotix return packet

**R = ARB.RECEIVE()** reads and parses the return packet from the **Arbotix** and returns a structure with the following fields:

<b>id</b>	The id of the servo that sent the message
<b>error</b>	Error code, 0 means OK
<b>params</b>	The returned parameters, can be a vector of byte values

## Notes

- Every command sent to the Arbotix elicits a reply.
  - The method receive() should be called after every command.
  - Some Arbotix commands also return diagnostic text information.
  - If ‘debug’ was enabled in the constructor then the hex values are echoed
- 

# Arbotix.relax

## Control relax state

ARB.RELAX(**id**) causes the servo **id** to enter zero-torque (relax state) ARB.RELAX(**id**, FALSE) causes the servo **id** to enter position-control mode ARB.RELAX([]) causes servos 1 to N to relax ARB.RELAX() as above ARB.RELAX([], FALSE) causes servos 1 to N to enter position-control mode.

## Notes

- N is defined at construction time by the ‘nservos’ option.
- 

# Arbotix.setled

## Set LEDs on servos

ARB.led(**id**, **status**) sets the LED on servo **id** to on or off according to the **status** (logical).

ARB.led([], **status**) as above but the LEDs on servos 1 to N are set.

## Notes

- N is defined at construction time by the ‘nservos’ option.
- 

# Arbotix.setpath

## Load a path into Arbotix controller

ARB.setpath(**jt**) stores the path **jt** ( $P \times N$ ) in the **Arbotix** controller where P is the number of points on the path and N is the number of robot joints. Allows for smooth multi-axis motion.

### See also

[Arbotix.a2e](#)

---

## Arbotix.setpos

### Set position

ARB.**SETPOS**(**id**, **pos**) sets the position (0-1023) of servo **id**. ARB.**SETPOS**(**id**, **pos**, **SPEED**) as above but also sets the speed.

ARB.**SETPOS**(**pos**) sets the position of servos 1-N to corresponding elements of the vector **pos** ( $1 \times N$ ). ARB.**SETPOS**(**pos**, **SPEED**) as above but also sets the velocity **SPEED** ( $1 \times N$ ).

### Notes

- **id** is in the range 1 to N
- N is defined at construction time by the ‘nservos’ option.
- SPEED varies from 0 to 1023, 0 means largest possible speed.

### See also

[Arbotix.a2e](#)

---

## Arbotix.writedata1

### Write byte data to servo control table

ARB.**WRITEDATA1**(**id**, **addr**, **data**) writes the successive elements of **data** to the servo control table for servo **id**, starting at address **addr**. The values of **data** must be in the range 0 to 255.

### Notes

- **id** is in the range 0 to N-1, where N is the number of servos in the system.
- If ‘debug’ was enabled in the constructor then the hex values are echoed to the screen as well as being sent to the Arbotix.

### See also

[Arbotix.writedata2](#), [Arbotix.command](#)

---

## Arbotix.writedata2

### Write word data to servo control table

ARB.**WRITEDATA2**(**id**, **addr**, **data**) writes the successive elements of **data** to the servo control table for servo **id**, starting at address **addr**. The values of **data** must be in the range 0 to 65535.

### Notes

- **id** is in the range 0 to N-1, where N is the number of servos in the system.
- If ‘debug’ was enabled in the constructor then the hex values are echoed to the screen as well as being sent to the Arbotix.

### See also

[Arbotix.writedata1](#), [Arbotix.command](#)

---

---

## 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 (**x1,y1**) to (**x2,y2**).

**p** = **bresenham**(**p1**, **p2**) as above but **p1**=[**x1,y1**] and **p2**=[**x2,y2**].

### Notes

- Endpoints must be integer values.

## See also

[icanvas](#)

---

# 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

path	Compute a path from start to goal
visualize	Display the obstacle map (deprecated)
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
bug.goal = [50, 35]; % set the goal
bug.path([20, 10]); % animate path to (20,10)
```

## 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

## bug2 navigation object constructor

**b = Bug2(map)** 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 ( $1 \times 2$ )
- ‘inflate’, K    Inflate all obstacles by K cells.

### See also

[Navigation.Navigation](#)

---

# 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      Specify the name of the generated C-function. If this optional argument is omitted, the variable name of the first input argument is used, if possible.
- ‘output’, outVar      Defines the identifier of the output variable in the C-function.
- ‘vars’, varCells      The inputs to the C-code function must be defined as a cell array. The elements of this cell array contain the symbolic variables required to compute the output. The elements may be scalars, vectors or matrices symbolic variables. The C-function prototype will be composed accordingly as exemplified above.
- ‘flag’, sig            Specifies if function signature only is generated, default (false).

## 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](#)

---

# circle

## Compute points on a circle

**circle(C, R, opt)** plots a **circle** centred at **C** ( $1 \times 2$ ) with radius **R** on the current axes.

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

**C** is normally  $2 \times 1$  but if  $3 \times 1$  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)

---

# CodeGenerator

## Class for code generation

Objects of the CodeGenerator class automatically generate robot specific code, as either M-functions, C-functions, C-MEX functions, or real-time capable Simulink blocks.

The various methods return symbolic expressions for robot kinematic and dynamic functions, and optionally support side effects such as:

- M-functions with symbolic robot specific model code
- real-time capable robot specific Simulink blocks
- mat-files with symbolic robot specific model expressions
- C-functions and -headers with symbolic robot specific model code
- robot specific MEX functions based on the generated C-code (C-compiler must be installed).

## Example

```
% load robot model
mdl_twolink

cg = CodeGenerator(twolink);
cg.geneverything();

% a new class has been automatically generated in the robot directory.
addpath robot

tl = @robot();
% this class is a subclass of SerialLink, and thus polymorphic with
% SerialLink but its methods have been overloaded with robot-specific code,
% for example
T = tl.fkine([0.2 0.3]);
% uses concise symbolic expressions rather than the generalized A-matrix
% approach

% The Simulink block library containing robot-specific blocks can be
% opened by
open robot/robotslib.slx
% and the blocks dragged into your own models.
```

## Methods

gencoriolis	generate Coriolis/centripetal code
genfdyn	generate forward dynamics code
genfkine	generate forward kinematics code
genfriction	generate joint friction code
gengravload	generate gravity load code
geninertia	generate inertia matrix code
geninvdyn	generate inverse dynamics code
genjacobian	generate Jacobian code
geneverything	generate code for all of the above

## Properties (read/write)

basepath	basic working directory of the code generator
robjectpath	subdirectory for specialized MATLAB functions
sympath	subdirectory for symbolic expressions
slib	filename of the Simulink library
slibpath	subdirectory for the Simulink library
verbose	print code generation progress on console (logical)
saveresult	save symbolic expressions to .mat-files (logical)
logfile	print modeling progress to specified text file (string)
genmfun	generate executable M-functions (logical)
genslblock	generate Embedded MATLAB Function blocks (logical)
genccode	generate C-functions and -headers (logical)
genmex	generate MEX-functions as replacement for M-functions (logical)
compilemex	automatically compile MEX-functions after generation (logical)

## Properties (read only)

rob    SerialLink object to generate code for ( $1 \times 1$ ).

## Notes

- Requires the MATLAB Symbolic Toolbox.
- For robots with  $> 3$  joints the symbolic expressions are massively complex, they are slow and you may run out of memory.
- As much as possible the symbolic calculations are done row-wise to reduce the computation/memory burden.
- Requires a C-compiler if robot specific MEX-functions shall be generated as m-functions replacement (see MATLAB documentation of MEX files).

## Author

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

## See also

[SerialLink](#), [Link](#)

---

# CodeGenerator.CodeGenerator

## Construct a code generator object

`cGen = CodeGenerator(rob, options)` is a code generator object for the SerialLink object `rob`.

## Options

**CodeGenerator** has many options, and useful sets of options are called optionSets, and the following are recognized:

‘default’	set the options: verbose, saveResult, genMFun, genSLBlock
‘debug’	set the options: verbose, saveResult, genMFun, genSLBlock and create a logfile named ‘robModel.log’ in the working directory
‘silent’	set the options: saveResult, genMFun, genSLBlock
‘disk’	set the options: verbose, saveResult
‘workspace’	set the option: verbose; just outputs symbolic expressions to workspace
‘mfun’	set the options: verbose, saveResult, genMFun
‘slblock’	set the options: verbose, saveResult, genSLBlock
‘ccode’	set the options: verbose, saveResult, genCcode
‘mex’	set the options: verbose, saveResult, genMEX

If no optionSet is provided, then ‘default’ is used.

The options themselves control the code generation and user information:

‘verbose’	write code generation progress to command window
‘saveResult’	save results to hard disk (always enabled, when genMFun and genSLBlock are set)
‘logFile’, logfile	write code generation progress to specified logfile
‘genMFun’	generate robot specific m-functions
‘genSLBlock’	generate real-time capable robot specific Simulink blocks
‘genccode’	generate robot specific C-functions and -headers
‘mex’	generate robot specific MEX-functions as replacement for the m-functions
‘compilemex’	select whether generated MEX-function should be compiled directly after generation

Any option may also be modified individually as optional parameter value pairs.

## Author

Joern Malzahn 2012 RST, Technische Universitaet Dortmund, Germany. <http://www.rst.e-technik.tu-dortmund.de>

---

## CodeGenerator.addpath

### Adds generated code to search path

cGen.**addpath()** adds the generated m-functions and block library to the MATLAB function search path.

### Author

Joern Malzahn 2012 RST, Technische Universitaet Dortmund, Germany. <http://www.rst.e-technik.tu-dortmund.de>

### See also

[addpath](#)

---

## CodeGenerator.genccodecoriolis

### Generate C-function for robot inertia matrix

cGen.**genccodecoriolis()** generates robot-specific C-functions to compute the robot coriolis matrix.

### Notes

- Is called by CodeGenerator.gencoriolis if cGen has active flag gencode or genmex.
- The .c and .h files are generated in the directory specified by the ccodepath property of the CodeGenerator object.

### Author

Joern Malzahn, ([joern.malzahn@tu-dortmund.de](mailto:joern.malzahn@tu-dortmund.de))

### See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.gencoriolis](#), [CodeGenerator.genmexcoriolis](#)

---

## CodeGenerator.genccodefdyn

### Generate C-code for forward dynamics

cGen.**genccodeinvdyn()** generates a robot-specific C-code to compute the forward dynamics.

#### Notes

- Is called by CodeGenerator.genfdyn if cGen has active flag genccode or genmex.
- The .c and .h files are generated in the directory specified by the ccodepath property of the CodeGenerator object.
- The resulting C-function is composed of previously generated C-functions for the inertia matrix, Coriolis matrix, vector of gravitational load and joint friction vector. This function recombines these components to compute the forward dynamics.

#### Author

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

#### See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genfdyn](#), [CodeGenerator.genccodeinvdyn](#)

---

## CodeGenerator.genccodefkine

### Generate C-code for the forward kinematics

cGen.**genccodefkine()** generates a robot-specific C-function to compute forward kinematics.

#### Notes

- Is called by CodeGenerator.genfkine if cGen has active flag genccode or genmex
- The generated .c and .h files are written to the directory specified in the ccodepath property of the CodeGenerator object.

## **Author**

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

## **See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genfkine](#), [CodeGenerator.genmexfkine](#)

---

# **CodeGenerator.gencodefriction**

## **Generate C-code for the joint friction**

cGen.**gencodefriction**() generates a robot-specific C-function to compute vector of friction torques/forces.

## **Notes**

- Is called by CodeGenerator.genfriction if cGen has active flag gencode or genmex
- The generated .c and .h files are written to the directory specified in the ccodepath property of the CodeGenerator object.

## **Author**

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

## **See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genfriction](#), [CodeGenerator.genmexfriction](#)

---

# **CodeGenerator.gencodegravload**

## **Generate C-code for the vector of**

gravitational load torques/forces

cGen.**gencodegravload**() generates a robot-specific C-function to compute vector of gravitational load torques/forces.

## Notes

- Is called by `CodeGenerator.gengravload` if `cGen` has active flag `genccode` or `genmex`
- The generated .c and .h files are written to the directory specified in the `ccodepath` property of the `CodeGenerator` object.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.gengravload](#), [CodeGenerator.genmexgravload](#)

---

# CodeGenerator.genccodeinertia

## Generate C-function for robot inertia matrix

`cGen.genccodeinertia()` generates robot-specific C-functions to compute the robot inertia matrix.

## Notes

- Is called by `CodeGenerator.geninertia` if `cGen` has active flag `genccode` or `genmex`.
- The generated .c and .h files are generated in the directory specified by the `ccodepath` property of the `CodeGenerator` object.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninertia](#), [CodeGenerator.genmixinertia](#)

---

# CodeGenerator.genccodeinvdyn

## Generate C-code for inverse dynamics

cGen.**genccodeinvdyn()** generates a robot-specific C-code to compute the inverse dynamics.

### Notes

- Is called by CodeGenerator.geninvdyn if cGen has active flag genccode or genmex.
- The .c and .h files are generated in the directory specified by the ccodepath property of the CodeGenerator object.
- The resulting C-function is composed of previously generated C-functions for the inertia matrix, coriolis matrix, vector of gravitational load and joint friction vector. This function recombines these components to compute the inverse dynamics.

### Author

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

### See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninvdyn](#), [CodeGenerator.genccodef](#)

---

# CodeGenerator.genccodejacobian

## Generate C-functions for robot jacobians

cGen.**genccodejacobian()** generates a robot-specific C-function to compute the jacobians with respect to the robot base as well as the end effector.

### Notes

- Is called by CodeGenerator.genjacobian if cGen has active flag genccode or genmex.
- The generated .c and .h files are generated in the directory specified by the ccodepath property of the CodeGenerator object.

## **Author**

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

## **See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genccodefkine](#), [CodeGenerator.genjacobian](#)

---

# **CodeGenerator.gencoriolis**

## **Generate code for Coriolis force**

**coriolis** = cGen.**gencoriolis**() is a symbolic matrix ( $N \times N$ ) of centrifugal and Coriolis forces/torques.

## **Notes**

- The Coriolis matrix is stored row by row to avoid memory issues. The generated code recombines these rows to output the full matrix.
- Side effects of execution depends on the cGen flags:
  - saveresult: the symbolic expressions are saved to disk in the directory specified by cGen.sympath
  - genmfun: ready to use m-functions are generated and provided via a subclass of SerialLink stored in cGen.robjpath
  - genslblock: a Simulink block is generated and stored in a robot specific block library cGen.slib in the directory cGen.basepath
  - gencode: generates C-functions and -headers in the directory specified by the ccodepath property of the CodeGenerator object.
  - mex: generates robot specific MEX-functions as replacement for the m-functions mentioned above. Access is provided by the SerialLink subclass. The MEX files rely on the C code generated before.

## **Author**

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

## **See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninertia](#), [CodeGenerator.genfkine](#)

---

# CodeGenerator.genfdyn

## Generate code for forward dynamics

**Iqdd = cGen.genfdyn()** is a symbolic vector ( $1 \times N$ ) of joint inertial reaction forces/torques.

### Notes

- Side effects of execution depends on the cGen flags:
  - saveresult: the symbolic expressions are saved to disk in the directory specified by cGen.sympath
  - genmfun: ready to use m-functions are generated and provided via a subclass of SerialLink stored in cGen.robjpath
  - genslblock: a Simulink block is generated and stored in a robot specific block library cGen.slib in the directory cGen.basepath
  - gencode: generates C-functions and -headers in the directory specified by the ccodepath property of the CodeGenerator object.
  - mex: generates robot specific MEX-functions as replacement for the m-functions mentioned above. Access is provided by the SerialLink subclass. The MEX files rely on the C code generated before.

### Author

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

### See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninertia](#), [CodeGenerator.genfkine](#)

---

# CodeGenerator.genfkine

## Generate code for forward kinematics

**T = cGen.genfkine()** generates a symbolic homogeneous transform matrix ( $4 \times 4$ ) representing the pose of the robot end-effector in terms of the symbolic joint coordinates q1, q2, ...

**[T, allt] = cGen.genfkine()** as above but also generates symbolic homogeneous transform matrices ( $4 \times 4 \times N$ ) for the poses of the individual robot joints.

## Notes

- Side effects of execution depends on the cGen flags:
  - saveresult: the symbolic expressions are saved to disk in the directory specified by cGen.sympath
  - genmfun: ready to use m-functions are generated and provided via a subclass of SerialLink stored in cGen.robjpath
  - genslblock: a Simulink block is generated and stored in a robot specific block library cGen.slib in the directory cGen.basepath
  - gencode: generates C-functions and -headers in the directory specified by the ccodempath property of the CodeGenerator object.
  - mex: generates robot specific MEX-functions as replacement for the m-functions mentioned above. Access is provided by the SerialLink subclass. The MEX files rely on the C code generated before.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninvdyn](#), [CodeGenerator.genjacobian](#)

---

# CodeGenerator.genfriction

## Generate code for joint friction

**f** = cGen.**genfriction**() is the symbolic vector ( $1 \times N$ ) of joint friction forces.

## Notes

- Side effects of execution depends on the cGen flags:
  - saveresult: the symbolic expressions are saved to disk in the directory specified by cGen.sympath
  - genmfun: ready to use m-functions are generated and provided via a subclass of SerialLink stored in cGen.robjpath
  - genslblock: a Simulink block is generated and stored in a robot specific block library cGen.slib in the directory cGen.basepath
  - gencode: generates C-functions and -headers in the directory specified by the ccodempath property of the CodeGenerator object.

- mex: generates robot specific MEX-functions as replacement for the m-functions mentioned above. Access is provided by the SerialLink subclass. The MEX files rely on the C code generated before.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninvdyn](#), [CodeGenerator.genfdyn](#)

---

# CodeGenerator.gengravload

## Generate code for gravitational load

**g = cGen.gengravload()** is a symbolic vector ( $1 \times N$ ) of joint load forces/torques due to gravity.

## Notes

- Side effects of execution depends on the cGen flags:
  - saveresult: the symbolic expressions are saved to disk in the directory specified by cGen.sympath
  - genmfun: ready to use m-functions are generated and provided via a subclass of SerialLink stored in cGen.robjpath
  - genslblock: a Simulink block is generated and stored in a robot specific block library cGen.slib in the directory cGen.basepath
  - genccode: generates C-functions and -headers in the directory specified by the ccodempath property of the CodeGenerator object.
  - mex: generates robot specific MEX-functions as replacement for the m-functions mentioned above. Access is provided by the SerialLink subclass. The MEX files rely on the C code generated before.

## Author

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

**See also**

[CodeGenerator](#), [CodeGenerator.geninvdyn](#), [CodeGenerator.genfdyn](#)

---

## CodeGenerator.geninertia

### Generate code for inertia matrix

**i** = cGen.**geninertia**() is the symbolic robot inertia matrix ( $N \times N$ ).

### Notes

- The inertia matrix is stored row by row to avoid memory issues. The generated code recombines these rows to output the full matrix.
- Side effects of execution depends on the cGen flags:
  - saveresult: the symbolic expressions are saved to disk in the directory specified by cGen.sympath
  - genmfun: ready to use m-functions are generated and provided via a subclass of SerialLink stored in cGen.robjpath
  - genslblock: a Simulink block is generated and stored in a robot specific block library cGen.slib in the directory cGen.basepath
  - genccode: generates C-functions and -headers in the directory specified by the ccodepath property of the CodeGenerator object.
  - mex: generates robot specific MEX-functions as replacement for the m-functions mentioned above. Access is provided by the SerialLink subclass. The MEX files rely on the C code generated before.

### Author

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

**See also**

[CodeGenerator](#), [CodeGenerator](#), [CodeGenerator.geninvdyn](#), [CodeGenerator.genfdyn](#)

---

# CodeGenerator.geninvdyn

## Generate code for inverse dynamics

**tau** = cGen.**geninvdyn**() is the symbolic vector ( $1 \times N$ ) of joint forces/torques.

## Notes

- The inverse dynamics vector is composed of the previously computed inertia matrix coriolis matrix, vector of gravitational load and joint friction for speedup. The generated code recombines these components to output the final vector.
- Side effects of execution depends on the cGen flags:
  - saveresult: the symbolic expressions are saved to disk in the directory specified by cGen.sympath
  - genmfun: ready to use m-functions are generated and provided via a subclass of SerialLink stored in cGen.robjpath
  - genslblock: a Simulink block is generated and stored in a robot specific block library cGen.slib in the directory cGen.basepath
  - gencode: generates C-functions and -headers in the directory specified by the ccodepath property of the CodeGenerator object.
  - mex: generates robot specific MEX-functions as replacement for the m-functions mentioned above. Access is provided by the SerialLink subclass. The MEX files rely on the C code generated before.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genfdyn](#), [CodeGenerator.genfkine](#)

---

# CodeGenerator.genjacobian

## Generate code for robot Jacobians

**j0** = cGen.**genjacobian**() is the symbolic expression for the Jacobian matrix ( $6 \times N$ ) expressed in the base coordinate frame.

[**j0**, **Jn**] = cGen.**genjacobian**() as above but also returns the symbolic expression for the Jacobian matrix ( $6 \times N$ ) expressed in the end-effector frame.

## Notes

- Side effects of execution depends on the cGen flags:
  - saveresult: the symbolic expressions are saved to disk in the directory specified by cGen.sympath
  - genmfun: ready to use m-functions are generated and provided via a subclass of SerialLink stored in cGen.robjpath
  - genslblock: a Simulink block is generated and stored in a robot specific block library cGen.slib in the directory cGen.basepath
  - gencode: generates C-functions and -headers in the directory specified by the ccodepath property of the CodeGenerator object.
  - mex: generates robot specific MEX-functions as replacement for the m-functions mentioned above. Access is provided by the SerialLink subclass. The MEX files rely on the C code generated before.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genfkine](#)

---

# CodeGenerator.genmexcoriolis

## Generate C-MEX-function for robot coriolis matrix

cGen.[genmexcoriolis\(\)](#) generates robot-specific MEX-functions to compute robot coriolis matrix.

## Notes

- Is called by CodeGenerator.gencoriolis if cGen has active flag genmex
- The MEX file uses the .c and .h files generated in the directory specified by the ccodepath property of the CodeGenerator object.
- Access to generated functions is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.
- You will need a C compiler to use the generated MEX-functions. See the MATLAB documentation on how to setup the compiler in MATLAB. Nevertheless

the basic C-MEX-code as such may be generated without a compiler. In this case switch the cGen flag compilemex to false.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.gencoriolis](#)

---

# CodeGenerator.genmexfdyn

## Generate C-MEX-function for forward dynamics

cGen.**genmexfdyn**() generates a robot-specific MEX-function to compute the forward dynamics.

## Notes

- Is called by CodeGenerator.genfdyn if cGen has active flag genmex
- The MEX file uses the .c and .h files generated in the directory specified by the ccodempath property of the CodeGenerator object.
- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.
- You will need a C compiler to use the generated MEX-functions. See the MATLAB documentation on how to setup the compiler in MATLAB. Nevertheless the basic C-MEX-code as such may be generated without a compiler. In this case switch the cGen flag compilemex to false.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genfdyn](#), [CodeGenerator.genmexinvdyn](#)

---

# CodeGenerator.genmexfkine

## Generate C-MEX-function for forward kinematics

cGen.**genmexfkine()** generates a robot-specific MEX-function to compute forward kinematics.

### Notes

- Is called by CodeGenerator.genfkine if cGen has active flag genmex
- The MEX file uses the .c and .h files generated in the directory specified by the ccodempath property of the CodeGenerator object.
- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.
- You will need a C compiler to use the generated MEX-functions. See the MATLAB documentation on how to setup the compiler in MATLAB. Nevertheless the basic C-MEX-code as such may be generated without a compiler. In this case switch the cGen flag compilemex to false.

### Author

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

### See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genfkine](#)

---

# CodeGenerator.genmexfriction

## Generate C-MEX-function for joint friction

cGen.**genmexfriction()** generates a robot-specific MEX-function to compute the vector of joint friction.

### Notes

- Is called by CodeGenerator.genfriction if cGen has active flag genmex
- The MEX file uses the .c and .h files generated in the directory specified by the ccodempath property of the CodeGenerator object.

- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.
- You will need a C compiler to use the generated MEX-functions. See the MATLAB documentation on how to setup the compiler in MATLAB. Nevertheless the basic C-MEX-code as such may be generated without a compiler. In this case switch the cGen flag compilemex to false.

## **Author**

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

## **See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.gengravload](#)

---

# **CodeGenerator.genmexgravload**

## **Generate C-MEX-function for gravitational load**

cGen.**genmexgravload()** generates a robot-specific MEX-function to compute gravitation load forces and torques.

## **Notes**

- Is called by CodeGenerator.gengravload if cGen has active flag genmex
- The MEX file uses the .c and .h files generated in the directory specified by the ccodepath property of the CodeGenerator object.
- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.
- You will need a C compiler to use the generated MEX-functions. See the MATLAB documentation on how to setup the compiler in MATLAB. Nevertheless the basic C-MEX-code as such may be generated without a compiler. In this case switch the cGen flag compilemex to false.

## **Author**

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

**See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.gengravload](#)

---

## CodeGenerator.genmixinertia

### Generate C-MEX-function for robot inertia matrix

cGen.**genmixinertia()** generates robot-specific MEX-functions to compute robot inertia matrix.

#### Notes

- Is called by CodeGenerator.geninertia if cGen has active flag genmex
- The MEX file uses the .c and .h files generated in the directory specified by the ccodempath property of the CodeGenerator object.
- Access to generated functions is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.
- You will need a C compiler to use the generated MEX-functions. See the MATLAB documentation on how to setup the compiler in MATLAB. Nevertheless the basic C-MEX-code as such may be generated without a compiler. In this case switch the cGen flag compilemex to false.

#### Author

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

**See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninertia](#)

---

## CodeGenerator.genmexinvdyn

### Generate C-MEX-function for inverse dynamics

cGen.**genmexinvdyn()** generates a robot-specific MEX-function to compute the inverse dynamics.

## Notes

- Is called by `CodeGenerator.geninvdyn` if `cGen` has active flag `genmex`.
- The MEX file uses the .c and .h files generated in the directory specified by the `ccodepath` property of the `CodeGenerator` object.
- Access to generated function is provided via subclass of `SerialLink` whose class definition is stored in `cGen.robjpath`.
- You will need a C compiler to use the generated MEX-functions. See the MATLAB documentation on how to setup the compiler in MATLAB. Nevertheless the basic C-MEX-code as such may be generated without a compiler. In this case switch the `cGen` flag `compilemex` to false.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.gengravload](#)

---

# CodeGenerator.genmexjacobian

## Generate C-MEX-function for the robot Jacobians

`cGen.genmexjacobian()` generates robot-specific MEX-function to compute the robot Jacobian with respect to the base as well as the end effector frame.

## Notes

- Is called by `CodeGenerator.genjacobian` if `cGen` has active flag `genmex`.
- The MEX file uses the .c and .h files generated in the directory specified by the `ccodepath` property of the `CodeGenerator` object.
- Access to generated function is provided via subclass of `SerialLink` whose class definition is stored in `cGen.robjpath`.
- You will need a C compiler to use the generated MEX-functions. See the MATLAB documentation on how to setup the compiler in MATLAB. Nevertheless the basic C-MEX-code as such may be generated without a compiler. In this case switch the `cGen` flag `compilemex` to false.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genjacobian](#)

---

# CodeGenerator.genmfuncoriolis

## Generate M-functions for Coriolis matrix

cGen.[genmfuncoriolis](#)() generates a robot-specific M-function to compute the Coriolis matrix.

## Notes

- Is called by CodeGenerator.gencoriolis if cGen has active flag genmf
- The Coriolis matrix is stored row by row to avoid memory issues.
- The generated M-function recombines the individual M-functions for each row.
- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.gencoriolis](#), [CodeGenerator.geninertia](#)

---

# CodeGenerator.genmfunfdyn

## Generate M-function for forward dynamics

cGen.[genmfunfdyn](#)() generates a robot-specific M-function to compute the forward dynamics.

## Notes

- Is called by `CodeGenerator.genfdyn` if `cGen` has active flag `genmfun`
- The generated M-function is composed of previously generated M-functions for the inertia matrix, coriolis matrix, vector of gravitational load and joint friction vector. This function recombines these components to compute the forward dynamics.
- Access to generated function is provided via subclass of `SerialLink` whose class definition is stored in `cGen.robjpath`.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninvdyn](#)

---

# CodeGenerator.genmfunkine

## Generate M-function for forward kinematics

`cGen.genmfunkine()` generates a robot-specific M-function to compute forward kinematics.

## Notes

- Is called by `CodeGenerator.genfkine` if `cGen` has active flag `genmfun`
- Access to generated function is provided via subclass of `SerialLink` whose class definition is stored in `cGen.robjpath`.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genjacobian](#)

---

## CodeGenerator.genmfunfriction

### Generate M-function for joint friction

cGen.**genmfunfriction()** generates a robot-specific M-function to compute joint friction.

#### Notes

- Is called only if cGen has active flag genmf
- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.

#### Author

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

#### See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.gengravload](#)

---

## CodeGenerator.genmfungravload

### Generate M-functions for gravitational load

cGen.**genmfungravload()** generates a robot-specific M-function to compute gravitational load forces and torques.

#### Notes

- Is called by CodeGenerator.gengravload if cGen has active flag genmf
- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.

#### Author

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

**See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninertia](#)

---

## CodeGenerator.genmfuninertia

### Generate M-function for robot inertia matrix

cGen.**genmfuninertia()** generates a robot-specific M-function to compute robot inertia matrix.

**Notes**

- Is called by CodeGenerator.geninertia if cGen has active flag genmfun
- The inertia matrix is stored row by row to avoid memory issues.
- The generated M-function recombines the individual M-functions for each row.
- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.

**Author**

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

**See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.gencoriolis](#)

---

## CodeGenerator.genmfuninvdyn

### Generate M-functions for inverse dynamics

cGen.**genmfuninvdyn()** generates a robot-specific M-function to compute inverse dynamics.

**Notes**

- Is called by CodeGenerator.geninvdyn if cGen has active flag genmfun

- The generated M-function is composed of previously generated M-functions for the inertia matrix, coriolis matrix, vector of gravitational load and joint friction vector. This function recombines these components to compute the inverse dynamics.
- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.

## **Author**

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

## **See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninvdyn](#)

---

# **CodeGenerator.genmfunjacobian**

## **Generate M-functions for robot Jacobian**

cGen.**genmfunjacobian()** generates a robot-specific M-function to compute robot Jacobian.

## **Notes**

- Is called by CodeGenerator.genjacobian, if cGen has active flag genmfun
- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.

## **Author**

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

## **See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.gencoriolis](#)

---

## CodeGenerator.genslblockcoriolis

### Generate Simulink block for Coriolis matrix

cGen.**genslblockcoriolis()** generates a robot-specific Simulink block to compute Coriolis/centripetal matrix.

#### Notes

- Is called by CodeGenerator.gencoriolis if cGen has active flag genslblock
- The Coriolis matrix is stored row by row to avoid memory issues.
- The Simulink block recombines the individual blocks for each row.
- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.

#### Author

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

#### See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.gencoriolis](#)

---

## CodeGenerator.genslblockfdyn

### Generate Simulink block for forward dynamics

cGen.**genslblockfdyn()** generates a robot-specific Simulink block to compute forward dynamics.

#### Notes

- Is called by CodeGenerator.genfdyn if cGen has active flag genslblock
- The generated Simulink block is composed of previously generated blocks for the inertia matrix, coriolis matrix, vector of gravitational load and joint friction vector. The block recombines these components to compute the forward dynamics.
- Access to generated function is provided via subclass of SerialLink whose class definition is stored in cGen.robjpath.

## **Author**

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

## **See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genfdyn](#)

---

# **CodeGenerator.genslblockfkine**

## **Generate Simulink block for forward kinematics**

cGen.[genslblockfkine](#)() generates a robot-specific Simulink block to compute forward kinematics.

## **Notes**

- Is called by CodeGenerator.genfkine if cGen has active flag genslblock.
- The Simulink blocks are generated and stored in a robot specific block library cGen.slib in the directory cGen.basepath.
- Blocks are created for intermediate transforms T0, T1 etc. as well.

## **Author**

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

## **See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genfkine](#)

---

# **CodeGenerator.genslblockfriction**

## **Generate Simulink block for joint friction**

cGen.[genslblockfriction](#)() generates a robot-specific Simulink block to compute the joint friction model.

## Notes

- Is called by `CodeGenerator.genfriction` if `cGen` has active flag `genslblock`
- The Simulink blocks are generated and stored in a robot specific block library `cGen.slib` in the directory `cGen.basepath`.

## Author

Joern Malzahn, ([joern.malzahn@tu-dortmund.de](mailto:joern.malzahn@tu-dortmund.de))

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genfriction](#)

---

# CodeGenerator.genslblockgravload

## Generate Simulink block for gravitational load

`cGen.genslblockgravload()` generates a robot-specific Simulink block to compute gravitational load.

## Notes

- Is called by `CodeGenerator.gengravload` if `cGen` has active flag `genslblock`
- The Simulink blocks are generated and stored in a robot specific block library `cGen.slib` in the directory `cGen.basepath`.

## Author

Joern Malzahn, ([joern.malzahn@tu-dortmund.de](mailto:joern.malzahn@tu-dortmund.de))

See also [CodeGenerator.CodeGenerator](#), [CodeGenerator.gengravload](#)

---

# CodeGenerator.genslblockinertia

## Generate Simulink block for inertia matrix

`cGen.genslbgenslblockinertia()` generates a robot-specific Simulink block to compute robot inertia matrix.

## Notes

- Is called by `CodeGenerator.geninertia` if `cGen` has active flag `genslblock`
- The Inertia matrix is stored row by row to avoid memory issues.
- The Simulink block recombines the individual blocks for each row.
- The Simulink blocks are generated and stored in a robot specific block library `cGen.slib` in the directory `cGen.basepath`.

## Author

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

## See also

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninertia](#)

---

# CodeGenerator.genslblockinvdyn

## Generate Simulink block for inverse dynamics

`cGen.genslblockinvdyn()` generates a robot-specific Simulink block to compute inverse dynamics.

## Notes

- Is called by `CodeGenerator.geninvdyn` if `cGen` has active flag `genslblock`
- The generated Simulink block is composed of previously generated blocks for the inertia matrix, coriolis matrix, vector of gravitational load and joint friction vector. The block recombines these components to compute the forward dynamics.
- The Simulink blocks are generated and stored in a robot specific block library `cGen.slib` in the directory `cGen.basepath`.

## Author

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

**See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.geninvdyn](#)

---

## CodeGenerator.genslblockjacobian

### Generate Simulink block for robot Jacobians

cGen.**genslblockjacobian()** generates a robot-specific Simulink block to compute robot Jacobians (world and tool frame).

#### Notes

- Is called by CodeGenerator.genjacobian if cGen has active flag genslblock
- The Simulink blocks are generated and stored in a robot specific block library cGen.slib in the directory cGen.basepath.

#### Author

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

**See also**

[CodeGenerator.CodeGenerator](#), [CodeGenerator.genjacobian](#)

---

## CodeGenerator.logmsg

### Print CodeGenerator logs.

count = CGen.logmsg( FORMAT, A, ... ) is the number of characters written to the CGen.logfile. For the additional arguments see fprintf.

#### Note

Matlab ships with a function for writing formatted strings into a text file or to the console (fprintf). The function works with single target identifiers (file, console, string). This function uses the same syntax as for the fprintf function to output log messages to either the Matlab console, a log file or both.

## **Authors**

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

## **See also**

[multidfprintf](#), [fprintf](#), [sprintf](#)

---

# **CodeGenerator.purge**

## **Cleanup generated files**

cGen.**purge**() deletes all generated files, first displays a question dialog to make sure the user really wants to delete all generated files.

cGen.**purge**(1) as above but skips the question dialog.

## **Author**

Joern Malzahn 2012 RST, Technische Universitaet Dortmund, Germany. <http://www.rst.e-technik.tu-dortmund.de>

---

# **CodeGenerator.rmpath**

## **Removes generated code from search path**

cGen.**rmpath**() removes generated m-functions and block library from the MATLAB function search path.

## **Author**

Joern Malzahn 2012 RST, Technische Universitaet Dortmund, Germany. <http://www.rst.e-technik.tu-dortmund.de>

## **See also**

[rmpath](#)

---

## colnorm

### Column-wise norm of a matrix

**cn** = **colnorm(a)** is vector ( $1 \times M$ ) of the norms of each column of the matrix **a** ( $N \times M$ ).

---

## colorname

### Map between color names and RGB values

**rgb** = **colorname(name)** is the **rgb**-tristimulus value ( $1 \times 3$ ) 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 ( $1 \times 3$ ). 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
-

## 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  $i^{\text{th}}$  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  $i^{\text{th}}$  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).

### Reference

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

### See also

[lspb](#), [mstraj](#), [trinterp](#), [Quaternion.interp](#), [transl](#)

---

## delta2tr

### Convert differential motion to a homogeneous transform

**T = delta2tr(d)** is a homogeneous transform ( $4 \times 4$ ) 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](#)

---

# 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	a command string that will create a SerialLink() object representing the specified kinematics
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 generated.

## 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](#)

---

# 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](#)

---

# distanceform

## Distance transform of occupancy grid

**d = distanceform(world, goal)** is the distance transform of the occupancy grid **world** 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.

**d = distanceform(world, goal, metric)** as above but specifies the distance metric as either ‘cityblock’ or ‘Euclidean’ (default).

**d = distanceform(world, goal, metric, show)** as above but shows an animation of the distance transform being formed, with a delay of **show** seconds between frames.

## Notes

- The Machine Vision Toolbox function imorph is required.
- The goal is [X, Y] not MATLAB [row,col].

## See also

[imorph](#), [DXform](#)

---

# 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 expressions 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](#)

---

## 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

---

## 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

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

costmap Distance from each point to the goal.

## 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.path(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.
- 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 converted to a costmap with a unit cost for traversing a cell.

### Options

‘goal’, G	Specify the goal point ( $2 \times 1$ )
‘metric’, M	Specify the distance metric as ‘euclidean’ (default) or ‘cityblock’.
‘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](#)

---

## 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.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](#)

---

## Dstar.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](#)

---

## Dstar.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 traverse. 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](#)

---

## Dstar.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](#)

---

## Dstar.modify\_cost

### Modify cost map

### Notes

- Deprecated: use `modify_cost` instead instead.

### See also

[Dstar.costmap\\_set](#), [Dstar.costmap\\_get](#)

---

## Dstar.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.
-

## 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.

---

## 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

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

- distancemap The distance transform of the occupancy grid.  
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.path(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](#), [distanceform](#)

---

# 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 ( $2 \times 1$ )  
‘metric’, M Specify the distance metric as ‘euclidean’ (default) or ‘cityblock’.  
‘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**() updates the internal distancemap where the value of each element is the minimum distance from the corresponding point to the goal. The goal is as specified to the constructor.

DX.**plan(goal)** as above but uses the specified goal.

DX.**plan(goal, s)** as above but displays the evolution of the distancemap, with one iteration displayed every **s** seconds.

### Notes

- This may take many seconds.

**See also**

[Navigation.path](#)

---

## DXform.plot

### Visualize navigation environment

DX.**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.

DX.**plot(p)** as above but also overlays a path given by the set of points **p** ( $2 \times M$ ).

**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 linestyle **ls**.

**See also**

[Navigation.plot](#)

---

## 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  $\mathbb{R}^K$ .

**See also**

[h2e](#)

---

## edgelist

### Return list of edge pixels for region

**eg = edgelist(im, seed)** is a list of edge pixels ( $N \times 2$ ) of a region in the image **im** starting at edge coordinate **seed=[X,Y]**. The **edgelist** has one row 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 connectvity.

## Reference

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

## See also

[ilabel](#)

---

# 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 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 object.
- The vehicle must be driven within the area of the map and this is achieved by connecting the Vehicle object to a Driver object.
- a map containing the position of a number of landmark points and is represented by a Map object.
- a sensor that returns measurements about landmarks relative to the vehicle's location 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. 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 feature points and confidence limits
plot_ellipse	plot estimated path with 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
features	maps sensor feature id to filter state element
robot	reference to the Vehicle object
sensor	reference to the Sensor subclass object
history	vector of structs that hold the detailed filter state from each time step
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 at every 20 time steps

```
ekf.plot_ellipse(20, '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 features, create a sensor that uses the map and vehicle state to estimate feature range and bearing with covariance W, the Kalman filter with estimated covariances V\_est and W\_est and initial vehicle state covariance P0

```
veh = Vehicle(V);
veh.add_driver( RandomPath(20, 2) );
map = Map(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 at every 20 time steps

```
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 feature 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) );
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 3 sigma ellipses

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

## Simultaneous localization and mapping (SLAM)

Create a vehicle with odometry covariance V, add a driver to it, create a map with 20 point features, create a sensor that uses the map and vehicle state to estimate feature 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 = Map(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 at every 20 time steps

```
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](#), [Map](#), [ParticleFilter](#)

---

# EKF.EKF

## EKF object constructor

**E = EKF(vehicle, v\_est, p0, options)** is an **EKF** that estimates the state of the **vehicle** with estimated odometry covariance **v\_est** ( $2 \times 2$ ) and initial covariance ( $3 \times 3$ ).

**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**.

## Options

‘verbose’	Be verbose.
‘nohistory’	Don’t keep history.
‘joseph’	Use Joseph form for covariance
‘dim’, D	Dimension of the robot’s workspace. Scalar D is $D \times D$ , 2-vector D(1)xD(2), 4-vector is D(1)<x<D(2), D(3)<y<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](#), [Sensor](#), [RangeBearingSensor](#), [Map](#)

---

## 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.init

### Reset the filter

E.[init\(\)](#) resets the filter state and clears the 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\(i\)](#) as above but for i points along the path.

E.[plot\\_ellipse\(i, ls\)](#) as above but pass line style arguments ls to [plot\\_ellipse](#). If i is [] then assume 20.

### 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). Heading error is wrapped into the range [-pi,pi)

**out** = E.[plot\\_error\(\)](#) is the estimation error versus time as a matrix ( $N \times 3$ ) where each row is x, y, theta.

### Options

‘bound’, S	Display the S sigma confidence bounds (default 3). If S =0 do not display bounds.
‘boundcolor’, 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.
- 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()` overlay the current plot with the estimated landmark position (a +-marker) and a covariance ellipses.

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

`p = E.plot_map()` is the estimated landmark locations ( $2 \times N$ ) and column  $I$  is the  $I$ 'th map feature. If the landmark was not estimated the corresponding column contains NaNs.

## See also

[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.

`m = E.plot_P()` is the estimated covariance magnitude at all time steps as a vector.

---

## 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.

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

### See also

[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.
- 

## eul2jac

### Euler angle rate Jacobian

`J = eul2jac(eul)` is a Jacobian matrix ( $3 \times 3$ ) that maps Euler angle rates to angular velocity at the operating point `eul=[PHI, THETA, PSI]`.

`J = eul2jac(phi, theta, psi)` as above but the Euler angles are passed as separate arguments.

## Notes

- Used in the creation of an analytical Jacobian.

## See also

[rpy2jac](#), [SerialLink.JACOBN](#)

---

# eul2r

## Convert Euler angles to rotation matrix

**R = eul2r(phi, theta, psi)** is an SO(2) orthonormal rotation matrix ( $3 \times 3$ ) 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 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 and **R** is a three-dimensional matrix ( $3 \times 3 \times N$ ), where the last index corresponds to rows of **eul** which are assumed to be [**phi**, **theta**, **psi**].

## Options

‘deg’ Compute angles in degrees (radians default)

## Note

- The vectors **phi**, **theta**, **psi** must be of the same length.

## See also

[eul2tr](#), [rpy2tr](#), [tr2eul](#)

---

## eul2tr

### Convert Euler angles to homogeneous transform

**T = eul2tr(phi, theta, psi, options)** is a SE(3) homogeneous transformation matrix ( $4 \times 4$ ) 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**.

**T = eul2tr(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 and T is a three-dimensional matrix ( $4 \times 4 \times N$ ), where the last index corresponds to rows of **eul** which are assumed to be [**phi**, **theta**, **psi**].

### Options

‘deg’ Compute angles 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](#)

---

## gauss2d

### Gaussian kernel

**out = gauss2d(im, sigma, C)** is a unit volume Gaussian kernel rendered into matrix **out** ( $W \times H$ ) the same size as **im** ( $W \times H$ ). The Gaussian has a standard deviation of **sigma**. The Gaussian is centered at **C**=[U,V].

---

---

## **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  $P^K$ .

#### **See also**

[e2h](#)

---

## **homline**

### **Homogeneous line from two points**

**L = homline(x1, y1, x2, y2)** is a vector ( $3 \times 1$ ) which describes a line in homogeneous form that contains the two Euclidean points **(x1,y1)** and **(x2,y2)**.

Homogeneous points X ( $3 \times 1$ ) on the line must satisfy  $L^*X = 0$ .

#### **See also**

[plot\\_homline](#)

---

## **homtrans**

### **Apply a homogeneous transformation**

**p2 = homtrans(T, p)** applies homogeneous transformation **T** to the points stored columnwise in **p**.

- If **T** is in SE(2) ( $3 \times 3$ ) and
  - **p** is  $2 \times N$  (2D points) they are considered Euclidean ( $R^2$ )
  - **p** is  $3 \times N$  (2D points) they are considered projective ( $p^2$ )
- If **T** is in SE(3) ( $4 \times 4$ ) and
  - **p** is  $3 \times N$  (3D points) they are considered Euclidean ( $R^3$ )

–  $\mathbf{p}$  is  $4 \times N$  (3D points) they are considered projective ( $\mathbf{p}^3$ )

**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** =  $N \times N$  and **T**= $N \times N \times \mathbf{p}$  then the result is  $N \times N \times \mathbf{p}$ .

### See also

[e2h](#), [h2e](#)

---

## ishomog

### Test if SE(3) homogeneous transformation

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

**ishomog**(**T**, ‘valid’) as above, but also checks the validity of the rotation sub-matrix.

### Notes

- The first form is a fast, but incomplete, test for a transform is SE(3).
- Does not work for the SE(2) case.

### See also

[isrot](#), [isvec](#)

---

## ishomog2

### Test if SE(2) homogeneous transformation

**ishomog2**(**T**) is true (1) if the argument **T** is of dimension  $3 \times 3$  or  $3 \times 3 \times N$ , else false (0).

**ishomog2**(**T**, ‘valid’) as above, but also checks the validity of the rotation sub-matrix.

## Notes

- The first form is a fast, but incomplete, test for a transform in SE(3).
- Does not work for the SE(3) case.

## See also

[ishomog](#), [isrot2](#), [isvec](#)

---

# isrot

## Test if SO(3) rotation matrix

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

**isrot(R, ‘valid’)** as above, but also checks the validity of the rotation matrix.

## Notes

- A valid rotation matrix has determinant of 1.

## See also

[ishomog](#), [isvec](#)

---

# isrot2

## Test if SO(2) rotation matrix

**isrot2(R)** is true (1) if the argument is of dimension  $2 \times 2$  or  $2 \times 2 \times N$ , else false (0).

**isrot2(R, ‘valid’)** as above, but also checks the validity of the rotation matrix.

## Notes

- A valid rotation matrix has determinant of 1.

## See also

[ishomog2](#), [isvec](#)

---

# isvec

## Test if vector

**isvec**(v) is true (1) if the argument v is a 3-vector, else 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, the latter returns true for the case of a scalar, **isvec** does not.
- Gives same result for row- or column-vector, ie.  $3 \times 1$  or  $1 \times 3$  gives true.

## See also

[ishomog](#), [isrot](#)

---

# joy2tr

## Update transform from joystick

T = **joy2tr**(T, options) updates the SE(3) homogeneous transform ( $4 \times 4$ ) 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](http://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 between two configurations

[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 and final joint velocity for the trajectory.

[q,qd,qdd] = **jtraj**(q0, qf, T) as above but the trajectory length 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.

#### See also

[qplot](#), [ctraj](#), [SerialLink.jtraj](#)

---

# Link

## Robot manipulator Link class

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

### Methods

A	link transform matrix
RP	joint type: 'R' or 'P'
friction	friction force
nofriction	Link object with friction parameters set to zero
dyn	display link dynamic parameters
islimit	test if joint exceeds soft limit
isrevolute	test if joint is revolute
isprismatic	test if joint is prismatic
display	print the link parameters in human readable form
char	convert to string

### Properties (read/write)

theta	kinematic: joint angle
d	kinematic: link offset
a	kinematic: link length
alpha	kinematic: link twist
sigma	kinematic: 0 if revolute, 1 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 $3 \times 1$
I	dynamic: link inertia matrix, symmetric $3 \times 3$ , 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.

## References

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

## 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**.

**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 ( $3 \times 1$ , $6 \times 1$ or $3 \times 3$ )
‘r’, R	link centre of gravity ( $3 \times 1$ )
‘m’, M	link mass ( $1 \times 1$ )
‘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 ( $1 \times 1$ or $2 \times 1$ ), (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

- It is an error to specify both ‘theta’ and ‘d’
- The link inertia matrix ( $3 \times 3$ ) is symmetric and can be specified by giving a  $3 \times 3$  matrix, the diagonal elements [Ix<sub>x</sub> Iy<sub>y</sub> Iz<sub>z</sub>], or the moments and products of inertia [Ix<sub>x</sub> Iy<sub>y</sub> Iz<sub>z</sub> 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 OFFSET is a constant displacement between the user joint angle vector and the true kinematic solution.
- **dh** = [THETA D A ALPHA SIGMA] where SIGMA=0 for a revolute and 1 for a prismatic joint, 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’

## 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 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.
- 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](#)

---

## Link.A

### Link transform matrix

$T = L.A(q)$  is the link homogeneous transformation matrix ( $4 \times 4$ ) corresponding to the link variable  $q$  which is either the Denavit-Hartenberg parameter THETA (revolute) or D (prismatic).

### 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.
- 

## Link.char

### Convert to string

$s = L.\text{char}()$  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 for link velocity **qd**.

## Notes

- 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.
- 

## 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 symbolic parameters.

---

## Link.nofriction

### Remove friction

`In = L.nofriction()` is a link object with the same parameters as `L` except nonlinear (Coulomb) friction parameter is zero.

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

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

`In = 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**

[SerialLink.nofriction](#), [SerialLink.fdyn](#)

---

## Link.RP

### Joint type

`c = L.RP()` is a character ‘R’ or ‘P’ depending on whether joint is revolute or prismatic respectively. If `L` is a vector of **Link** objects return a string of characters in joint order.

---

## **Link.set.I**

### **Set link inertia**

L.I = [Ix Iy Iz] sets link inertia to a diagonal matrix.

L.I = [Ix Iy Iz Ixy Iyz Ixz] sets link inertia to a symmetric matrix with specified inertia and product of inertia elements.

L.I = M set **Link** inertia matrix to M ( $3 \times 3$ ) 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](#)

---

## **lspb**

### **Linear segment with parabolic blend**

**[s,sd,sdd] = lspb(s0, sf, m)** is a scalar trajectory ( $\mathbf{m} \times 1$ ) that varies smoothly from **s0** to **sf** in **m** steps using a constant velocity segment and parabolic blends (a trapezoidal path). Velocity and acceleration can be optionally returned as **sd** ( $\mathbf{m} \times 1$ ) and **sdd** ( $\mathbf{m} \times 1$ ).

**[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** ( $\mathbf{m} \times 1$ ).

**[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**

- 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** ( $\mathbf{n} \times \mathbf{n}$ ) 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 <b>map</b>
q	leave editing mode and return <b>map</b>

## See also

[dxfread](#), [PRM](#), [RRT](#)

---

# Map

## Map of planar point features

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

### Methods

plot	Plot the feature map
feature	Return a specified map feature
display	Display map parameters in human readable form
char	Convert map parameters to human readable string

### Properties

map	Matrix of map feature coordinates $2 \times N$
dim	The dimensions of the map region x,y in [-dim,dim]
nfeatures	The number of map features 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 feature points

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

which can be displayed by

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

## Reference

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

### See also

[RangeBearingSensor](#), [EKF](#)

---

## Map.Map

### Create a map of point feature landmarks

**m = Map(n, dim, options)** is a **Map** object that represents **n** random point features in a planar region bounded by +/-**dim** in the x- and y-directions.

### Options

‘verbose’ Be verbose

---

## Map.char

### Convert map parameters to a string

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

---

## Map.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 Map object and the command has no trailing semicolon.

### See also

[map.char](#)

---

## Map.feature

### Get landmarks from map

**f** = M.**feature**(**k**) is the coordinate ( $2 \times 1$ ) of the **k**'th map **feature** (landmark).

---

## Map.plot

### Plot the map

M.**plot**() plots the feature map in the current figure, as a square region with dimensions given by the M.dim property. Each feature 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.
- 

## Map.show

### Show the feature map

### Notes

- Deprecated, use **plot** method.
- 

## Map.verbosity

### Set verbosity

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

## **mdl\_3link3d**

**(C) 1993-2015, by Peter I. Corke**

This file is part of The Robotics Toolbox for MATLAB (RTB).

RTB is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

RTB is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with RTB. If not, see <<http://www.gnu.org/licenses/>>.

<http://www.petercorke.com>

---

## **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

[SerialLink](#), [mdl\\_coil](#)

---

# 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 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

[SerialLink](#), [mdl\\_nao](#)

---

## **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**

[SerialLink](#), [mdl\\_ball](#)

---

## **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](mailto:wynand.swart@gmail.com)

## **See also**

[SerialLink](#), [mdl\\_irb140](#), [mdl\\_m16](#), [mdl\\_motomanhp6](#), [mdl\\_puma560](#)

---

# **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**

- 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**

[SerialLink](#), [mdl\\_hyper3d](#), [mdl\\_coil](#), [mdl\\_ball](#), [mdl\\_twolink](#)

---

## 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

- 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

[SerialLink](#), [mdl\\_hyper2d](#), [mdl\\_ball](#), [mdl\\_coil](#)

---

## mdl\_irb140

### Create model of ABB IRB 140 manipulator

MDL\_IRB140 is a script that creates the workspace variable robot 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

[SerialLink](#), [mdl\\_fanuc10l](#), [mdl\\_m16](#), [mdl\\_motomanhp6](#), [mdl\\_S4ABB2p8](#), [mdl\\_puma560](#)

---

# mdl\_irb140\_mdh

## Create model of the ABB IRB 140 manipulator

`mdl_irb140_mod`

Script creates the workspace variable `irb` 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

[SerialLink](#), [mdl\\_irb140](#), [mdl\\_puma560](#), [mdl\\_stanford](#), [mdl\\_twolink](#)

## 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

[SerialLink](#), [mdl\\_mico](#), [mdl\\_puma560](#)

---

## **mdl\_KR5**

### **Create model of Kuka KR5 manipulator**

MDL\_KR5 is a script that creates the workspace variable `mico` 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 11.5cm tool in the z-direction

### **Author**

- Gautam sinha Indian Institute of Technology, Kanpur.

### **See also**

[SerialLink](#), [mdl\\_irb140](#), [mdl\\_fanuc10l](#), [mdl\\_motomanhp6](#), [mdl\\_S4ABB2p8](#), [mdl\\_puma560](#)

---

## **mdl\_m16**

### **Create model of Fanuc M16 manipulator**

MDL\_M16 is a script that creates the workspace variable `mico` 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

## Reference

- “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

[SerialLink](#), [mdl\\_irb140](#), [mdl\\_fanuc10l](#), [mdl\\_motomanhp6](#), [mdl\\_S4ABB2p8](#), [mdl\\_puma560](#)

---

# 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**

[SerialLink](#), [Revolute](#), [mdl\\_jaco](#), [mdl\\_puma560](#), [mdl\\_twolink](#)

---

## mdl\_MotomanHP6

### Create kinematic data of a Motoman HP6 manipulator

MDL\_MotomanHP6 is a script that creates the workspace variable R 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](mailto:wynand.swart@gmail.com)

### Notes

- SI units of metres are used.

**See also**

[SerialLink](#), [mdl\\_irb140](#), [mdl\\_m16](#), [mdl\\_fanuc10l](#), [mdl\\_S4ABB2p8](#), [mdl\\_puma560](#)

---

## 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 et al. 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

[SerialLink](#), [Revolute](#)

---

# mdl\_offset3

## A minimalistic 3DOF robot arm with shoulder offset

MDL\_OFFSET3 is a script that creates the workspace variable off3 which describes the kinematic characteristics of a simple arm manipulator with a shoulder offset, using standard DH conventions.

Somewhat like a Puma arm without the wrist.

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

[SerialLink](#), [mdl\\_offset6](#), [mdl\\_simple6](#), [mdl\\_puma560](#)

---

# 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

[SerialLink](#), [mdl\\_simple6](#), [mdl\\_puma560](#), [mdl\\_twolink](#)

---

# 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

[SerialLink](#), [mdl\\_twolink](#), [mdl\\_planar1](#)

---

# 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, Sec 7.3.4, P. Corke, Springer 2011.

**See also**

[SerialLink](#), [mdl\\_puma560](#)

---

## 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

[SerialLink](#), [mdl\\_onelink](#), [mdl\\_planar2](#), [mdl\\_planar3](#)

---

# 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.

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

[SerialLink](#), [mdl\\_twolink](#), [mdl\\_planar1](#), [mdl\\_planar3](#)

---

# 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

[SerialLink](#), [mdl\\_twolink](#), [mdl\\_planar1](#), [mdl\\_planar2](#)

---

# 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 characteristics 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**

[SerialLink](#), [mdl\\_puma560](#), [mdl\\_stanford\\_mdh](#)

---

## mdl\_S4ABB2p8

### Create kinematic model of ABB S4 2.8robot

MDL\_S4ABB2p8 is a script creates the workspace variable R 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](mailto:wynand.swart@gmail.com)

## **See also**

[SerialLink](#), [mdl\\_fanuc10l](#), [mdl\\_m16](#), [mdl\\_motormanhp6](#), [mdl\\_irb140](#), [mdl\\_puma560](#)

---

# **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**

[SerialLink](#), [mdl\\_offset6](#), [mdl\\_puma560](#)

---

# **mdl\_stanford**

## **Create model of Stanford arm**

[mdl\\_stanford](#)

Script 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

[SerialLink](#), [mdl\\_puma560](#), [mdl\\_puma560akb](#)

---

## mdl\_stanford\_mdh

### Create model of Stanford arm using MDH conventions

`mdl_stanford_mdh`

Script 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

[SerialLink](#), [mdl\\_puma560](#), [mdl\\_puma560akb](#)

---

# mdl\_twolink

## Create model of a 2-link mechanism

MDL\_TWOLINK is a script that creates the workspace variable tl which describes the kinematic and dynamic characteristics of a simple planar 2-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**

[SerialLink](#), [mdl\\_onelink](#), [mdl\\_twolink\\_mdh](#), [mdl\\_planar2](#)

---

## mdl\_twolink\_mdh

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

MDL\_TWOLINK\_MDH is a script that the workspace variable tl 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 XY (horizontal) plane and is therefore not affected by gravity.

### References

- Based on Fig 3.8 (p71) of Craig (3rd edition).

**See also**

[SerialLink](#), [mdl\\_onelink](#), [mdl\\_twolink](#), [mdl\\_planar2](#)

---

## mstraj

### Multi-segment multi-axis trajectory

`traj = mstraj(p, 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 **p** ( $M \times N$ ), and finish at the point defined by the last row of **p**. 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.

- **p** ( $M \times N$ ) 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** ( $1 \times 1$ ) this acceleration time is applied to all segment transitions
- **tacc** ( $1 \times M$ ) acceleration time for each 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** = **mtraj**(**segments**, **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** should 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.

## 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 ( $\mathbf{m} \times N$ ) 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** ( $\mathbf{m} \times N$ ) and **qdd** ( $\mathbf{m} \times N$ ) 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**

```
[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** ( $\mathbf{m} \times 1$ ) 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](#)

---

## multidprintf

### Print formatted text to multiple streams

COUNT = MULTIDPRINTF(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 fprintf.

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 console 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:  
multidprintf([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

plot	Display the occupancy grid
visualize	Display the occupancy grid (deprecated)
plan	Plan a path to goal
path	Return/animate a path from start to goal
display	Display the parameters in human readable form
char	Convert to string
rand	Uniformly distributed random number
randn	Normally distributed random number
randi	Uniformly distributed random integer

## Properties (read only)

occgrid Occupancy grid representing the navigation environment  
goal Goal 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 overridden 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

[Dstar](#), [dxform](#), [PRM](#), [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 passed.

## Options

‘navhook’, F	Specify a function to be called at every step of path
‘goal’, G	Specify the goal point ( $2 \times 1$ )
‘verbose’	Display debugging information
‘inflate’, K	Inflate all obstacles by K cells.
‘private’	Use private random number stream.
‘reset’	Reset random number stream.
‘seed’, S	Set the initial state of the random number stream. S must be a proper random number generator state such as saved in 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.
  - The ‘private’ option creates a private random number stream for the methods rand, randn and randi. If not given the global stream is used.
- 

# 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 overridden in a subclass to take particular action such as invalidating a costmap.

---

## 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 path() method is invoked. Typically overridden in a subclass to take particular action such as computing some path parameters. start is the initial position for this path, and nav.goal is the final position.

---

## Navigation.path

### Follow path from start to goal

N.path(start) animates the robot moving from start ( $2 \times 1$ ) to the goal (which is a property of the object).

N.path() as above but first displays the occupancy grid, and prompts the user to click a start location. the object).

x = N.path(start) returns the path ( $2 \times M$ ) from start to the goal (which is a property of the object).

The method performs the following steps:

- Get start position interactively if not given
- 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.

### **See also**

[Navigation.plot](#), [Navigation.goal](#)

---

## **Navigation.plot**

### **Visualize navigation environment**

`N.plot()` displays the occupancy grid in a new figure.

`N.plot(p)` as above but overlays the points along the path ( $2 \times M$ ) matrix.

### **Options**

<code>'goal'</code>	Superimpose the goal position if set
<code>'distance'</code> , D	Display a distance field D behind the obstacle map. D is a matrix of the same size as the occupancy grid.

### **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.
- 

## **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 ( $m \times m$ ) of random numbers.

`R = N.rand(L,m)` as above but return a matrix ( $L \times m$ ) 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** × **m**) of random integers.

**i = N.randn(rm, L,m)** as above but returns a matrix (**L** × **m**) of random integers.

## 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.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** × **m**) of random numbers.

**R = N.randn(L,m)** as above but returns a matrix ( $L \times m$ ) 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.

---

## 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 ( $3 \times 3$ ) for the specified orientation and approach vectors ( $3 \times 1$ ) formed from 3 vectors such that **R** = [N o a] and N = o x a.

**Notes**

- The 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: mathematis, programming and control Richard Paul, MIT Press, 1981.

**See also**

[rpy2r](#), [eul2r](#), [oa2r](#)

---

## oa2tr

### Convert orientation and approach vectors to homogeneous transformation

$T = \text{oa2tr}(\mathbf{o}, \mathbf{a})$  is an SE(3) homogeneous transformation ( $4 \times 4$ ) for the specified orientation and approach vectors ( $3 \times 1$ ) formed from 3 vectors such that  $R = [N \ \mathbf{o} \ \mathbf{a}]$  and  $N = \mathbf{o} \times \mathbf{a}$ .

### Notes

- The rotation submatrix is guaranteed to be orthonormal so long as  $\mathbf{o}$  and  $\mathbf{a}$  are not parallel.
- The translational part is zero.
- The vectors  $\mathbf{o}$  and  $\mathbf{a}$  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**

[rpy2tr](#), [eul2tr](#), [oa2r](#)

---

## 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	vector of structs that hold the detailed information from each time step
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 = Map(20);
```

and a vehicle with odometry covariance and a driver

```
W = diag([0.1, 1*pi/180].^2);
veh = Vehicle(W);
veh.add_driver(RandomPath(10));
```

and create a range bearing sensor

```
R = diag([0.005, 0.5*pi/180].^2);
sensor = RangeBearingSensor(veh, map, R);
```

For the particle filter we need to define two covariance matrices. The first 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](#), [Map](#), [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	Set the initial state of the random number stream. S must be a proper random number generator state such as saved in 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](#), [Map](#)

---

# 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

‘nopl’ Do not show animation.

### Notes

- All previously estimated states and estimation history is cleared.
- 

## peak

### Find peaks in vector

yp = **peak**(y, options) are the values of the maxima in the vector y.

[yp,i] = **peak**(y, options) as above but also returns the indices of the maxima in the vector y.

[yp,xp] = **peak**(y, x, options) as above but also returns the corresponding x-coordinates of the maxima in the vector y. x is the same length as y and contains the corresponding x-coordinates.

### Options

‘npeaks’, N Number of peaks to return (default all)  
‘scale’, S Only consider as peaks the largest value in the horizontal range +/- S points.  
‘interp’, M Order of interpolation polynomial (default no interpolation)  
‘plot’ Display the interpolation polynomial overlaid on the point data

### Notes

- A maxima is defined as an element that larger than its two neighbours. The first and last element will never be returned as maxima.
- To find minima, use **peak**(-V).

- The interp options fits points in the neighbourhood about the **peak** with an M'th order polynomial and its **peak** position is returned. Typically choose M to be odd. In this case **xp** will be non-integer.

## See also

[peak2](#)

---

# peak2

## Find peaks in a matrix

**zp = peak2(z, options)** are the peak values in the 2-dimensional signal **z**.

**[zp,ij] = peak2(z, options)** as above but also returns the indices of the maxima in the matrix **z**. Use SUB2IND to convert these to row and column coordinates

## Options

‘npeaks’, N	Number of peaks to return (default all)
‘scale’, S	Only consider as peaks the largest value in the horizontal and vertical range +/- S points.
‘interp’	Interpolate peak (default no interpolation)
‘plot’	Display the interpolation polynomial overlaid on the point data

## Notes

- A maxima is defined as an element that larger than its eight neighbours. Edges elements will never be returned as maxima.
- To find minima, use **peak2(-V)**.
- The interp options fits points in the neighbourhood about the peak with a paraboloid and its peak position is returned. In this case **ij** will be non-integer.

## See also

[peak](#), [sub2ind](#)

---

# 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
- have symmetric cost edges (A to B is same cost as B to A)
- have no loops (edges from A to A)
- have vertices that are represented by integers VID
- have edges that are represented by integers EID

## Methods

### Constructing the graph

<code>g.add_node(coord)</code>	add vertex, return vid
<code>g.add_edge(v1, v2)</code>	add edge from v1 to v2, return eid
<code>g.setcost(e, c)</code>	set cost for edge e
<code>g.setdata(v, u)</code>	set user data for vertex v
<code>g.data(v)</code>	get user data for vertex v
<code>g.clear()</code>	remove all vertices and edges from the graph

### Information from graph

<code>g.edges(v)</code>	list of edges for vertex v
<code>g.cost(e)</code>	cost of edge e
<code>g.neighbours(v)</code>	neighbours of vertex v
<code>g.component(v)</code>	component id for vertex v
<code>g.connectivity()</code>	number of edges for all vertices

## Display

<code>g.plot()</code>	set goal vertex for path planning
<code>g.highlight_node(v)</code>	highlight vertex v
<code>g.highlight_edge(e)</code>	highlight edge e
<code>g.highlight_component(c)</code>	highlight all nodes in component c
<code>g.highlight_path(p)</code>	highlight nodes and edge along path p
<code>g.pick(coord)</code>	vertex closest to coord

g.char() convert graph to string  
g.display() display summary of graph

## Matrix representations

g.adjacency() adjacency matrix  
g.incidence() incidence matrix  
g.degree() degree 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.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  
g.closest(coord) vertex closest to coord

## Object properties (read only)

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

## Examples

```
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

- Graph connectivity is maintained by a labeling algorithm and this is updated every time an edge is added.

- Nodes and edges cannot be deleted.
  - Support for edge direction is rudimentary.
- 

## PGraph.PGraph

### Graph class constructor

`g=PGraph(d, options)` is a graph object embedded in `d` dimensions.

### Options

'distance', M	Use the distance metric M for path planning which is either 'Euclidean' (default) or 'SE2'.
'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 2pi.
  - 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

- Distance is computed according to the metric specified in the constructor.
- Graph connectivity is maintained by a labeling algorithm and this is updated every time an edge is added.

### See also

[PGraph.add\\_node](#), [PGraph.edgedir](#)

---

## PGraph.add\_node

### Add a node

$v = G.\text{add\_node}(x)$  adds a node/vertex with coordinate  $x$  ( $D \times 1$ ) and returns the integer node id  $v$ .

$v = G.\text{add\_node}(x, v2)$  as above but connected by a directed edge from vertex  $v$  to vertex  $v2$  with cost equal to the distance between the vertices.

$v = G.\text{add\_node}(x, v2, C)$  as above but the added edge has 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

$a = G.\text{adjacency}()$  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  $a^N$  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.

### 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.connectivity

### Graph 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.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.data

### 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.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.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.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.goal

### Set goal node

G.goal(vg) computes the cost of reaching every vertex in the graph connected to the **goal** vertex vg.

### Notes

- Combined with G.path performs a breadth-first search for paths to the **goal**.

### See also

[PGraph.path](#), [PGraph.Astar](#), [astar](#)

---

## 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)

### 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.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, and negative if the link is from the node to v.

[**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.path

### Find path to goal node

**p** = G.**path**(vs) is a vector of vertex ids that form a **path** from the starting vertex vs to the previously specified goal. The **path** includes the start and goal vertex id.

To compute **path** to goal vertex 5

```
g.goal(5);
```

then the **path**, starting from vertex 1 is

```
p1 = g.path(1);
```

and the **path** starting from vertex 2 is

```
p2 = g.path(2);
```

## Notes

- Pgraph.goal must have been invoked first.
- Can be used repeatedly to find paths from different starting points to the goal specified to Pgraph.goal().

## See also

[PGraph.goal](#), [PGraph.Astar](#)

---

# 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

## PGraph.setcost

### Set cost of edge

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

---

## PGraph.setdata

### 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.vertices

### Find vertices given edge

v = G.**vertices**(E) return the id of the **vertices** that define edge E.

---

## plot2

### Plot trajectories

**plot2(p)** plots a line with coordinates taken from successive rows of p. p can be  $N \times 2$  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**

[plot](#)

---

## plot\_arrow

### Draw an arrow

**plot\_arrow(p, options)** draws an arrow from P1 to P2 where **p=[P1; P2]**.

### Options

All options are passed through to arrow3. Pass in a single character MATLAB color-spec (eg. ‘r’) to set the color.

**See also**

[arrow3](#)

---

## plot\_box

### a box

**plot\_box(b, ls)** 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, ls)** draws a box with corners at **(x1,y1)** and **(x2,y2)**, and optional MATLAB linestyle options **ls**.

**plot\_box(‘centre’, P, ‘size’, W, ls)** draws a box with center at **P=[X,Y]** and with dimensions **W=[WIDTH HEIGHT]**.

**plot\_box(‘topleft’, P, ‘size’, W, ls)** draws a box with top-left at **P=[X,Y]** and with dimensions **W=[WIDTH HEIGHT]**.

### Notes

- The box is added to the current plot.
- Additional options **ls** are MATLAB LineSpec options and are passed to PLOT.

**See also**

[plot\\_poly](#), [plot\\_circle](#), [plot\\_ellipse](#)

---

## plot\_circle

### Draw a circle

**plot\_circle(C, 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.

**H = plot\_circle(C, R, options)** as above but return handles. For multiple circles **H** is a vector of handles, one per circle.

If **C** ( $2 \times N$ ) then N circles are drawn and **H** is  $N \times 1$ . If **R** ( $1 \times 1$ ) then all circles have the same radius or else **R** ( $1 \times N$ ) to specify the radius of each 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’, <b>H</b>	alter existing circles with handle <b>H</b>

For an unfilled ellipse any MATLAB LineProperty **options** can be given, for a filled ellipse any MATLAB PatchProperty **options** can be given.

### Notes

- The circle(s) is added to the current plot.

**See also**

[plot\\_ellipse](#), [plot\\_box](#), [plot\\_poly](#)

---

## plot\_ellipse

### Draw an ellipse or ellipsoid

**plot\_ellipse(a, options)** draws an ellipse defined by  $X'AX = 0$  on the current plot, centred at the origin.

**plot\_ellipse(a, 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(a, C, options)** as above but return graphic handle.

### 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

### Notes

- If  $a$  ( $2 \times 2$ ) draw an ellipse, else if  $a(3 \times 3)$  draw an ellipsoid.
- The ellipse is added to the current plot.

### See also

[plot\\_ellipse\\_inv](#), [plot\\_circle](#), [plot\\_box](#), [plot\\_poly](#)

---

## plot\_ellipse\_inv

### Draw an ellipse or ellipsoid

**plot\_ellipse\_inv(a, options)** draws an ellipse defined by  $X'.inv(a).X = 0$  on the current plot, centred at the origin.

**plot\_ellipse\_inv(a, 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\_inv(a, C, options)** as above but return graphic handle.

## 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', <b>H</b>	alter existing circles with handle <b>H</b>

## Notes

- For the case where the inverse of ellipse parameters are known, perhaps an inverse covariance matrix.
- If **a** ( $2 \times 2$ ) draw an ellipse, else if **a** ( $3 \times 3$ ) draw an ellipsoid.
- The ellipse is added to the current plot.

## See also

[plot\\_ellipse](#), [plot\\_circle](#), [plot\\_box](#), [plot\\_poly](#)

---

# plot\_homline

## Draw a homogeneous

**plot\_homline(L, ls)** draws a line in the current plot  $L.X = 0$  where **L** ( $3 \times 1$ ). The current axis limits are used to determine the endpoints of the line. MATLAB line specification **ls** can be set. If **L** ( $3 \times N$ ) then  $N$  lines are drawn, one per column.

**H = plot\_homline(L, ls)** as above but returns a vector of graphics handles for the lines.

## Notes

- The line(s) is added to the current plot.

## See also

[plot\\_box](#), [plot\\_poly](#), [homline](#)

---

# plot\_point

## a feature point

**plot\_point(p, options)** adds point markers to the current plot, where **p** ( $2 \times N$ ) and each column is the point coordinate.

## Options

‘textcolor’, colspec	Specify color of text
‘textsize’, size	Specify size of text
‘bold’	Text in bold font.
‘printf’, {fmt, data}	Label points according to printf format string and corresponding element of data
‘sequence’	Label points sequentially

Additional options are passed through to PLOT for creating the marker.

## Examples

Simple point plot

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

Plot points with markers

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

Plot points with square markers and labels 1 to 4

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

Plot points with circles and annotations P1 to P4

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

## Notes

- The point(s) is added to the current plot.
- 2D points only.

## See also

[plot](#), [text](#)

---

## plot\_poly

### Draw a polygon

**plot\_poly(p, options)** draws a polygon defined by columns of **p** ( $2 \times N$ ), in the current plot.

### options

‘fill’, F      the color of the circle’s interior, MATLAB color spec  
‘alpha’, A      transparency of the filled circle: 0=transparent, 1=solid.

### Notes

- If **p** ( $3 \times N$ ) the polygon is drawn in 3D
- The line(s) is added to the current plot.

### See also

[plot\\_box](#), [patch](#), [Polygon](#)

---

## plot\_sphere

### Draw sphere

**plot\_sphere(C, R, ls)** draws spheres in the current plot. **C** is the centre of the sphere ( $3 \times 1$ ), **R** is the radius and **ls** is an optional MATLAB color spec, either a letter or a 3-vector.

**H = plot\_sphere(C, R, color)** as above but returns the handle(s) for the spheres.

**H = plot\_sphere(C, R, color, alpha)** as above but **alpha** specifies the opacity of the sphere were 0 is transparent 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** ( $1 \times 1$ ) then all spheres have the same radius or else **R** ( $1 \times N$ ) to specify the radius of each sphere.

## Example

Create four spheres

```
plot_sphere( mgrid(2, 1), .2, 'b')
```

and now turn on a full lighting model

```
lighting gouraud  
light
```

## NOTES

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

# plot\_vehicle

## Draw ground vehicle pose

**plot\_vehicle**(*x,options*) draws a representation of ground robot as an oriented triangle with pose *x* ( $1 \times 3$ ) = [x,y,theta] or *x* ( $3 \times 3$ ) as an SE(2) homogeneous transform.

## Options

‘scale’, *S*    Draw vehicle with length *S* x maximum axis dimension (default 1/60)  
‘size’, *S*    Draw vehicle with length *S*

## See also

[Vehicle.plot](#)

---

# 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](#)

---

## plotp

### Plot trajectories

**plotp(p)** plots a set of points **p**, which by Toolbox convention are stored one per column. **p** can be  $N \times 2$  or  $N \times 3$ . 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** ( $1 \times N$ ).

**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 polygon 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/glen/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 polygon 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) homogenous transformation **T** ( $3 \times 3$ ).

---

## 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”.
-

# Prismatic

## Robot manipulator prismatic link class

A subclass of the Link class: holds all information related to a prismatic (sliding) robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

### Notes

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

### References

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

### See also

[Link](#), [Revolute](#), [SerialLink](#)

---

# PrismaticMDH

## Robot manipulator prismatic link class for MDH convention

A subclass of the Link class: holds all information related to a prismatic (sliding) robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

### Notes

- 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, Chap 7 P. Corke, Springer 2011.

## See also

[Link](#), [Prismatic](#), [RevoluteMDH](#), [SerialLink](#)

---

# 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

plan	Compute the roadmap
path	Compute a path to the goal
visualize	Display the obstacle map (deprecated)
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.path(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` Distance threshold, edges only connect vertices closer than D (default 0.3  
max(size(occgrid)))

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.path

### Find a path between two points

**P.path(start, goal)** finds and displays a **path** from **start** to **goal** which is overlaid on the occupancy grid.

**x = P.path(start)** returns the **path** ( $2 \times M$ ) from **start** to **goal**.

---

## PRM.plan

### Create a probabilistic roadmap

P.**plan**() 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.

---

## PRM.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 PRM graph          |
- 

## 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](#), [plot](#)

---

# Quaternion

## Quaternion class

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

A unit-quaternion 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)>
```

**q = quaternion(x)** is a unit-**quaternion** equivalent to **x** which can be any of:

- orthonormal rotation matrix.
- homogeneous transformation matrix (rotation part only).
- rotation angle and vector

## Methods

inv	inverse of quaternion
norm	norm of <b>quaternion</b>
unit	unitized <b>quaternion</b>
plot	same options as trplot()
interp	interpolation (slerp) between q and q2, $0 \leq s \leq 1$
scale	interpolation (slerp) between identity and q, $0 \leq s \leq 1$
dot	derivative of <b>quaternion</b> with angular velocity w
R	equivalent $3 \times 3$ rotation matrix
T	equivalent $4 \times 4$ homogeneous transform matrix
double	<b>quaternion</b> elements as 4-vector
inner	inner product of two quaternions

## Overloaded operators

q1==q2	test for <b>quaternion</b> equality
q1 =q2	test for <b>quaternion</b> inequality
q+q2	elementwise sum of quaternions
q-q2	elementwise difference of quaternions
q*q2	<b>quaternion</b> product
q*v	rotate vector by <b>quaternion</b> , v is $3 \times 1$
s*q	elementwise multiplication of <b>quaternion</b> by scalar
q/q2	q*q2.inv
q <sup>n</sup>	q to power n (integer only)

## Properties (read only)

s real part  
v vector part

## Notes

- **quaternion** objects can be used in vectors and arrays.

## References

- Animating rotation with **quaternion** curves, K. Shoemake, in Proceedings of ACM SIGGRAPH, (San Francisco), 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.
- Robotics, Vision & Control, P. Corke, Springer 2011.

## See also

[trinterp](#), [trplot](#)

---

# Quaternion.Quaternion

## Constructor for **quaternion** objects

Construct a **quaternion** from various other orientation representations.

**q = Quaternion()** is the identity unit-quaternion  $1<0,0,0>$  representing a null rotation.

**q = Quaternion(q1)** is a copy of the quaternion **q1**

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

**q = Quaternion(s)** is a quaternion formed from the scalar **s** and zero vector part:  
 $s<0,0,0>$

**q = Quaternion(v)** is a pure quaternion with the specified vector part:  $0<\mathbf{v}>$

**q = Quaternion(th, v)** is a unit-quaternion corresponding to rotation of **th** about the vector **v**.

**q = Quaternion(R)** is a unit-quaternion corresponding to the SO(3) orthonormal rotation matrix **R** ( $3 \times 3$ ). 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 unit-quaternion equivalent to the rotational part of the SE(3) homogeneous transform **T** ( $4 \times 4$ ). 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**.

---

## 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.

---

## 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.

### See also

[Quaternion.char](#)

---

## Quaternion.dot

### Quaternion derivative

**qd = Q.dot(omega)** is the rate of change of a frame with attitude Q and angular velocity OMEGA ( $1 \times 3$ ) expressed as a quaternion.

---

## Quaternion.double

### Convert a quaternion to a 4-element vector

`v = Q.double()` is a 4-vector comprising the quaternion elements [s vx vy vz].

---

## Quaternion.eq

### Test quaternion equality

`Q1==Q2` is true if the quaternions `Q1` and `Q2` are equal.

### Notes

- Overloaded operator ‘`==`’.
- Note that for unit Quaternions `Q` and `-Q` are the equivalent rotation, so non-equality does not mean rotations are not equivalent.
- If `Q1` is a vector of quaternions, 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 quaternions, each element is compared to `Q1` and the result is a logical array of the same length as `Q2`.
- If `Q1` and `Q2` are vectors of the same length, then the result is a logical array of the same length.

### See also

[Quaternion.ne](#)

---

## Quaternion.inner

### Quaternion inner product

`v = Q1.inner(q2)` is the **inner** (dot) product of two vectors ( $1 \times 4$ ), comprising the elements of `Q1` and `q2` respectively.

### Notes

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

**See also**

[Quaternion.norm](#)

---

## Quaternion.interp

### Interpolate quaternions

**qi** = Q1.**interp**(q2, s) is a unit-quaternion that interpolates a rotation between Q1 for s=0 and q2 for s=1.

If s is a vector **qi** is a vector of quaternions, each element corresponding to sequential elements of s.

### Notes

- This is a spherical linear interpolation (slerp) that can be interpreted as interpolation along a great circle arc on a sphere.
- The value of s is clipped to 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**

[Quaternion.scale](#), [ctraj](#)

---

## Quaternion.inv

### Invert a unit-quaternion

**qi** = Q.**inv**() is a quaternion object representing the inverse of Q.

---

## Quaternion.minus

### Subtract quaternions

Q1-Q2 is the element-wise difference of quaternion elements.

## Notes

- Overloaded operator ‘-’
- The result is not guaranteed to be a unit-quaternion.

## See also

[Quaternion.plus](#), [Quaternion.mtimes](#)

---

# Quaternion.mpower

## Raise quaternion to integer power

$Q^N$  is the quaternion Q raised to the integer power N.

## Notes

- Overloaded operator ‘^’
- Computed by repeated multiplication.
- If the argument is a unit-quaternion, the result will be a unit quaternion.

## See also

[Quaternion.mrdivide](#), [Quaternion.mpower](#), [Quaternion.plus](#), [Quaternion.minus](#)

---

# Quaternion.mrdivide

## Quaternion quotient.

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

## Notes

- Overloaded operator ‘/’
- If the dividend and divisor are unit-quaternions, the quotient will be a unit quaternion.

**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.
- $Q * V$  is a vector formed by rotating the vector  $V$  by the quaternion  $Q$ .
- $Q * S$  is the element-wise multiplication of quaternion elements by the scalar  $S$ .

**Notes**

- Overloaded operator ‘\*’
- If the two multiplicands are unit-quaternions, the product will be a unit quaternion.

**See also**

[Quaternion.mrdivide](#), [Quaternion.mpower](#), [Quaternion.plus](#), [Quaternion.minus](#)

---

## Quaternion.ne

### Test quaternion inequality

$Q1 = Q2$  is true if the quaternions  $Q1$  and  $Q2$  are not equal.

**Notes**

- Overloaded operator ‘=’
- Note that for unit Quaternions  $Q$  and  $-Q$  are the equivalent rotation, so non-equality does not mean rotations are not equivalent.
- If  $Q1$  is a vector of quaternions, 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 quaternions, each element is compared to  $Q1$  and the result is a logical array of the same length as  $Q2$ .
- If  $Q1$  and  $Q2$  are vectors of the same length, then the result is a logical array of the same length.

**See also**

[Quaternion.eq](#)

---

## Quaternion.norm

### Quaternion magnitude

**qn = 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.

**See also**

[Quaternion.inner](#), [Quaternion.unit](#)

---

## Quaternion.plot

### Plot a quaternion object

**Q.plot(options)** plots the quaternion as an oriented coordinate frame.

### Options

Options are passed to trplot and include:

- |            |  |
|------------|--|
| ‘color’, C | The color to draw the axes, MATLAB colorspec C   |
| ‘frame’, F | The frame is named {F} and the subscript on the axis labels is F.                                      |
| ‘view’, V  | Set <b>plot</b> view parameters V=[az el] angles, or ‘auto’ for view toward origin of coordinate frame |

**See also**

[trplot](#)

---

## Quaternion.plus

### Add quaternions

`Q1+Q2` is the element-wise sum of quaternion elements.

### Notes

- Overloaded operator ‘+’
- The result is not guaranteed to be a unit-quaternion.

### See also

[Quaternion.minus](#), [Quaternion.mtimes](#)

---

## Quaternion.R

### Convert to orthonormal rotation matrix

`R = Q.R()` is the equivalent SO(3) orthonormal rotation matrix ( $3 \times 3$ ). If `Q` represents a sequence ( $N \times 1$ ) then `R` is  $3 \times 3 \times N$ .

---

## Quaternion.scale

### Interpolate rotations expressed by quaternion objects

`qi = Q.scale(s)` is a unit-quaternion that interpolates between a null rotation (identity quaternion) for `s=0` to `Q` for `s=1`. This is a spherical linear interpolation (slerp) that can be interpreted as interpolation along a great circle arc on a sphere.

If `s` is a vector `qi` is a vector of quaternions, each element corresponding to sequential elements of `s`.

### Notes

- This is a spherical linear interpolation (slerp) that can be interpreted as interpolation along a great circle arc on a sphere.

**See also**

[Quaternion.interp](#), [ctraj](#)

---

## Quaternion.T

### Convert to homogeneous transformation matrix

**T = Q.T()** is the equivalent SE(3) homogeneous transformation matrix ( $4 \times 4$ ). If Q represents a sequence ( $N \times 1$ ) then T is  $4 \times 4 \times N$ .

Notes:

- Has a zero translational component.
- 

## Quaternion.unit

### Unitize a quaternion

**qu = Q.unit()** is a **unit**-quaternion representing the same orientation as Q.

**See also**

[Quaternion.norm](#)

---

## 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.

**Notes**

- Works for T in either SE(2) or SE(3)
  - if R is  $2 \times 2$  then T is  $3 \times 3$ , or
  - if R is  $3 \times 3$  then T is  $4 \times 4$ .

- Translational component is zero.
- For a rotation matrix sequence returns a homogeneous transform sequence.

**See also**

[t2r](#)

---

## randinit

### Reset random number generator

RANDINIT resets the default random number stream.

**See also**

[randstream](#)

---

## RandomPath

### Vehicle driver class

Create a “driver” object capable of steering a Vehicle 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	return speed and steer angle to next waypoint
display	display the state and parameters in human readable form
char	convert to string

## Properties

goal	current goal/waypoint coordinate
veh	the Vehicle object being controlled
dim	dimensions of the work space ( $2 \times 1$ ) [m]
speed	speed of travel [m/s]
closeenough	proximity to waypoint at which next is chosen [m]

## Example

```
veh = Vehicle(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](#)

---

# RandomPath.RandomPath

## Create a driver object

**d = RandomPath(dim, options)** returns a “driver” object capable of driving a Vehicle object through random waypoints. The waypoints are positioned inside a rectangular region bounded by +/- **dim** in the x- and y-directions.

## 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 a compact human readable format.

---

# RandomPath.demand

## Compute speed and heading to waypoint

[**speed,steer**] = R.**demand**() returns the speed and steer angle to drive the vehicle toward the next waypoint. When the vehicle is within R.closeenough 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.

## 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.

### 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 point features in the world. To enable this it has references to a map of the world (Map object) and a robot moving through the world (Vehicle object).

### Methods

reading	range/bearing observation of random feature
h	range/bearing observation of specific feature
Hx	Jacobian matrix dh/dxv
Hxf	Jacobian matrix dh/dxf
Hw	Jacobian matrix dh/dw
g	feature position given vehicle pose and observation
Gx	Jacobian matrix dg/dxv
Gz	Jacobian matrix dg/dz

### Properties (read/write)

W	measurement covariance matrix ( $2 \times 2$ )
interval	valid measurements returned every interval'th call to reading()

### Reference

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

**See also**

[Sensor](#), [Vehicle](#), [Map](#), [EKF](#)

---

## RangeBearingSensor.RangeBearingSensor

### Range and bearing sensor constructor

**s = RangeBearingSensor(vehicle, map, w, options)** is an object representing a range and bearing angle sensor mounted on the Vehicle object **vehicle** and observing an environment of known landmarks represented by the map object **map**. The sensor covariance is **w** ( $2 \times 2$ ) representing range and bearing covariance.

### Options

‘range’, xmax	maximum range of sensor
‘range’, [xmin xmax]	minimum and maximum range of sensor
‘angle’, TH	detection for angles between -TH to +TH
‘angle’, [THMIN THMAX]	detection for angles between THMIN and THMAX
‘skip’, I	return a valid reading on every I’th call
‘fail’, [TMIN TMAX]	sensor simulates failure between timesteps TMIN and TMAX

**See also**

[options for sensor constructor](#)

**See also**

[Sensor.Sensor](#), [Vehicle](#), [Map](#), [EKF](#)

---

## RangeBearingSensor.g

### Compute landmark location

**p = S.g(xv, z)** is the world coordinate ( $1 \times 2$ ) of a feature given the sensor observation **z** ( $1 \times 2$ ) and vehicle state **xv** ( $3 \times 1$ ).

**See also**

[RangeBearingSensor.Gx](#), [RangeBearingSensor.Gz](#)

---

## RangeBearingSensor.Gx

### Jacobian dg/dx

**J** = S.**Gx**(**xv**, **z**) is the Jacobian dg/dxv ( $2 \times 3$ ) at the vehicle state **xv** ( $3 \times 1$ ) for sensor observation **z** ( $2 \times 1$ ).

### See also

[RangeBearingSensor.g](#)

---

## RangeBearingSensor.Gz

### Jacobian dg/dz

**J** = S.**Gz**(**xv**, **z**) is the Jacobian dg/dz ( $2 \times 2$ ) at the vehicle state **xv** ( $3 \times 1$ ) for sensor observation **z** ( $2 \times 1$ ).

### See also

[RangeBearingSensor.g](#)

---

## RangeBearingSensor.h

### Landmark range and bearing

**z** = S.**h**(**xv**, **k**) is a sensor observation ( $1 \times 2$ ), range and bearing, from vehicle at pose **xv** ( $1 \times 3$ ) to the **k**'th map feature.

**z** = S.**h**(**xv**, **xf**) as above but compute range and bearing to a feature at coordinate **xf**.

**z** = s.**h**(**xv**) as above but computes range and bearing to all map features. **z** has one row per feature.

### Notes

- Noise with covariance **W** (property**W**) is added to each row of **z**.
- Supports vectorized operation where **xv** ( $N \times 3$ ) and **z** ( $N \times 2$ ).

**See also**

[RangeBearingSensor.Hx](#), [RangeBearingSensor.Hw](#), [RangeBearingSensor.Hxf](#)

---

## RangeBearingSensor.Hw

**Jacobian dh/dv**

**J** = S.**Hw**(**xv**, **k**) is the Jacobian dh/dv ( $2 \times 2$ ) at the vehicle state **xv** ( $3 \times 1$ ) for map feature **k**.

**See also**

[RangeBearingSensor.h](#)

---

## RangeBearingSensor.Hx

**Jacobian dh/dxv**

**J** = S.**Hx**(**xv**, **k**) returns the Jacobian dh/dxv ( $2 \times 3$ ) at the vehicle state **xv** ( $3 \times 1$ ) for map feature **k**.

**J** = S.**Hx**(**xv**, **xf**) as above but for a feature at coordinate **xf**.

**See also**

[RangeBearingSensor.h](#)

---

## RangeBearingSensor.Hxf

**Jacobian dh/dxf**

**J** = S.**Hxf**(**xv**, **k**) is the Jacobian dh/dxv ( $2 \times 2$ ) at the vehicle state **xv** ( $3 \times 1$ ) for map feature **k**.

**J** = S.**Hxf**(**xv**, **xf**) as above but for a feature at coordinate **xf** ( $1 \times 2$ ).

**See also**

[RangeBearingSensor.h](#)

---

## RangeBearingSensor.reading

### Landmark range and bearing

`[z,k] = S.reading()` is an observation of a random 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. If no valid measurement, ie. no features within range, interval subsampling enabled or simulated failure the return is `z=[]` and `k=NaN`.

### See also

[RangeBearingSensor.h](#)

---

## Revolute

### Robot manipulator Revolute link class

A subclass of the Link class: holds all information related to a robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

### Notes

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

### References

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

### See also

[Link](#), [Prismatic](#), [SerialLink](#)

---

## RevoluteMDH

### Robot manipulator Revolute link class for MDH convention

A subclass of the Link class: holds all information related to a robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

#### Notes

- 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, Chap 7 P. Corke, Springer 2011.

#### See also

[Link](#), [Prismatic](#), [SerialLink](#)

#### See also

[Link](#), [PrismaticMDH](#), [Revolute](#), [SerialLink](#)

---

---

## 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 ( $4 \times 4$ ).

### 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](#)

---

## rot2

### SO(2) Rotation matrix

R = **rot2(theta)** is an SO(2) rotation matrix representing a rotation of **theta** radians.

R = **rot2(theta, ‘deg’)** as above but **theta** is in degrees.

### See also

[trot2](#), [rotx](#), [roty](#), [rotz](#)

---

## rotx

### Rotation about X axis

R = **rotx(theta)** is an SO(3) rotation matrix ( $3 \times 3$ ) representing a rotation of **theta** radians about the x-axis.

R = **rotx(theta, ‘deg’)** as above but **theta** is in degrees.

### See also

[roty](#), [rotz](#), [angvec2r](#), [rot2](#)

---

## roty

### Rotation about Y axis

**R = roty(theta)** is an SO(3) rotation matrix ( $3 \times 3$ ) representing a rotation of **theta** radians about the y-axis.

**R = roty(theta, ‘deg’)** as above but **theta** is in degrees.

### See also

[rotx](#), [rotz](#), [angvec2r](#), [rot2](#)

---

## rotz

### Rotation about Z axis

**R = rotz(theta)** is an SO(3) rotation matrix ( $3 \times 3$ ) representing a rotation of **theta** radians about the z-axis.

**R = rotz(theta, ‘deg’)** as above but **theta** is in degrees.

### See also

[rotx](#), [roty](#), [angvec2r](#), [rot2](#)

---

## rpy2jac

### Jacobian from RPY angle rates to angular velocity

**J = rpy2jac(eul)** is a Jacobian matrix ( $3 \times 3$ ) that maps roll-pitch-yaw angle rates to angular velocity at the operating point RPY=[R,P,Y].

**J = rpy2jac(R, p, y)** as above but the roll-pitch-yaw angles are passed as separate arguments.

## Notes

- Used in the creation of an analytical Jacobian.

## See also

[eul2jac](#), [SerialLink.JACOBN](#)

---

# rpy2r

## Roll-pitch-yaw angles to rotation matrix

**R = rpy2r(roll, pitch, yaw, options)** is an SO(3) orthonormal rotation matrix ( $3 \times 3$ ) equivalent to the specified roll, pitch, yaw angles angles. These correspond to rotations about the X, Y, Z 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 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 **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)  
‘zyx’ Return solution for sequential rotations about Z, Y, X axes (Paul book)

## Note

- In previous releases (<8) the angles corresponded to rotations about ZYX. Many texts (Paul, Spong) use the rotation order ZYX. This old behaviour can be enabled by passing the option ‘zyx’

## See also

[tr2rpy](#), [eul2tr](#)

---

## rpy2tr

### Roll-pitch-yaw angles to homogeneous transform

**T = rpy2tr(roll, pitch, yaw, options)** is an SE(3) homogeneous transformation matrix ( $4 \times 4$ ) equivalent to the specified roll, pitch, yaw angles angles. These correspond to rotations about the X, Y, Z 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 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 **T** 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)
- ‘zyx’ Return solution for sequential rotations about Z, Y, X axes (Paul book)

### Note

- In previous releases (<8) the angles corresponded to rotations about ZYX. Many texts (Paul, Spong) use the rotation order ZYX. This old behaviour can be enabled by passing the option ‘zyx’

### See also

[tr2rpy](#), [rpy2r](#), [eul2tr](#)

---

## 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

plan	Compute the tree
path	Compute a path
plot	Display the tree
display	Display the parameters in human readable form
char	Convert to string

## Example

```
goal = [0,0,0];
start = [0,2,0];
veh = Vehicle([], 'stlim', 1.2);
rrt = RRT([], veh, 'goal', goal, 'range', 5);
rrt.plan()           % create navigation tree
rrt.path(start, goal) % animate path from this start location
```

Robotics, Vision & Control compatibility mode:

```
goal = [0,0,0];
start = [0,2,0];
rrt = RRT();           % create navigation object
rrt.plan()           % create navigation tree
rrt.path(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(map, veh, options)** is a rapidly exploring tree navigation object for a region with obstacles defined by the map object **map**.

**R = RRT.RRT()** as above but internally creates a Vehicle class object and does not support any **map** or **options**. For compatibility with RVC book.

## Options

‘npoints’, N	Number of nodes in the tree (default 500)
‘time’, T	Interval over which to simulate dynamic model toward random point (default 0.5s)
‘range’, R	Specify rectangular bounds
	• 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)
‘goal’, P	Goal position (1 × 2) or pose (1 × 3) in workspace
‘speed’, S	Speed of vehicle [m/s] (default 1)
‘steermax’, S	Steering angle of vehicle in the range -S to +S [rad] (default 1.2)

## Notes

- Does not (yet) support obstacles, ie. **map** is ignored but must be given.
- ‘steermax’ selects the range of steering angles that the vehicle will be asked to track. If not given the steering angle range of the vehicle object will be used.
- There is no check that the steering range or speed is within the limits of the vehicle object.

## Reference

- Robotics, Vision & Control Peter Corke, Springer 2011. p102.

## See also

[Vehicle](#)

---

# RRT.char

## Convert to string

R.**char**() is a string representing the state of the **RRT** object in human-readable form.

---

# RRT.path

## Find a path between two points

x = R.**path**(start, goal) finds a **path** ( $N \times 3$ ) from state **start** ( $1 \times 3$ ) to the **goal** ( $1 \times 3$ ).

R.**path**(start, goal) as above but plots the **path** in 3D. 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](#)

---

# 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 ( $1 \times 3$ )
‘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.

---

## rt2tr

### Convert rotation and translation to homogeneous transform

`TR = rt2tr(R, t)` is a homogeneous transformation matrix ( $M \times M$ ) formed from an orthonormal rotation matrix `R` ( $N \times N$ ) and a translation vector `t` ( $N \times 1$ ) where  $M=N+1$ .

For a sequence `R` ( $N \times N \times K$ ) and `t` ( $N \times K$ ) results in a transform sequence ( $M \times M \times K$ ).

### Notes

- Works for `R` in SO(2) or SO(3)
  - If `R` is  $2 \times 2$  and `t` is  $2 \times 1$ , then `TR` is  $3 \times 3$
  - If `R` is  $3 \times 3$  and `t` is  $3 \times 1$ , then `TR` is  $4 \times 4$
- The validity of `R` is not checked

### See also

[t2r](#), [r2t](#), [tr2rt](#)

---

## rtbdemo

### Robot toolbox demonstrations

`rtbdemo` displays a menu of toolbox demonstration scripts that illustrate:

- homogeneous transformations
- trajectories

- forward kinematics
- inverse kinematics
- robot animation
- inverse dynamics
- forward dynamics

**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.
- 

# runscript

## Run an M-file in interactive fashion

**runscript(fname, options)** runs the M-file **fname** 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 <b>fname</b> in the folder P (default .)
‘dock’	Dock figures within GUI

## Notes

- If no file extension is given in **fname**, .m is assumed.
- 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 Central.

- If the file has a lot of boilerplate, you can skip over and not display it by giving the ‘begin’ option which searches 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 = RVC PATH` is the path of the top level folder for the installed RVC tools.

---

## se2

### Create planar translation and rotation transformation

`T = se2(x, y, theta)` is an SE(2) homogeneous transformation ( $3 \times 3$ ) representing translation `x` and `y`, and rotation `theta` in the plane.

`T = se2(xy)` as above where `xy=[x,y]` and rotation is zero

`T = se2(xy, theta)` as above where `xy=[x,y]`

`T = se2(xyt)` as above where `xyt=[x,y,theta]`

### See also

[transl2](#), [rot2](#), [ishomog2](#), [isrot2](#), [trplot2](#)

---

## se3

### Lift SE(2) transform to SE(3)

**t3 = se3(t2)** returns a homogeneous transform ( $4 \times 4$ ) that represents the same X,Y translation and Z rotation as does **t2** ( $3 \times 3$ ).

### See also

[se2](#), [transl](#), [rotx](#)

---

## Sensor

### Sensor superclass

A 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 Map object representing the landmarks around the robot

### Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

### See also

[rangebearing](#), [EKF](#), [Vehicle](#), [Map](#)

---

## Sensor.Sensor

### Sensor object constructor

`s = Sensor(vehicle, map, options)` is a sensor mounted on a vehicle described by the Vehicle class object `vehicle` and observing landmarks in a map described by the Map 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]

---

## 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
- 

## SerialLink

### Serial-link robot class

A concrete class that represents a serial-link arm-type robot. The mechanism is described using Denavit-Hartenberg parameters, one set per joint.

## Methods

<b>plot</b>	display graphical representation of robot
plot3d	display 3D graphical model of robot
teach	drive the graphical robot
getpos	get position of graphical robot
jtraj	a joint space trajectory
edit	display and edit kinematic and dynamic parameters
isspherical	test if robot has spherical wrist
islimit	test if robot at joint limit
isconfig	test robot joint configuration
fkine	forward kinematics
A	link transforms
trchain	forward kinematics as a chain of elementary transforms
ikine6s	inverse kinematics for 6-axis spherical wrist revolute robot
ikine	inverse kinematics using iterative numerical method
ikunc	inverse kinematics using optimisation
ikcon	inverse kinematics using optimisation with joint limits
ikine_sym	analytic inverse kinematics obtained symbolically
jacob0	Jacobian matrix in world frame
jacobn	Jacobian matrix in tool frame
jacob_dot	Jacobian derivative
maniplty	manipulability
vellipse	display velocity ellipsoid
fellipse	display force ellipsoid
qmincon	null space motion to centre joints between limits
accel	joint acceleration
coriolis	Coriolis joint force
dyn	show dynamic properties of links
friction	friction force
gravload	gravity joint force
inertia	joint inertia matrix
cinertia	Cartesian inertia matrix
nofriction	set friction parameters to zero
rne	inverse dynamics
fdyn	forward dynamics
payload	add a payload in end-effector frame
perturb	randomly perturb link dynamic parameters
gravjac	gravity load and Jacobian
paycap	payload capacity
pay	payload effect
sym	a symbolic version of the object
gencoords	symbolic generalized coordinates
genforces	symbolic generalized forces
issym	test if object is symbolic

## Properties (read/write)

links	vector of Link objects ( $1 \times N$ )
gravity	direction of gravity [gx gy gz]
base	pose of robot's base ( $4 \times 4$ homog xform)
tool	robot's tool transform, T6 to tool tip ( $4 \times 4$ homog xform)
qlim	joint limits, [qmin qmax] ( $N \times 2$ )
offset	kinematic joint coordinate offsets ( $N \times 1$ )
name	name of robot, used for graphical display
manuf	annotation, manufacturer's name
comment	annotation, general comment
plotopt	options for <b>plot()</b> method (cell array)
fast	use MEX version of RNE. Can only be set true if the mex file exists. Default is true.

## Properties (read only)

n	number of joints
config	joint configuration string, eg. 'RRRRRR'
mdh	kinematic convention boolean (0=DH, 1=MDH)
theta	kinematic: joint angles ( $1 \times N$ )
d	kinematic: link offsets ( $1 \times N$ )
a	kinematic: link lengths ( $1 \times N$ )
alpha	kinematic: link twists ( $1 \times N$ )

## Overloaded operators

R1\*R2 concatenate two SerialLink manipulators R1 and R2

## Note

- SerialLink is a reference object.
- SerialLink objects can be used in vectors and arrays

## Reference

- Robotics, Vision & Control, Chaps 7-9, P. Corke, Springer 2011.
- Robot, Modeling & Control, M.Spong, S. Hutchinson & M. Vidyasagar, Wiley 2006.

## See also

[Link](#), [DHFactor](#)

---

# SerialLink.SerialLink

## Create a SerialLink robot object

**R = SerialLink(links, options)** is a robot object defined by a vector of Link class objects which can be instances of Link, Revolute, Prismatic, RevoluteMDH or PrismaticMDH.

**R = SerialLink(options)** is a null robot object with no links.

**R = SerialLink([R1 R2 ...], options)** concatenate robots, the base of R2 is attached to the tip of R1. Can also be written R1\*R2 etc.

**R = SerialLink(R1, options)** is a deep copy of the robot object **R1**, with all the same properties.

**R = SerialLink(dh, options)** is a robot object with kinematics defined by the matrix **dh** which has one row per joint and each row is [theta d a alpha] and joints are assumed revolute. An optional fifth column sigma indicate revolute (sigma=0, default) or prismatic (sigma=1).

## Options

‘name’, NAME	set robot name property to NAME
‘comment’, COMMENT	set robot comment property to COMMENT
‘manufacturer’, MANUF	set robot manufacturer property to MANUF
‘base’, T	set base transformation matrix property to T
‘tool’, T	set tool transformation matrix property to T
‘gravity’, G	set gravity vector property to G
‘plotopt’, P	set default <b>options</b> for .plot() to P
‘plotopt3d’, P	set default <b>options</b> for .plot3d() to P
‘nofast’	don’t use RNE MEX file

## Examples

Create a 2-link robot

```
L(1) = Link([ 0      0    a1  pi/2], 'standard');
L(2) = Link([ 0      0    a2  0], 'standard');
twolink = SerialLink(L, 'name', 'two link');
```

Create a 2-link robot (most descriptive)

```
L(1) = Revolute('d', 0, 'a', a1, 'alpha', pi/2);
L(2) = Revolute('d', 0, 'a', a2, 'alpha', 0);
twolink = SerialLink(L, 'name', 'two link');
```

Create a 2-link robot (least descriptive)

```
twolink = SerialLink([0 0 a1 0; 0 0 a2 0], 'name', 'two link');
```

Robot objects can be concatenated in two ways

```
R = R1 * R2;
R = SerialLink([R1 R2]);
```

## Note

- SerialLink is a reference object, a subclass of Handle object.
- SerialLink objects can be used in vectors and arrays
- Link subclass elements passed in must be all standard, or all modified, **dh** parameters.
- When robots are concatenated (either syntax) the intermediate base and tool transforms are removed since general constant transforms cannot be represented in Denavit-Hartenberg notation.

## See also

[Link](#), [Revolute](#), [Prismatic](#), [RevoluteMDH](#), [PrismaticMDH](#), [SerialLink.plot](#)

---

# SerialLink.A

## Link transformation matrices

**s = R.A(J, qj)** is an SE(3) homogeneous transform ( $4 \times 4$ ) that transforms from link frame  $\{J-1\}$  to frame  $\{J\}$  which is a function of the  $J$ 'th joint variable **qj**.

**s = R.A(jlist, q)** as above but is a composition of link transform matrices given in the list JLIST, and the joint variables are taken from the corresponding elements of Q.

## Exmaples

For example, the link transform for joint 4 is

```
robot.A(4, q4)
```

The link transform for joints 3 through 6 is

```
robot.A([3 4 5 6], q)
```

where **q** is  $1 \times 6$  and the elements **q(3) .. q(6)** are used.

## Notes

- base and tool transforms are not applied.
-

# SerialLink.accel

## Manipulator forward dynamics

**qdd = R.accel(q, qd, torque)** is a vector ( $N \times 1$ ) of joint accelerations that result from applying the actuator force/torque to the manipulator robot R in state **q** and **qd**, and N is the number of robot joints.

If **q**, **qd**, **torque** are matrices ( $K \times N$ ) then **qdd** is a matrix ( $K \times N$ ) where each row is the acceleration corresponding to the equivalent rows of **q**, **qd**, **torque**.

**qdd = R.accel(x)** as above but **x=[q, qd, torque]** ( $1 \times 3N$ ).

## Note

- Useful for simulation of manipulator dynamics, in conjunction with a numerical integration function.
- Uses the method 1 of Walker and Orin to compute the forward dynamics.
- Featherstone's method is more efficient for robots with large numbers of joints.
- Joint friction is considered.

## References

- Efficient dynamic computer simulation of robotic mechanisms, M. W. Walker and D. E. Orin, ASME Journal of Dynamic Systems, Measurement and Control, vol. 104, no. 3, pp. 205-211, 1982.

## See also

[SerialLink.rne](#), [SerialLink](#), [ode45](#)

---

# SerialLink.animate

## Update a robot animation

**R.animate(q)** updates an existing animation for the robot R. This will have been created using **R.plot()**. Updates graphical instances of this robot in all figures.

## Notes

- Called by **plot()** and **plot3d()** to actually move the arm models.
- Used for Simulink robot animation.

**See also**

[SerialLink.plot](#)

---

## SerialLink.char

**Convert to string**

**s** = R.**char**() is a string representation of the robot's kinematic parameters, showing DH parameters, joint structure, comments, gravity vector, base and tool transform.

---

## SerialLink.cinertia

**Cartesian inertia matrix**

**m** = R.**cinertia**(**q**) is the  $N \times N$  Cartesian (operational space) inertia matrix which relates Cartesian force/torque to Cartesian acceleration at the joint configuration **q**, and  $N$  is the number of robot joints.

**See also**

[SerialLink.inertia](#), [SerialLink.rne](#)

---

## SerialLink.collisions

**Perform collision checking**

**C** = R.**collisions**(**q**, **model**) is true if the **SerialLink** object R at pose **q** ( $1 \times N$ ) intersects the solid model **model** which belongs to the class CollisionModel. The model comprises a number of geometric primitives and associate pose.

**C** = R.**collisions**(**q**, **model**, **dynmodel**, **tdyn**) as above but also checks dynamic collision model **dynmodel** whose elements are at pose **tdyn**. **tdyn** is an array of transformation matrices ( $4 \times 4 \times P$ ), where  $P = \text{length}(\text{dynmodel.primitives})$ . The  $P$ 'th plane of **tdyn** premultiplies the pose of the  $P$ 'th primitive of **dynmodel**.

**C** = R.**collisions**(**q**, **model**, **dynmodel**) as above but assumes **tdyn** is the robot's tool frame.

## Trajectory operation

If **q** is  $M \times N$  it is taken as a pose sequence and **C** is  $M \times 1$  and the collision value applies to the pose of the corresponding row of **q**. **tdyn** is  $4 \times 4 \times M \times P$ .

## Notes

- Requires the pHRIWARE package which defines CollisionModel class. Available from: <https://code.google.com/p/phriware/>.
- The robot is defined by a point cloud, given by its points property.
- The function does not currently check the base of the SerialLink object.
- If **model** is [] then no static objects are assumed.

## Author

Bryan Moutrie

## See also

[collisionmodel](#), [SerialLink](#)

---

# SerialLink.coriolis

## Coriolis matrix

**C = R.coriolis(q, qd)** is the Coriolis/centripetal matrix ( $N \times N$ ) for the robot in configuration **q** and velocity **qd**, where  $N$  is the number of joints. The product **C\*qd** is the vector of joint force/torque due to velocity coupling. The diagonal elements are due to centripetal effects and the off-diagonal elements are due to Coriolis effects. This matrix is also known as the velocity coupling matrix, since it describes the disturbance forces on any joint due to velocity of all other joints.

If **q** and **qd** are matrices ( $K \times N$ ), each row is interpreted as a joint state vector, and the result ( $N \times N \times K$ ) is a 3d-matrix where each plane corresponds to a row of **q** and **qd**.

**C = R.coriolis( qqd)** as above but the matrix **qqd** ( $1 \times 2N$ ) is **[q qd]**.

## Notes

- Joint viscous friction is also a joint force proportional to velocity but it is eliminated in the computation of this value.

- Computationally slow, involves  $N^2/2$  invocations of RNE.

**See also**

[SerialLink.rne](#)

---

## SerialLink.display

### Display parameters

R.**display**() displays the robot parameters in human-readable form.

**Notes**

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

**See also**

[SerialLink.char](#), [SerialLink.dyn](#)

---

## SerialLink.dyn

### Print inertial properties

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

R.**dyn(J)** as above but display parameters for joint **J** only.

**See also**

[Link.dyn](#)

---

## SerialLink.edit

### Edit kinematic and dynamic parameters of a seriallink manipulator

R.edit displays the kinematic parameters of the robot as an editable table in a new figure.

R.edit('dyn') as above but also displays the dynamic parameters.

#### Notes

- The ‘Save’ button copies the values from the table to the SerialLink manipulator object.
  - To exit the editor without updating the object just kill the figure window.
- 

## SerialLink.fdyn

### Integrate forward dynamics

[T,q,qd] = R.fdyn(T, torqfun) integrates the dynamics of the robot over the time interval 0 to T and returns vectors of time T, joint position q and joint velocity qd. The initial joint position and velocity are zero. The torque applied to the joints is computed by the user-supplied control function torqfun:

```
TAU = TORQFUN(T, Q, QD)
```

where q and qd are the manipulator joint coordinate and velocity state respectively, and T is the current time.

[ti,q,qd] = R.fdyn(T, torqfun, q0, qd0) as above but allows the initial joint position and velocity to be specified.

[T,q,qd] = R.fdyn(T1, torqfun, q0, qd0, ARG1, ARG2, ...) allows optional arguments to be passed through to the user-supplied control function:

```
TAU = TORQFUN(T, Q, QD, ARG1, ARG2, ...)
```

For example, if the robot was controlled by a PD controller we can define a function to compute the control

```
function tau = mytorqfun(t, q, qd, qstar, P, D)
tau = P*(qstar-q) + D*qd;
```

and then integrate the robot dynamics with the control

```
[t,q] = robot.fdyn(10, @mytorqfun, qstar, P, D);
```

**Note**

- This function performs poorly with non-linear joint friction, such as Coulomb friction. The `R.nofriction()` method can be used to set this friction to zero.
- If `torqfun` is not specified, or is given as 0 or [], then zero torque is applied to the manipulator joints.
- The builtin integration function `ode45()` is used.

**See also**

[SerialLink.accel](#), [SerialLink.nofriction](#), [SerialLink.rne](#), [ode45](#)

---

# SerialLink.fkine

## Forward kinematics

`T = R.fkine(q, options)` is the pose of the robot end-effector as an SE(3) homogeneous transformation ( $4 \times 4$ ) for the joint configuration `q` ( $1 \times N$ ).

If `q` is a matrix ( $K \times N$ ) the rows are interpreted as the generalized joint coordinates for a sequence of points along a trajectory. `q(i,j)` is the j'th joint parameter for the i'th trajectory point. In this case `T` is a 3d matrix ( $4 \times 4 \times K$ ) where the last subscript is the index along the path.

`[T,all] = R.fkine(q)` as above but `all` ( $4 \times 4 \times N$ ) is the pose of the link frames 1 to N, such that `all(:, :, k)` is the pose of link frame k.

## Options

‘deg’ Assume that revolute joint coordinates are in degrees not radians

**Note**

- The robot’s base or tool transform, if present, are incorporated into the result.
- Joint offsets, if defined, are added to `q` before the forward kinematics are computed.

**See also**

[SerialLink.ikine](#), [SerialLink.ikine6s](#)

---

## SerialLink.friction

### Friction force

**tau** = R.**friction**(**qd**) is the vector of joint **friction** forces/torques for the robot moving with joint velocities **qd**.

The **friction** model includes:

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

### See also

[Link.friction](#)

---

## SerialLink.gencoords

### Vector of symbolic generalized coordinates

**q** = R.**gencoords**() is a vector ( $1 \times N$ ) of symbols [q<sub>1</sub> q<sub>2</sub> ... q<sub>N</sub>].

[**q**,**qd**] = R.**gencoords**() as above but **qd** is a vector ( $1 \times N$ ) of symbols [qd<sub>1</sub> qd<sub>2</sub> ... qd<sub>N</sub>].

[**q**,**qd**,**qdd**] = R.**gencoords**() as above but **qdd** is a vector ( $1 \times N$ ) of symbols [qdd<sub>1</sub> qdd<sub>2</sub> ... qdd<sub>N</sub>].

---

## SerialLink.genforces

### Vector of symbolic generalized forces

**q** = R.**genforces**() is a vector ( $1 \times N$ ) of symbols [Q<sub>1</sub> Q<sub>2</sub> ... Q<sub>N</sub>].

---

## SerialLink.getpos

### Get joint coordinates from graphical display

**q** = R.**getpos**() returns the joint coordinates set by the last plot or teach operation on the graphical robot.

**See also**

[SerialLink.plot](#), [SerialLink.teach](#)

---

## SerialLink.gravjac

### Fast gravity load and Jacobian

**[tau,jac0] = R.gravjac(q)** is the generalised joint force/torques due to gravity ( $1 \times N$ ) and the manipulator Jacobian in the base frame ( $6 \times N$ ) for robot pose **q** ( $1 \times N$ ), where  $N$  is the number of robot joints.

**[tau,jac0] = R.gravjac(q,grav)** as above but gravity is given explicitly by **grav** ( $3 \times 1$ ).

### Trajectory operation

If **q** is  $M \times N$  where  $N$  is the number of robot joints then a trajectory is assumed where each row of **q** corresponds to a pose. **tau** ( $M \times N$ ) is the generalised joint torque, each row corresponding to an input pose, and **jac0** ( $6 \times N \times M$ ) where each plane is a Jacobian corresponding to an input pose.

### Notes

- The gravity vector is defined by the SerialLink property if not explicitly given.
- Does not use inverse dynamics function RNE.
- Faster than computing gravity and Jacobian separately.

### Author

Bryan Moutrie

**See also**

[SerialLink.pay](#), [SerialLink](#), [SerialLink.gravload](#), [SerialLink.jacob0](#)

---

## SerialLink.gravload

### Gravity load on joints

**taug = R.gravload(q)** is the joint gravity loading ( $1 \times N$ ) for the robot R in the joint configuration **q** ( $1 \times N$ ), where N is the number of robot joints. Gravitational acceleration is a property of the robot object.

If **q** is a matrix ( $M \times N$ ) each row is interpreted as a joint configuration vector, and the result is a matrix ( $M \times N$ ) each row being the corresponding joint torques.

**taug = R.gravload(q, grav)** as above but the gravitational acceleration vector **grav** is given explicitly.

### See also

[SerialLink.rne](#), [SerialLink.itorque](#), [SerialLink.coriolis](#)

---

## SerialLink.ikcon

### Numerical inverse kinematics with joint limits

**q = R.ikcon(T)** are the joint coordinates ( $1 \times N$ ) corresponding to the robot end-effector pose **T** ( $4 \times 4$ ) which is a homogenous transform.

**[q,err] = robot.ikcon(T)** as above but also returns **err** which is the scalar final value of the objective function.

**[q,err,exitflag] = robot.ikcon(T)** as above but also returns the status **exitflag** from fmincon.

**[q,err,exitflag] = robot.ikcon(T, q0)** as above but specify the initial joint coordinates **q0** used for the minimisation.

**[q,err,exitflag] = robot.ikcon(T, q0, options)** as above but specify the **options** for fmincon to use.

### Trajectory operation

In all cases if **T** is  $4 \times 4 \times M$  it is taken as a homogeneous transform sequence and **R.ikcon()** returns the joint coordinates corresponding to each of the transforms in the sequence. **q** is  $M \times N$  where N is the number of robot joints. The initial estimate of **q** for each time step is taken as the solution from the previous time step.

**err** and **exitflag** are also  $M \times 1$  and indicate the results of optimisation for the corresponding trajectory step.

## Notes

- Requires fmincon from the Optimization Toolbox.
- Joint limits are considered in this solution.
- Can be used for robots with arbitrary degrees of freedom.
- In the case of multiple feasible solutions, the solution returned depends on the initial choice of **q0**.
- Works by minimizing the error between the forward kinematics of the joint angle solution and the end-effector frame as an optimisation. The objective function (error) is described as:

```
sumsqr( (inv(T)*robot.fkine(q) - eye(4)) * omega )
```

Where omega is some gain matrix, currently not modifiable.

## Author

Bryan Moutrie

## See also

[SerialLink.ikunc](#), [fmincon](#), [SerialLink.ikine](#), [SerialLink.fkine](#)

---

# SerialLink.ikine

## Numerical inverse kinematics

**q = R.ikine(T)** are the joint coordinates ( $1 \times N$ ) corresponding to the robot end-effector pose **T** ( $4 \times 4$ ) which is a homogenous transform.

**q = R.ikine(T, q0, options)** specifies the initial estimate of the joint coordinates.

This method can be used for robots with 6 or more degrees of freedom.

## Underactuated robots

For the case where the manipulator has fewer than 6 DOF the solution space has more dimensions than can be spanned by the manipulator joint coordinates.

**q = R.ikine(T, q0, m, options)** similar to above but where **m** is a mask vector ( $1 \times 6$ ) which specifies the Cartesian DOF (in the wrist coordinate frame) that will be ignored in reaching a solution. The mask vector has six elements that correspond to translation in X, Y and Z, and rotation about X, Y and Z respectively. The value should be 0 (for ignore) or 1. The number of non-zero elements should equal the number of manipulator DOF.

For example when using a 3 DOF manipulator rotation orientation might be unimportant in which case  $\mathbf{m} = [1 \ 1 \ 1 \ 0 \ 0 \ 0]$ .

For robots with 4 or 5 DOF this method is very difficult to use since orientation is specified by  $\mathbf{T}$  in world coordinates and the achievable orientations are a function of the tool position.

## Trajectory operation

In all cases if  $\mathbf{T}$  is  $4 \times 4 \times \mathbf{m}$  it is taken as a homogeneous transform sequence and  $\text{R.ikine}()$  returns the joint coordinates corresponding to each of the transforms in the sequence.  $\mathbf{q}$  is  $\mathbf{m} \times N$  where  $N$  is the number of robot joints. The initial estimate of  $\mathbf{q}$  for each time step is taken as the solution from the previous time step.

## Options

‘pinv’	use pseudo-inverse instead of Jacobian transpose (default)
‘ilimit’, L	set the maximum iteration count (default 1000)
‘tol’, T	set the tolerance on error norm (default 1e-6)
‘alpha’, A	set step size gain (default 1)
‘varstep’	enable variable step size if pinv is false
‘verbose’	show number of iterations for each point
‘verbose=2’	show state at each iteration
‘plot’	plot iteration state versus time

## References

- Robotics, Vision & Control, Section 8.4, P. Corke, Springer 2011.

## Notes

- Solution is computed iteratively.
- Solution is sensitive to choice of initial gain. The variable step size logic (enabled by default) does its best to find a balance between speed of convergence and divergence.
- Some experimentation might be required to find the right values of tol, ilimit and alpha.
- The pinv option leads to much faster convergence (default)
- The tolerance is computed on the norm of the error between current and desired tool pose. This norm is computed from distances and angles without any kind of weighting.
- The inverse kinematic solution is generally not unique, and depends on the initial guess  $\mathbf{q0}$  (defaults to 0).

- The default value of  $\mathbf{q0}$  is zero which is a poor choice for most manipulators (eg. puma560, twolink) since it corresponds to a kinematic singularity.
- Such a solution is completely general, though much less efficient than specific inverse kinematic solutions derived symbolically, like ikine6s or ikine3.
- This approach allows a solution to be obtained at a singularity, but the joint angles within the null space are arbitrarily assigned.
- Joint offsets, if defined, are added to the inverse kinematics to generate  $\mathbf{q}$ .
- Joint limits are not considered in this solution.

### See also

[SerialLink.ikcon](#), [SerialLink.ikunc](#), [SerialLink.fkine](#), [SerialLink.ikine6s](#)

---

## SerialLink.ikine3

### Inverse kinematics for 3-axis robot with no wrist

$\mathbf{q} = \text{R.ikine3}(\mathbf{T})$  is the joint coordinates corresponding to the robot end-effector pose  $\mathbf{T}$  represented by the homogenous transform. This is a analytic solution for a 3-axis robot (such as the first three joints of a robot like the Puma 560).

$\mathbf{q} = \text{R.ikine3}(\mathbf{T}, \text{config})$  as above but specifies the configuration of the arm in the form of a string containing one or more of the configuration codes:

- ‘l’ arm to the left (default)
- ‘r’ arm to the right
- ‘u’ elbow up (default)
- ‘d’ elbow down

### Notes

- The same as IKINE6S without the wrist.
- The inverse kinematic solution is generally not unique, and depends on the configuration string.
- Joint offsets, if defined, are added to the inverse kinematics to generate  $\mathbf{q}$ .

### Reference

Inverse kinematics for a PUMA 560 based on the equations by Paul and Zhang From The International Journal of Robotics Research Vol. 5, No. 2, Summer 1986, p. 32-44

## Author

Robert Biro with Gary Von McMurray, GTRI/ATRP/IIMB, Georgia Institute of Technology 2/13/95

## See also

[SerialLink.FKINE](#), [SerialLink.IKINE](#)

---

# SerialLink.ikine6s

### Analytical inverse kinematics

$\mathbf{q} = R.\text{ikine6s}(T)$  is the joint coordinates ( $1 \times N$ ) corresponding to the robot end-effector pose  $T$  represented by an SE(3) homogenous transform ( $4 \times 4$ ). This is a analytic solution for a 6-axis robot with a spherical wrist (the most common form for industrial robot arms).

If  $T$  represents a trajectory ( $4 \times 4 \times M$ ) then the inverse kinematics is computed for all  $M$  poses resulting in  $\mathbf{q}$  ( $M \times N$ ) with each row representing the joint angles at the corresponding pose.

$\mathbf{q} = R.\text{IKINE6S}(T, \text{config})$  as above but specifies the configuration of the arm in the form of a string containing one or more of the configuration codes:

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

## Notes

- Treats a number of specific cases:
  - Robot with no shoulder offset
  - Robot with a shoulder offset (has lefty/righty configuration)
  - Robot with a shoulder offset and a prismatic third joint (like Stanford arm)
  - The Puma 560 arms with shoulder and elbow offsets (4 lengths parameters)
  - The Kuka KR5 with many offsets (7 length parameters)
- The inverse kinematic solution is generally not unique, and depends on the configuration string.
- Joint offsets, if defined, are added to the inverse kinematics to generate  $\mathbf{q}$ .

- Only applicable for standard Denavit-Hartenberg parameters

## Reference

- Inverse kinematics for a PUMA 560, Paul and Zhang, The International Journal of Robotics Research, Vol. 5, No. 2, Summer 1986, p. 32-44

## Author

- The Puma560 case: Robert Biro with Gary Von McMurray, GTRI/ATRP/IIMB, Georgia Institute of Technology, 2/13/95
- Kuka KR5 case: Gautam Sinha, Autobirdz Systems Pvt. Ltd., SIDBI Office, Indian Institute of Technology Kanpur, Kanpur, Uttar Pradesh.

## See also

[SerialLink.FKINE](#), [SerialLink.IKINE](#)

---

# SerialLink.ikine\_sym

## Symbolic inverse kinematics

**q = R.IKINE\_SYM(k, options)** is a cell array ( $C \times 1$ ) of inverse kinematic solutions of the **SerialLink** object ROBOT. The cells of **q** represent the different possible configurations. Each cell of **q** is a vector ( $N \times 1$ ), and element  $J$  is the symbolic expressions for the  $J$ 'th joint angle. The solution is in terms of the desired end-point pose of the robot which is represented by the symbolic matrix ( $3 \times 4$ ) with elements

```
nx ox ax tx
ny oy ay ty
nz oz az tz
```

where the first three columns specify orientation and the last column specifies translation.

**k <= N** can have only specific values:

- 2 solve for translation tx and ty
- 3 solve for translation tx, ty and tz
- 6 solve for translation and orientation

## Options

‘file’, F Write the solution to an m-file named F

## Example

```
mdl_planar2
sol = p2.ikine_sym(2);
length(sol)
ans =
2           % there are 2 solutions
s1 = sol{1}  % is one solution
q1 = s1(1);    % the expression for q1
q2 = s1(2);    % the expression for q2
```

## Notes

- Requires the Symbolic Toolbox for MATLAB.
  - This code is experimental and has a lot of diagnostic prints.
  - Based on the classical approach using Pieper's method.
- 

# SerialLink.ikinem

## Numerical inverse kinematics by minimization

**q = R.ikinem(T)** is the joint coordinates corresponding to the robot end-effector pose **T** which is a homogenous transform.

**q = R.ikinem(T, q0, options)** specifies the initial estimate of the joint coordinates.

In all cases if **T** is  $4 \times 4 \times M$  it is taken as a homogeneous transform sequence and **R.ikinem()** returns the joint coordinates corresponding to each of the transforms in the sequence. **q** is  $M \times N$  where  $N$  is the number of robot joints. The initial estimate of **q** for each time step is taken as the solution from the previous time step.

## Options

'pweight', P	weighting on position error norm compared to rotation error (default 1)
'stiffness', S	Stiffness used to impose a smoothness constraint on joint angles, useful when N is large (default 0)
'qlimits'	Enforce joint limits
'ilimit', L	Iteration limit (default 1000)
'nolm'	Disable Levenberg-Marquardt

## Notes

- PROTOTYPE CODE UNDER DEVELOPMENT, intended to do numerical inverse kinematics with joint limits

- The inverse kinematic solution is generally not unique, and depends on the initial guess **q0** (defaults to 0).
- The function to be minimized is highly nonlinear and the solution is often trapped in a local minimum, adjust **q0** if this happens.
- The default value of **q0** is zero which is a poor choice for most manipulators (eg. puma560, twolink) since it corresponds to a kinematic singularity.
- Such a solution is completely general, though much less efficient than specific inverse kinematic solutions derived symbolically, like **ikine6s** or **ikine3.%** - Uses Levenberg-Marquadt minimizer **LMFsolve** if it can be found, if ‘nolm’ is not given, and ‘qlimits’ false
- The error function to be minimized is computed on the norm of the error between current and desired tool pose. This norm is computed from distances and angles and ‘pweight’ can be used to scale the position error norm to be congruent with rotation error norm.
- This approach allows a solution to be obtained at a singularity, but the joint angles within the null space are arbitrarily assigned.
- Joint offsets, if defined, are added to the inverse kinematics to generate **q**.
- Joint limits become explicit constraints if ‘qlimits’ is set.

## See also

[fminsearch](#), [fmincon](#), [SerialLink.fkine](#), [SerialLink.ikine](#), [tr2angvec](#)

---

# SerialLink.ikunc

## Numerical inverse manipulator without joint limits

**q = R.ikunc(T)** are the joint coordinates ( $1 \times N$ ) corresponding to the robot end-effector pose **T** ( $4 \times 4$ ) which is a homogenous transform, and **N** is the number of robot joints.

**[q,err] = robot.ikunc(T)** as above but also returns **err** which is the scalar final value of the objective function.

**[q,err,exitflag] = robot.ikunc(T)** as above but also returns the status **exitflag** from fminunc.

**[q,err,exitflag] = robot.ikunc(T, q0)** as above but specify the initial joint coordinates **q0** used for the minimisation.

**[q,err,exitflag] = robot.ikunc(T, q0, options)** as above but specify the **options** for fminunc to use.

## Trajectory operation

In all cases if  $\mathbf{T}$  is  $4 \times 4 \times M$  it is taken as a homogeneous transform sequence and  $\mathbf{R}.\text{ikunc}()$  returns the joint coordinates corresponding to each of the transforms in the sequence.  $\mathbf{q}$  is  $M \times N$  where  $N$  is the number of robot joints. The initial estimate of  $\mathbf{q}$  for each time step is taken as the solution from the previous time step.

**err** and **exitflag** are also  $M \times 1$  and indicate the results of optimisation for the corresponding trajectory step.

## Notes

- Requires fminunc from the Optimization Toolbox.
- Joint limits are not considered in this solution.
- Can be used for robots with arbitrary degrees of freedom.
- In the case of multiple feasible solutions, the solution returned depends on the initial choice of  $\mathbf{q}_0$
- Works by minimizing the error between the forward kinematics of the joint angle solution and the end-effector frame as an optimisation. The objective function (error) is described as:

```
sumsqr( (inv(T)*robot.fkine(q) - eye(4)) * omega )
```

Where omega is some gain matrix, currently not modifiable.

## Author

Bryan Moutrie

## See also

[SerialLink.ikcon](#), [fmincon](#), [SerialLink.ikine](#), [SerialLink.fkine](#)

---

# SerialLink.inertia

## Manipulator inertia matrix

$\mathbf{i} = \mathbf{R}.\text{inertia}(\mathbf{q})$  is the symmetric joint **inertia** matrix ( $N \times N$ ) which relates joint torque to joint acceleration for the robot at joint configuration  $\mathbf{q}$ .

If  $\mathbf{q}$  is a matrix ( $K \times N$ ), each row is interpreted as a joint state vector, and the result is a 3d-matrix ( $N \times N \times K$ ) where each plane corresponds to the **inertia** for the corresponding row of  $\mathbf{q}$ .

## Notes

- The diagonal elements  $i(J,J)$  are the **inertia** seen by joint actuator J.
- The off-diagonal elements  $i(J,K)$  are coupling inertias that relate acceleration on joint J to force/torque on joint K.
- The diagonal terms include the motor **inertia** reflected through the gear ratio.

## See also

[SerialLink.RNE](#), [SerialLink.CINERTIA](#), [SerialLink.ITORQUE](#)

---

# SerialLink.isconfig

## Test for particular joint configuration

`R.isconfig(s)` is true if the robot has the joint configuration string given by the string `s`.

Example:

```
robot.isconfig('RRRRRR');
```

## See also

[SerialLink.config](#)

---

# SerialLink.islimit

## Joint limit test

`v = R.islimit(q)` is a vector of boolean values, one per joint, false (0) if `q(i)` is within the joint limits, else true (1).

## Notes

- Joint limits are purely advisory and are not used in any other function. Just seemed like a useful thing to include...

## See also

[Link.islimit](#)

---

## SerialLink.isspherical

### Test for spherical wrist

`R.isspherical()` is true if the robot has a spherical wrist, that is, the last 3 axes are revolute and their axes intersect at a point.

### See also

[SerialLink.ikine6s](#)

---

## SerialLink.issym

### Check if SerialLink object is a symbolic model

`res = R.issym()` is true if the **SerialLink** manipulator `R` has symbolic parameters

### Authors

Joern Malzahn, ([joern.malzahn@tu-dortmund.de](mailto:joern.malzahn@tu-dortmund.de))

---

## SerialLink.itorque

### Inertia torque

`taui = R.itorque(q, qdd)` is the inertia force/torque vector ( $1 \times N$ ) at the specified joint configuration `q` ( $1 \times N$ ) and acceleration `qdd` ( $1 \times N$ ), and  $N$  is the number of robot joints. `taui = INERTIA(q)*qdd`.

If `q` and `qdd` are matrices ( $K \times N$ ), each row is interpreted as a joint state vector, and the result is a matrix ( $K \times N$ ) where each row is the corresponding joint torques.

### Note

- If the robot model contains non-zero motor inertia then this will included in the result.

### See also

[SerialLink.inertia](#), [SerialLink.rne](#)

---

## SerialLink.jacob0

### Jacobian in world coordinates

**j0 = R.jacob0(q, options)** is the Jacobian matrix ( $6 \times N$ ) for the robot in pose **q** ( $1 \times N$ ), and  $N$  is the number of robot joints. The manipulator Jacobian matrix maps joint velocity to end-effector spatial velocity  $V = j0 * QD$  expressed in the world-coordinate frame.

### Options

'rpy'	Compute analytical Jacobian with rotation rate in terms of roll-pitch-yaw angles
'eul'	Compute analytical Jacobian with rotation rates in terms of Euler angles
'trans'	Return translational submatrix of Jacobian
'rot'	Return rotational submatrix of Jacobian

### Note

- The Jacobian is computed in the end-effector frame and transformed to the world frame.
- The default Jacobian returned is often referred to as the geometric Jacobian, as opposed to the analytical Jacobian.

### See also

[SerialLink.jacobn](#), [jsingu](#), [deltatr](#), [tr2delta](#), [jsingu](#)

---

## SerialLink.jacob\_dot

### Derivative of Jacobian

**jdq = R.jacob\_dot(q, qd)** is the product ( $6 \times 1$ ) of the derivative of the Jacobian (in the world frame) and the joint rates.

### Notes

- Useful for operational space control  $XDD = J(q)QDD + JDOT(q)qd$
- Written as per the reference and not very efficient.

## References

- Fundamentals of Robotics Mechanical Systems (2nd ed) J. Angeles, Springer 2003.

## See also

[SerialLink.jacob0](#), [diff2tr](#), [tr2diff](#)

---

# SerialLink.jacobi

## Jacobian in end-effector frame

**jn = R.jacobi(q, options)** is the Jacobian matrix ( $6 \times N$ ) for the robot in pose **q**, and  $N$  is the number of robot joints. The manipulator Jacobian matrix maps joint velocity to end-effector spatial velocity  $V = jn*QD$  in the end-effector frame.

## Options

- ‘trans’    Return translational submatrix of Jacobian  
‘rot’      Return rotational submatrix of Jacobian

## Notes

- This Jacobian is often referred to as the geometric Jacobian.

## References

- Differential Kinematic Control Equations for Simple Manipulators, Paul, Shimano, Mayer, IEEE SMC 11(6) 1981, pp. 456-460

## See also

[SerialLink.jacob0](#), [jsingu](#), [delta2tr](#), [tr2delta](#)

---

## SerialLink.jtraj

### Joint space trajectory

**q = R.jtraj(T1, t2, k, options)** is a joint space trajectory ( $k \times N$ ) where the joint coordinates reflect motion from end-effector pose **T1** to **t2** in **k** steps with default zero boundary conditions for velocity and acceleration. The trajectory **q** has one row per time step, and one column per joint, where **N** is the number of robot joints.

### Options

‘ikine’, **F** A handle to an inverse kinematic method, for example **F = @p560.ikunc**. Default is **ikine6s()** for a 6-axis spherical wrist, else **ikine()**.

Additional options are passed as trailing arguments to the inverse kinematic function.

### See also

[jtraj](#), [SerialLink.ikine](#), [SerialLink.ikine6s](#)

---

## SerialLink.maniply

### Manipulability measure

**m = R.maniply(q, options)** is the manipulability index (scalar) for the robot at the joint configuration **q** ( $1 \times N$ ) where **N** is the number of robot joints. It indicates dexterity, that is, how isotropic the robot’s motion is with respect to the 6 degrees of Cartesian motion. The measure is high when the manipulator is capable of equal motion in all directions and low when the manipulator is close to a singularity.

If **q** is a matrix ( $m \times N$ ) then **m** ( $m \times 1$ ) is a vector of manipulability indices for each joint configuration specified by a row of **q**.

**[m,ci] = R.maniply(q, options)** as above, but for the case of the Asada measure returns the Cartesian inertia matrix **ci**.

Two measures can be computed:

- Yoshikawa’s manipulability measure is based on the shape of the velocity ellipsoid and depends only on kinematic parameters.
- Asada’s manipulability measure is based on the shape of the acceleration ellipsoid which in turn is a function of the Cartesian inertia matrix and the dynamic parameters. The scalar measure computed here is the ratio of the smallest/largest ellipsoid axis. Ideally the ellipsoid would be spherical, giving a ratio of 1, but in practice will be less than 1.

## Options

'T'	manipulability for translational motion only (default)
'R'	manipulability for rotational motion only
'all'	manipulability for all motions
'dof', D	D is a vector ( $1 \times 6$ ) with non-zero elements if the corresponding DOF is to be included for manipulability
'yoshikawa'	use Yoshikawa algorithm (default)
'asada'	use Asada algorithm

## Notes

- The ‘all’ option includes rotational and translational dexterity, but this involves adding different units. It can be more useful to look at the translational and rotational manipulability separately.
- Examples in the RVC book can be replicated by using the ‘all’ option

## References

- Analysis and control of robot manipulators with redundancy, T. Yoshikawa, Robotics Research: The First International Symposium (m. Brady and R. Paul, eds.), pp. 735-747, The MIT press, 1984.
- A geometrical representation of manipulator dynamics and its application to arm design, H. Asada, Journal of Dynamic Systems, Measurement, and Control, vol. 105, p. 131, 1983.

## See also

[SerialLink.inertia](#), [SerialLink.jacob0](#)

---

# SerialLink.mtimes

## Concatenate robots

$R = R1 * R2$  is a robot object that is equivalent to mechanically attaching robot R2 to the end of robot R1.

## Notes

- If R1 has a tool transform or R2 has a base transform these are discarded since DH convention does not allow for arbitrary intermediate transformations.

## SerialLink.nofriction

### Remove friction

**rnf = R.nofriction()** is a robot object with the same parameters as R but with non-linear (Coulomb) friction coefficients set to zero.

**rnf = R.nofriction('all')** as above but viscous and Coulomb friction coefficients set to zero.

**rnf = R.nofriction('viscous')** as above but viscous friction coefficients are set to zero.

### Notes

- Non-linear (Coulomb) friction can cause numerical problems when integrating the equations of motion (R.fdyn).
- The resulting robot object has its name string prefixed with ‘NF’.

### See also

[SerialLink.fdyn](#), [Link.nofriction](#)

---

## SerialLink.pay

### Joint forces due to payload

**tau = R.PAY(w, J)** returns the generalised joint force/torques due to a payload wrench **w** ( $1 \times 6$ ) and where the manipulator Jacobian is **J** ( $6 \times N$ ), and  $N$  is the number of robot joints.

**tau = R.PAY(q, w, f)** as above but the Jacobian is calculated at pose **q** ( $1 \times N$ ) in the frame given by **f** which is ‘0’ for world frame, ‘n’ for end-effector frame.

Uses the formula **tau = J'w**, where **w** is a wrench vector applied at the end effector, **w** = [Fx Fy Fz Mx My Mz].

### Trajectory operation

In the case **q** is  $M \times N$  or **J** is  $6 \times N \times M$  then **tau** is  $M \times N$  where each row is the generalised force/torque at the pose given by corresponding row of **q**.

## Notes

- Wrench vector and Jacobian must be from the same reference frame.
- Tool transforms are taken into consideration when  $\mathbf{f} = \text{'n'}$ .
- Must have a constant wrench - no trajectory support for this yet.

## Author

Bryan Moutrie

## See also

[SerialLink.paycap](#), [SerialLink.jacob0](#), [SerialLink.jacobn](#)

---

# SerialLink.paycap

## Static payload capacity of a robot

**[wmax,J] = R.`paycap`(q, w, f, tlim)** returns the maximum permissible payload wrench **wmax** ( $1 \times 6$ ) applied at the end-effector, and the index of the joint **J** which hits its force/torque limit at that wrench. **q** ( $1 \times N$ ) is the manipulator pose, **w** the payload wrench ( $1 \times 6$ ), **f** the wrench reference frame (either '0' or 'n') and **tlim** ( $2 \times N$ ) is a matrix of joint forces/torques (first row is maximum, second row minimum).

## Trajectory operation

In the case **q** is  $M \times N$  then **wmax** is  $M \times 6$  and **J** is  $M \times 1$  where the rows are the results at the pose given by corresponding row of **q**.

## Notes

- Wrench vector and Jacobian must be from the same reference frame
- Tool transforms are taken into consideration for  $\mathbf{f} = \text{'n'}$ .

## Author

Bryan Moutrie

**See also**

[SerialLink.pay](#), [SerialLink.gravjac](#), [SerialLink.gravload](#)

---

## SerialLink.payload

### Add payload mass

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

### Notes

- An added **payload** will affect the inertia, Coriolis and gravity terms.

**See also**

[SerialLink.rne](#), [SerialLink.gravload](#)

---

## SerialLink.perturb

### Perturb robot parameters

**rp** = R.**perturb**(p) is a new robot object in which the dynamic parameters (link mass and inertia) have been perturbed. The perturbation is multiplicative so that values are multiplied by random numbers in the interval (1-p) to (1+p). The name string of the perturbed robot is prefixed by ‘p/’.

Useful for investigating the robustness of various model-based control schemes. For example to vary parameters in the range +/- 10 percent is:

---

```
r2 = p560.perturb(0.1);
```

---

## SerialLink.plot

### Graphical display and animation

R.**plot**(q, options) displays a graphical animation of a robot based on the kinematic model. A stick figure polyline joins the origins of the link coordinate frames. 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 zmn zmx]
‘floorlevel’, L	Z-coordinate of floor (default -1)
‘delay’, D	Delay between frames for animation (s)
‘fps’, fps	Number of frames per second for display, inverse of ‘delay’ option
‘[no]loop’	Loop over the trajectory forever
‘[no]raise’	Autoraise the figure
‘movie’, M	Save frames as files in the folder M
‘trail’, L	Draw a line recording the tip path, with line style L
‘scale’, S	Annotation scale factor
‘zoom’, Z	Reduce size of auto-computed workspace by Z, makes robot look bigger
‘ortho’	Orthographic view
‘perspective’	Perspective view (default)
‘view’, V	Specify view V=’x’, ‘y’, ‘top’ or [az el] for side elevations, plan view, or general view by azimuth and elevation angle.
‘[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])
‘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 is taken.

## 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.

## Multiple views of the same robot

If one or more plots of this robot already exist then these will all be moved according to the argument **q**. All robots in all windows with the same name will be moved.

Create a robot in figure 1

```
figure(1)
p560.plot(qz);
```

Create a robot in figure 2

```
figure(2)
p560.plot(qz);
```

Now move both robots

```
p560.plot(qn)
```

## Multiple robots in the same figure

Multiple robots can be displayed in the same **plot**, by using “hold on” before calls to **robot.plot()**.

Create a robot in figure 1

```
figure(1)
p560.plot(qz);
```

Make a clone of the robot named bob

```
bob = SerialLink(p560, 'name', 'bob');
```

Draw bob in this figure

```
hold on  
bob.plot(qn)
```

To animate both robots so they move together:

```
qtg = jtraj(qr, qz, 100);  
for q=qtg'  
  
p560.plot(q');  
bob.plot(q');  
  
end
```

## Making an animation movie

- The ‘movie’ **options** saves frames as files NNNN.png into the specified folder
- The specified folder will be created
- To convert frames to a movie use a command like:

```
ffmpeg -r 10 -i %04d.png out.avi
```

## Notes

- The **options** are processed when the figure is first drawn, to make different **options** come into effect it is necessary to clear the figure.
- The link segments do not necessarily represent the links of the robot, they are a pipe network that joins the origins of successive link coordinate frames.
- Delay between frames can be eliminated by setting option ‘delay’, 0 or ‘fps’, Inf.
- By default a quite detailed **plot** is generated, but turning off labels, axes, shadows etc. will speed things up.
- Each graphical robot object is tagged by the robot’s name and has UserData that holds graphical handles and the handle of the robot object.
- The graphical state holds the last joint configuration
- 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

[SerialLink.plot3d](#), [plotbotopt](#), [SerialLink.animate](#), [SerialLink.teach](#), [SerialLink.fkine](#)

---

# SerialLink.plot3d

## Graphical display and animation of solid model robot

**R.plot3d(q, options)** displays and animates a solid model of the robot. 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

‘color’, C	A cell array of color names, one per link. These are mapped to RGB using colorname(). If not given, colors come from the axis ColorOrder property.
‘alpha’, A	Set alpha for all links, 0 is transparent, 1 is opaque (default 1)
‘path’, P	Overide path to folder containing STL model files
‘workspace’, W	Size of robot 3D workspace, W = [xmn xmx ymn ymx zmn zmx]
‘floorlevel’, L	Z-coordinate of floor (default -1)
‘delay’, D	Delay between frames for animation (s)
‘fps’, fps	Number of frames per second for display, inverse of ‘delay’ option
‘[no]loop’	Loop over the trajectory forever
‘[no]raise’	Autoraise the figure
‘movie’, M	Save frames as files in the folder M
‘scale’, S	Annotation scale factor
‘ortho’	Orthographic view (default)
‘perspective’	Perspective view
‘view’, V	Specify view V=’x’, ‘y’, ‘top’ or [az el] for side elevations, plan view, or general view by azimuth and elevation angle.
‘[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]jaxes’	Enable display of joint axes (default true)
‘[no]joints’	Enable display of joints
‘[no]base’	Enable display of base shape

### Notes

- Solid models of the robot links are required as STL ascii format files, with extensions .stl
- Suitable STL files can be found in the package ARTE: A ROBOTICS TOOLBOX FOR EDUCATION by Arturo Gil, <https://arvc.umh.es/arte>

- The root of the solid models is an installation of ARTE with an empty file called arte.m at the top level
- Each STL model is called ‘linkN’.stl where N is the link number 0 to N
- The specific folder to use comes from the SerialLink.model3d property
- The path of the folder containing the STL files can be specified using the ‘path’ option
- The height of the floor is set in decreasing priority order by:
  - ‘workspace’ option, the fifth element of the passed vector
  - ‘floorlevel’ option
  - the lowest z-coordinate in the link1.stl object

## Authors

- Peter Corke, based on existing code for plot()
- Bryan Moutrie, demo code on the Google Group for connecting ARTE and RTB
- Don Riley, function rndread() extracted from cad2matdemo (MATLAB File Exchange)

## See also

[SerialLink.plot](#), [plotbotopt3d](#), [SerialLink.animate](#), [SerialLink.teach](#), [SerialLink.fkine](#)

---

# SerialLink.qmincon

## Use redundancy to avoid joint limits

**qs = R.qmincon(q)** exploits null space motion and returns a set of joint angles **qs** ( $1 \times N$ ) that result in the same end-effector pose but are away from the joint coordinate limits. N is the number of robot joints.

**[q,err] = R.qmincon(q)** as above but also returns **err** which is the scalar final value of the objective function.

**[q,err,exitflag] = R.qmincon(q)** as above but also returns the status **exitflag** from fmincon.

## Trajectory operation

In all cases if **q** is  $M \times N$  it is taken as a pose sequence and **R.qmincon()** returns the adjusted joint coordinates ( $M \times N$ ) corresponding to each of the poses in the sequence.

**err** and **exitflag** are also  $M \times 1$  and indicate the results of optimisation for the corresponding trajectory step.

## Notes

- Requires fmincon from the Optimization Toolbox.
- Robot must be redundant.

## Author

Bryan Moutrie

## See also

[SerialLink.ikcon](#), [SerialLink.ikunc](#), [SerialLink.jacob0](#)

---

# SerialLink.rne

## Inverse dynamics

**tau = R.rne(q, qd, qdd)** is the joint torque required for the robot R to achieve the specified joint position **q** ( $1 \times N$ ), velocity **qd** ( $1 \times N$ ) and acceleration **qdd** ( $1 \times N$ ), where N is the number of robot joints.

**tau = R.rne(q, qd, qdd, grav)** as above but overriding the gravitational acceleration vector ( $3 \times 1$ ) in the robot object R.

**tau = R.rne(q, qd, qdd, grav, fext)** as above but specifying a wrench acting on the end of the manipulator which is a 6-vector [Fx Fy Fz Mx My Mz].

**tau = R.rne(x)** as above where **x=[q,qd,qdd]** ( $1 \times 3N$ ).

**tau = R.rne(x, grav)** as above but overriding the gravitational acceleration vector in the robot object R.

**tau = R.rne(x, grav, fext)** as above but specifying a wrench acting on the end of the manipulator which is a 6-vector [Fx Fy Fz Mx My Mz].

**[tau,wbase] = R.rne(x, grav, fext)** as above but the extra output is the wrench on the base.

## Trajectory operation

If **q, qd** and **qdd** ( $M \times N$ ), or **x** ( $M \times 3N$ ) are matrices with M rows representing a trajectory then **tau** ( $M \times N$ ) is a matrix with rows corresponding to each trajectory step.

## MEX file operation

This algorithm is relatively slow, and a MEX file can provide better performance. The MEX file is executed if:

- the robot is not symbolic, and
- the SerialLink property fast is true, and
- the MEX file frne.mexXXX exists in the subfolder rvctools/robot/mex.

## Notes

- The robot base transform is ignored.
- Currently the MEX-file version does not compute **wbase**.
- The torque computed contains a contribution due to armature inertia and joint friction.
- See the README file in the mex folder for details on how to configure MEX-file operation.
- The M-file is a wrapper which calls either RNE\_DH or RNE\_MDH depending on the kinematic conventions used by the robot object, or the MEX file.

## See also

[SerialLink.accel](#), [SerialLink.gravload](#), [SerialLink.inertia](#)

---

---

# SerialLink.teach

## Graphical teach pendant

R.**teach(q, options)** allows the user to “drive” a graphical robot by means of a graphical slider panel. If no graphical robot exists one is created in a new window. Otherwise all current instances of the graphical robot are driven. The robots are set to the initial joint angles **q**.

R.**teach(options)** as above but with options and the initial joint angles are taken from the pose of an existing graphical robot, or if that doesn’t exist then zero.

## 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)
'callback', CB	Set a callback function, called with robot object and joint angle vector: CB(R, q)

## Example

To display the velocity ellipsoid for a Puma 560

```
p560.teach('callback', @(r,q) r.vellipse(q));
```

## GUI

- The specified callback function is invoked every time the joint configuration changes. the joint coordinate vector.
- The Quit (red X) button destroys the **teach** window.

## Notes

- If the robot is displayed in several windows, only one has the **teach** panel added.
- 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

[SerialLink.plot](#), [SerialLink.getpos](#)

---

# SerialLink.trchain

## Convert to elementary transform sequence

**s = R.TRCHAIN(options)** is a sequence of elementary transforms that describe the kinematics of the serial link robot arm. 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 a joint variable, or a constant angle or length dimension.

For example:

```
>> mdl_puma560
>> p560.trchain
ans =
Rz (q1) Rx (90) Rz (q2) Tx (0.431800) Rz (q3) Tz (0.150050) Tx (0.020300) Rx (-90)
Rz (q4) Tz (0.431800) Rx (90) Rz (q5) Rx (-90) Rz (q6)
```

## Options

- ‘[no]deg’ Express angles in degrees rather than radians (default deg)
- ‘sym’ Replace length parameters by symbolic values L1, L2 etc.

## See also

[trchain](#), [trotx](#), [trotv](#), [trotz](#), [transl](#)

---

---

# 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 number is 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](#)

---

## skew

### Create skew-symmetric matrix

`s = skew(v)` is a **skew**-symmetric matrix formed from `v` ( $3 \times 1$ ).

$$\begin{vmatrix} 0 & -vz & vy \\ vz & 0 & -vx \\ -vy & vx & 0 \end{vmatrix}$$

### See also

[vex](#)

---

## startup\_rtb

### Initialize MATLAB paths for Robotics Toolbox

Adds demos, examples to the MATLAB path, and adds also to Java class path.

---

## 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](#)

---

# 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  $4 \times 4$ , then **R** is  $3 \times 3$ .
- If **T** is  $3 \times 3$ , then **R** is  $2 \times 2$ .

## Notes

- For a homogeneous transform sequence returns a rotation matrix sequence
- 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).

The software pattern is:

```

function(a, b, c, varargin)
opt.foo = false;
opt.bar = true;
opt.blah = [];
opt.choose = {'this', 'that', 'other'};
opt.select = {'#no', '#yes'};
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 := (the second element)

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.

Note:

- 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.
- To match an option that starts with a digit, prefix it with ‘d\_’, so the field ‘d\_3d’ matches the option ‘3d’.
- **opt** can be an object, rather than a structure, in which case the passed options are assigned to properties.

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	sets opt := S, if S.foo=4, and opt.foo is present, then opt.foo is set to 4.

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.

**[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**.

---

## 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** ( $m \times 1$ ).

**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 specifies the trajectory in terms of the length of the time vector **T** ( $m \times 1$ ).

Reference:

Robotics, Vision & Control Chap 3 Springer 2011

### See also

[lspb](#), [jtraj](#)

---

## tr2angvec

### Convert rotation matrix to angle-vector form

**[theta,v] = tr2angvec(R, options)** is rotation expressed in terms of an angle **theta** ( $1 \times 1$ ) about the axis **v** ( $1 \times 3$ ) equivalent to the orthonormal rotation matrix **R** ( $3 \times 3$ ).

**[theta,v] = tr2angvec(T, options)** as above but uses the rotational part of the homogeneous transform **T** ( $4 \times 4$ ).

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

## Notes

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

## See also

[angvec2r](#), [angvec2tr](#)

---

# tr2delta

## Convert homogeneous transform to differential motion

**d = tr2delta(T0, T1)** is the differential motion ( $6 \times 1$ ) corresponding to infinitessimal motion from pose **T0** to **T1** which are homogeneous transformations ( $4 \times 4$ ). **d**=(dx, dy, dz, dRx, dRy, dRz) and is an approximation to the average spatial velocity multiplied by time.

**d = tr2delta(T)** is the differential motion corresponding to the infinitessimal relative pose **T** expressed as a homogeneous transformation.

## Notes

- **d** is only an approximation to the motion **T**, and assumes that **T0≈T1** or **T≈eye(4,4)**.

## See also

[delta2tr](#), [skew](#)

---

# tr2eul

## Convert homogeneous transform to Euler angles

**eul = tr2eul(T, options)** are the ZYZ Euler angles ( $1 \times 3$ ) corresponding to the rotational part of a homogeneous transform **T** ( $4 \times 4$ ). 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** ( $3 \times 3$ ).

If  $\mathbf{R}$  ( $3 \times 3 \times K$ ) or  $\mathbf{T}$  ( $4 \times 4 \times K$ ) 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  $\text{THETA}=0$  in which case  $\text{PHI}$  is arbitrarily set to zero and  $\text{PSI}$  is the sum ( $\text{PHI}+\text{PSI}$ ).

## See also

[eul2tr](#), [tr2rpy](#)

---

# tr2jac

## Jacobian for differential motion

$\mathbf{J} = \text{tr2jac}(\mathbf{T})$  is a Jacobian matrix ( $6 \times 6$ ) that maps spatial velocity or differential motion from the world frame to the frame represented by the homogeneous transform  $\mathbf{T}$  ( $4 \times 4$ ).

## See also

[wtrans](#), [tr2delta](#), [delta2tr](#)

---

# tr2rpy

## Convert a homogeneous transform to roll-pitch-yaw angles

$\mathbf{rpy} = \text{tr2rpy}(\mathbf{T}, \text{options})$  are the roll-pitch-yaw angles ( $1 \times 3$ ) corresponding to the rotation part of a homogeneous transform  $\mathbf{T}$ . The 3 angles  $\mathbf{rpy}=[\mathbf{R}, \mathbf{P}, \mathbf{Y}]$  correspond to sequential rotations about the X, Y and Z axes respectively.

**rpy = tr2rpy(R, options)** as above but the input is an orthonormal rotation matrix **R** ( $3 \times 3$ ).

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)
- ‘zyx’ Return solution for sequential rotations about Z, Y, X axes (Paul book)

## 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**).
- Note that textbooks (Paul, Spong) use the rotation order ZYX.

## 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  $4 \times 4$ , then **R** is  $3 \times 3$  and **T** is  $3 \times 1$ .
- If **TR** is  $3 \times 3$ , then **R** is  $2 \times 2$  and **T** is  $2 \times 1$ .

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 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:

- homogeneous transformation matrices ( $4 \times 4$ )
- orthonormal rotation matrices ( $3 \times 3$ )
- Quaternion

**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

- 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]
‘movie’, M	Save frames as files in the folder M

Additional options are passed through to TRPLOT.

### Notes

- Uses the Animate helper class to record the frames.
- Poses X1 and X2 must both be of the same type
- The ‘movie’ option saves frames as files NNNN.png.
- To convert frames to a movie use a command like:

```
ffmpeg -r 10 -i %04d.png out.avi
```

**See also**

[trplot](#), [Animate](#)

---

## transl

**Create or unpack an SE3 translational transform**

**Create a translational transformation matrix**

**T = transl(x, y, z)** is an SE(3) homogeneous transform ( $4 \times 4$ ) representing a pure translation of **x**, **y** and **z**.

**T = transl(p)** is an SE(3) homogeneous transform ( $4 \times 4$ ) representing a translation of **p** = [x,y,z]. If **p** ( $M \times 3$ ) it 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**.

**Unpack the translational part of a transformation matrix**

**p = transl(T)** is the translational part of a homogeneous transform **T** as a 3-element column vector. If **T** ( $4 \times 4 \times M$ ) is a homogeneous transform sequence 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**

[ctraj](#)

---

## transl2

**Create or unpack an SE2 translational transform**

**Create a translational transformation matrix**

$T = \text{transl2}(x, y)$  is an SE2 homogeneous transform ( $3 \times 3$ ) representing a pure translation.

$T = \text{transl2}(p)$  is a homogeneous transform representing a translation or point  $p=[x,y]$ . If  $p$  ( $M \times 2$ ) it 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  $p$ .

**Unpack the translational part of a transformation matrix**

$p = \text{transl2}(T)$  is the translational part of a homogeneous transform as a 2-element column vector. If  $T$  ( $3 \times 3 \times M$ ) is a homogeneous transform sequence 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

[transl](#)

---

## trchain

**Chain 3D transforms from string**

$T = \text{trchain}(s, q)$  is a homogeneous transform ( $4 \times 4$ ) 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  $T_x$ ,  $T_y$ ,  $T_z$ ,  $R_x$ ,  $R_y$ , or  $R_z$ .  $ARG$  is the name of a variable in MATLAB workspace or  $qJ$  where  $J$  is an integer in the range 1 to  $N$  that selects the variable from the  $J$ th column of the vector  $q$  ( $1 \times N$ ).

For example:

```
trchain('Rx(q1)Tx(a1)Ry(q2)Ty(a3)Rz(q3)', [1 2 3])
```

is equivalent to computing:

```
trotx(1) * transl(a1,0,0) * troty(2) * transl(0,a3,0) * trotz(3)
```

## Notes

- The string can contain spaces between elements or on either side of ARG.
- Works for symbolic variables in the workspace and/or passed in via the vector **q**.
- For symbolic operations that involve use of the value pi, make sure you define it first in the workspace: `pi = sym('pi');`

## See also

[trchain2](#), [trotx](#), [trotz](#), [transl](#), [SerialLink.trchain](#), etc

---

# trchain2

## Chain 2D transforms from string

**T = trchain2(s, q)** is a homogeneous transform ( $3 \times 3$ ) 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  $X(\text{ARG})$  where  $X$  is one of  $\text{Tx}$ ,  $\text{Ty}$  or  $\text{R}$ .  $\text{ARG}$  is the name of a variable in MATLAB workspace or  $qJ$  where  $J$  is an integer in the range 1 to  $N$  that selects the variable from the  $J$ th column of the vector **q** ( $1 \times N$ ).

For example:

```
trchain('R(q1)Tx(a1)R(q2)Ty(a3)R(q3)', [1 2 3])
```

is equivalent to computing:

```
trot2(1) * transl2(a1,0) * trot2(2) * transl2(0,a3) * trot2(3)
```

## Notes

- The string can contain spaces between elements or on either side of ARG.
- Works for symbolic variables in the workspace and/or passed in via the vector **q**.
- For symbolic operations that involve use of the value pi, make sure you define it first in the workspace: `pi = sym('pi');`

**See also**

[trchain](#), [trot2](#), [transl2](#)

---

## trinterp

### Interpolate homogeneous transformations

**T = trinterp(T0, T1, s)** is a homogeneous transform ( $4 \times 4$ ) interpolated between **T0** when **s=0** and **T1** when **s=1**. **T0** and **T1** are both homogeneous transforms ( $4 \times 4$ ). Rotation is interpolated using quaternion spherical linear interpolation (slerp). 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**.

**T = trinterp(T1, s)** as above but interpolated between the identity matrix when **s=0** to **T1** when **s=1**.

**See also**

[ctraj](#), [quaternion](#)

---

---

## trnorm

### Normalize a rotation matrix

**rn = trnorm(R)** is guaranteed to be a proper orthogonal matrix rotation matrix ( $3 \times 3$ ) which is “close” to the non-orthogonal matrix **R** ( $3 \times 3$ ). 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**.

**tn = trnorm(T)** as above but the rotational submatrix of the homogeneous transformation **T** ( $4 \times 4$ ) is normalised while the translational part is passed unchanged.

If **R** ( $3 \times 3 \times K$ ) or **T** ( $4 \times 4 \times K$ ) represent a sequence then **rn** and **tn** have the same dimension and normalisation is performed on each plane.

## Notes

- Only the direction of A (the z-axis) is unchanged.
- Used to prevent finite word length arithmetic causing transforms to become ‘un-normalized’.

## See also

[oa2tr](#)

---

# trot2

## SE2 rotation matrix

**T = trot2(theta)** is a homogeneous transformation ( $3 \times 3$ ) 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](#), [trotx](#), [troty](#), [trotz](#)

---

# trotx

## Rotation about X axis

**T = trotx(theta)** is a homogeneous transformation ( $4 \times 4$ ) 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](#), [trony](#), [trotz](#), [trot2](#)

---

# trony

## Rotation about Y axis

**T** = **trony(theta)** is a homogeneous transformation ( $4 \times 4$ ) representing a rotation of **theta** radians about the y-axis.

**T** = **trony(theta, ‘deg’)** as above but **theta** is in degrees.

## Notes

- Translational component is zero.

## See also

[rotz](#), [trotx](#), [trony](#), [trot2](#)

---

# trotz

## Rotation about Z axis

**T** = **trotz(theta)** is a homogeneous transformation ( $4 \times 4$ ) 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](#)

---

## trplot

### Draw a coordinate frame

**trplot(T, options)** draws a 3D coordinate frame represented by the homogeneous transform  $\mathbf{T}$  ( $4 \times 4$ ).

$\mathbf{H} = \text{trplot}(\mathbf{T}, \text{options})$  as above but returns a handle.

**trplot(H, T)** moves the coordinate frame described by the handle  $\mathbf{H}$  to the pose  $\mathbf{T}$  ( $4 \times 4$ ).

**trplot(R, options)** as above but the coordinate frame is rotated about the origin according to the orthonormal rotation matrix  $\mathbf{R}$  ( $3 \times 3$ ).

$\mathbf{H} = \text{trplot}(\mathbf{R}, \text{options})$  as above but returns a handle.

**trplot(H, R)** moves the coordinate frame described by the handle  $\mathbf{H}$  to the orientation  $\mathbf{R}$ .

### 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 $\mathbf{A}=[\text{xmin } \text{xmax } \text{ymin } \text{ymax } \text{zmin } \text{zmax}]$
‘frame’, F	The coordinate frame is named {F} and the subscript on the axis labels is F.
‘text_opts’, opt	A cell array of MATLAB text properties
‘handle’, H	Draw in the MATLAB axes specified by the axis handle H
‘view’, V	Set plot view parameters V=[az el] angles, or ‘auto’ for view toward origin of coordinate frame
‘length’, s	Length of the coordinate frame arms (default 1)
‘arrow’	Use arrows rather than line segments for the axes
‘width’, w	Width of arrow tips (default 1)
‘thick’, t	Thickness of lines (default 0.5)
‘3d’	Plot in 3D using anaglyph graphics
‘anaglyph’, A	Specify anaglyph colors for ‘3d’ as 2 characters for left and right (default colors ‘rc’): chosen from r(ed), g(reen), b(lue), c(yan), m(agenta).
‘dispar’, D	Disparity for 3d display (default 0.1)
‘text’	Enable display of X,Y,Z labels on the frame
‘labels’, L	Label the X,Y,Z axes with the 1st, 2nd, 3rd character of the string L
‘rgb’	Display X,Y,Z axes in colors red, green, blue respectively
‘rviz’	Display chunky rviz style axes

## 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

- The ‘rviz’ option is equivalent to ‘rgb’, ‘notext’, ‘noarrow’, ‘thick’, 5.
- The arrow option requires the third party package arrow3 from File Exchange.
- The handle **H** is an hgtransform object.
- When using the form **trplot(H, ...)** to animate a frame it is best to set the axis bounds.
- The ‘3d’ option requires that the plot is viewed with anaglyph glasses.
- You cannot specify ‘color’ and ‘3d’ at the same time.

## See also

[trplot2](#), [tranimate](#)

---

# trplot2

## Plot a planar transformation

**trplot2(T, options)** draws a 2D coordinate frame represented by the SE(2) homogeneous transform **T** ( $3 \times 3$ ).

**H = trplot2(T, options)** as above but returns a handle.

**trplot2(H, T)** moves the coordinate frame described by the handle **H** to the SE(2) pose **T** ( $3 \times 3$ ).

## Options

‘axis’, A	Set dimensions of the MATLAB axes to A=[xmin xmax ymin ymax]
‘color’, c	The color to draw the axes, MATLAB colorspec
‘noaxes’	Don’t display axes on the plot
‘frame’, F	The frame is named {F} and the subscript on the axis labels is F.
‘text_opts’, opt	A cell array of Matlab text properties
‘handle’, h	Draw in the MATLAB axes specified by h
‘view’, V	Set plot view parameters V=[az el] angles, or ‘auto’ for view toward origin of coordinate frame
‘length’, s	Length of the coordinate frame arms (default 1)
‘arrow’	Use arrows rather than line segments for the axes
‘width’, w	Width of arrow tips

## Examples

```
trplot2(T, 'frame', 'A')
trplot2(T, 'frame', 'A', 'color', 'b')
trplot2(T1, 'frame', 'A', 'text_opts', {'FontSize', 10, 'FontWeight', 'bold'})
```

## Notes

- The arrow option requires the third party package arrow3 from File Exchange.
- When using the form TRPLOT(H, ...) to animate a frame it is best to set the axis bounds.

## See also

[trplot](#)

---

# trprint

## Compact display of homogeneous transformation

**trprint(T, options)** displays the homogeneous transform in a compact single-line format. If T is a homogeneous transform sequence then each element is printed on a separate line.

s = **trprint(T, options)** as above but returns the string.

**trprint** T is the command line form of above, and displays in RPY format.

## Options

'rpy'	display with rotation in roll/pitch/yaw angles (default)
'euler'	display with rotation in ZYX Euler angles
'angvec'	display with rotation in angle/vector format
'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

```
>> trprint(T2)
t = (0,0,0), RPY = (-122.704,65.4084,-8.11266) deg
>> trprint(T1, 'label', 'A')
A:t = (0,0,0), RPY = (-0,0,-0) deg
```

## See also

[tr2eul](#), [tr2rpy](#), [tr2angvec](#)

---

# trscole

## Homogeneous transformation for pure scale

**T = trscale(s)** is a homogeneous transform ( $4 \times 4$ ) 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.

---

# unit

## Unitize a vector

**vn = unit(v)** is a **unit**-vector parallel to v.

## Note

- Reports error for the case where  $\text{norm}(\mathbf{v})$  is zero.
- 

# Vehicle

## Car-like vehicle class

This class models the kinematics of a car-like vehicle (bicycle model) on a plane that moves in SE(2). 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: $x, y, \theta$ ( $3 \times 1$ )
V	odometry covariance ( $2 \times 2$ )
odometry	distance moved in the last interval ( $2 \times 1$ )
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

Create a vehicle with odometry covariance

```
v = Vehicle( 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.update([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 waypoints.

## 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](#)

## Vehicle.Vehicle

### Vehicle object constructor

**v = Vehicle(v\_act, options)** creates a **Vehicle** object with actual odometry covariance **v\_act** ( $2 \times 2$ ) matrix corresponding to the odometry vector [dx dtheta].

### Options

‘stlim’, A	Steering angle limited to -A to +A (default 0.5 rad)
‘vmax’, S	Maximum speed (default 5m/s)
‘L’, L	Wheel base (default 1m)
‘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.

## 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 a 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 ( $1 \times 2$ ) = [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.f

### Predict next state based on odometry

$\mathbf{xn} = \text{V.f}(\mathbf{x}, \mathbf{odo})$  is the predicted next state  $\mathbf{xn}$  ( $1 \times 3$ ) based on current state  $\mathbf{x}$  ( $1 \times 3$ ) and odometry  $\mathbf{odo}$  ( $1 \times 2$ ) = [distance, heading\_change].

$\mathbf{xn} = \text{V.f}(\mathbf{x}, \mathbf{odo}, \mathbf{w})$  as above but with odometry noise  $\mathbf{w}$ .

**Notes**

- Supports vectorized operation where  $\mathbf{x}$  and  $\mathbf{xn}$  ( $N \times 3$ ).
- 

## Vehicle.Fv

### Jacobian df/dv

$\mathbf{J} = \text{V.Fv}(\mathbf{x}, \mathbf{odo})$  is the Jacobian df/dv ( $3 \times 2$ ) at the state  $\mathbf{x}$ , for odometry input  $\mathbf{odo}$  ( $1 \times 2$ ) = [distance, heading\_change].

**See also**

[Vehicle.F](#), [Vehicle.Fx](#)

---

## Vehicle.Fx

### Jacobian df/dx

$\mathbf{J} = \text{V.Fx}(\mathbf{x}, \mathbf{odo})$  is the Jacobian df/dx ( $3 \times 3$ ) at the state  $\mathbf{x}$ , for odometry input  $\mathbf{odo}$  ( $1 \times 2$ ) = [distance, heading\_change].

**See also**

[Vehicle.f](#), [Vehicle.Fv](#)

---

## Vehicle.init

### Reset state of vehicle object

`V.init()` sets the state  $V.x := V.x_0$ , initializes the driver object (if attached) and clears the history.

`V.init(x0)` as above but the state is initialized to  $x_0$ .

---

## Vehicle.plot

### Plot vehicle

`V.plot(options)` plots the vehicle on the current axes at a pose given by the current state. If the vehicle has been previously plotted its pose is updated. The vehicle is depicted as a narrow triangle that travels “point first” and has a length  $V.rdim$ .

`V.plot(x, options)` plots the vehicle on the current axes at the pose  $x$ .

`H = V.plotv(x, options)` draws a representation of a ground robot as an oriented triangle with pose  $x$  ( $1 \times 3$ ) [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

### See also

[Vehicle.plotv](#)

---

## 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` ( $1 \times 3$ ) [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
‘color’, C	Color of vehicle.
‘fill’	Filled with solid color as per ‘color’ option
‘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** × 3) without plotting. Each row is (x,y,theta).

### See also

[Vehicle.step](#)

---

## 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.

---

# vex

## Convert skew-symmetric matrix to vector

**v** = **vex**(s) is the vector ( $3 \times 1$ ) which has the skew-symmetric matrix s ( $3 \times 3$ )

$$\begin{vmatrix} 0 & -vz & vy \\ vz & 0 & -vx \\ -vy & vx & 0 \end{vmatrix}$$

## Notes

- This is the inverse of the function SKEW().
- No checking 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, ie.  $vx = 0.5*(s(3,2)-s(2,3))$

## See also

[skew](#)

---

# VREP

## V-REP simulator communications object

A VREP object holds all information related to the state of a connection to an instance of the V-REP simulator running on this or a networked computer. Allows the creation of references to other objects/models in V-REP which can be manipulated in MATLAB.

This class handles the interface to the simulator and low-level object handle operations.

Methods throw exception if an error occurs.

## Methods

gethandle	get handle to named object
getchildren	get children belonging to handle
getobjname	get names of objects
object	return a VREP_obj object for named object
arm	return a VREP_arm object for named robot
camera	return a VREP_camera object for named vision sensor
hokuyo	return a VREP_hokuyo object for named Hokuyo scanner
getpos	return position of object given handle
setpos	set position of object given handle
getorient	return orientation of object given handle
setorient	set orientation of object given handle
getpose	return pose of object given handle
setpose	set pose of object given handle
setobjparam_bool	set object boolean parameter
setobjparam_int	set object integer parameter
setobjparam_float	set object float parameter
getobjparam_bool	get object boolean parameter
getobjparam_int	get object integer parameter
getobjparam_float	get object float parameter
signal_int	send named integer signal
signal_float	send named float signal
signal_str	send named string signal
setparam_bool	set simulator boolean parameter
setparam_int	set simulator integer parameter
setparam_str	set simulator string parameter
setparam_float	set simulator float parameter
getparam_bool	get simulator boolean parameter
getparam_int	get simulator integer parameter
getparam_str	get simulator string parameter
getparam_float	get simulator float parameter
delete	shutdown the connection and cleanup
simstart	start the simulator running
simstop	stop the simulator running
simpause	pause the simulator
getversion	get V-REP version number
checkcomms	return status of connection
pausecomms	pause the comms
loadscene	load a scene file
clearscene	clear the current scene
loadmodel	load a model into current scene
display	print the link parameters in human readable form
char	convert to string

## See also

[VREP\\_obj](#), [VREP\\_arm](#), [VREP\\_camera](#), [VREP\\_hokuyo](#)

## VREP.VREP

### VREP object constructor

**v = VREP(options)** create a connection to an instance of the V-REP simulator.

### Options

‘timeout’, T	Timeout T in ms (default 2000)
‘cycle’, C	Cycle time C in ms (default 5)
‘port’, P	Override communications port
‘reconnect’	Reconnect on error (default noreconnect)
‘path’, P	The path to VREP install directory

### Notes

- The default path is taken from the environment variable VREP
- 

## VREP.arm

### Return VREP\_arm object

**V.arm(name)** is a factory method that returns a VREP\_arm object for the V-REP robot object named NAME.

### Example

```
vrep.arm('IRB_140');
```

### See also

[VREP\\_arm](#)

---

## VREP.camera

### Return VREP\_camera object

V.**camera**(name) is a factory method that returns a VREP\_camera object for the V-REP vision sensor object named NAME.

#### See also

[VREP\\_camera](#)

---

## VREP.char

### Convert to string

V.**char**() is a string representation the VREP parameters in human readable format.

#### See also

[VREP.display](#)

---

## VREP.checkcomms

### Check communications to V-REP simulator

V.**checkcomms**() is true if a valid connection to the V-REP simulator exists.

---

## VREP.clearscene

### Clear current scene in the V-REP simulator

V.**clearscene**() clears the current scene and switches to another open scene, if none, a new (default) scene is created.

#### See also

[VREP.loadscene](#)

---

## VREP.delete

### VREP object destructor

**delete(v)** closes the connection to the V-REP simulator

---

## VREP.display

### Display parameters

V.**display()** displays the VREP parameters in compact format.

### Notes

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

### See also

[VREP.char](#)

---

## VREP.getchildren

### Find children of object

C = V.**getchildren(H)** is a vector of integer handles for the children of the V-REP object denoted by the integer handle H.

---

## VREP.gethandle

### Return handle to VREP object

H = V.**gethandle(name)** is an integer handle for named V-REP object.

H = V.**gethandle(fmt, arglist)** as above but the name is formed from sprintf(fmt, arglist).

**See also**

[sprintf](#)

---

## VREP.getjoint

### Get value of V-REP joint object

`V.getjoint(H, q)` is the position of joint object with integer handle `H`.

---

## VREP.getobjname

### Find names of objects

`V.getobjname()` will display the names and object handle (integers) for all objects in the current scene.

`name = V.getobjname(H)` will return the name of the object with handle `H`.

---

## VREP.getobjparam\_bool

### Get boolean parameter of a V-REP object

`V.getobjparam_bool(H, param)` gets the boolean parameter with identifier `param` of object with integer handle `H`.

---

## VREP.getobjparam\_float

### Get float parameter of a V-REP object

`V.getobjparam_float(H, param)` gets the float parameter with identifier `param` of object with integer handle `H`.

---

## VREP.getobjparam\_int

### Get integer parameter of a V-REP object

`V.getobjparam_int(H, param)` gets the integer parameter with identifier `param` of object with integer handle `H`.

---

## VREP.getorient

### Get orientation of V-REP object

`R = V.getorient(H)` is the orientation of the V-REP object with integer handle `H` as a rotation matrix ( $3 \times 3$ ).

`EUL = V.getorient(H, 'euler', OPTIONS)` as above but returns ZYZ Euler angles.

`V.getorient(H, hrr)` as above but orientation is relative to the position of object with integer handle HR.

`V.getorient(H, hrr, 'euler', OPTIONS)` as above but returns ZYZ Euler angles.

### Options

See `tr2eul`.

### See also

[VREP.setorient](#), [VREP.getpos](#), [VREP.getpose](#)

---

## VREP.getparam\_bool

### Get boolean parameter of the V-REP simulator

`V.getparam_bool(name)` is the boolean parameter with name `name` from the V-REP simulation engine.

### Example

```
v = VREP();  
v.getparam_bool('sim_boolparam_mirrors_enabled')
```

**See also**

[VREP.setparam\\_bool](#)

---

## VREP.getparam\_float

### Get float parameter of the V-REP simulator

`V.getparam_float(name)` gets the float parameter with name **name** from the V-REP simulation engine.

**Example**

```
v = VREP();
v.getparam_float('sim_floatparam_simulation_time_step')
```

**See also**

[VREP.setparam\\_float](#)

---

## VREP.getparam\_int

### Get integer parameter of the V-REP simulator

`V.getparam_int(name)` is the integer parameter with name **name** from the V-REP simulation engine.

**Example**

```
v = VREP();
v.getparam_int('sim_intparam_settings')
```

**See also**

[VREP.setparam\\_int](#)

---

## VREP.getparam\_str

### Get string parameter of the V-REP simulator

`V.getparam_str(name)` is the string parameter with name `name` from the V-REP simulation engine.

#### Example

```
v = VREP();  
v.getparam_str('sim_stringparam_application_path')
```

#### See also

[VREP.setparam\\_str](#)

---

## VREP.getpos

### Get position of V-REP object

`V.getpos(H)` is the position ( $1 \times 3$ ) of the V-REP object with integer handle `H`.

`V.getpos(H, hr)` as above but position is relative to the position of object with integer handle `hr`.

#### See also

[VREP.setpose](#), [VREP.getpose](#), [VREP.getorient](#)

---

## VREP.getpose

### Get pose of V-REP object

`T = V.getpose(H)` is the pose of the V-REP object with integer handle `H` as a homogeneous transformation matrix ( $4 \times 4$ ).

`T = V.getpose(H, hr)` as above but pose is relative to the pose of object with integer handle `R`.

**See also**

[VREP.setpose](#), [VREP.getpos](#), [VREP.getorient](#)

---

## VREP.getversion

### Get version of the V-REP simulator

`V.getversion()` is the version of the V-REP simulator server as an integer MNNNN where M is the major version number and NNNN is the minor version number.

---

## VREP.hokuyo

### Return VREP\_hokuyo object

`V.hokuyo(name)` is a factory method that returns a VREP\_hokuyo object for the V-REP Hokuyo laser scanner object named NAME.

---

**See also**

[VREP\\_hokuyo](#)

---

## VREP.loadmodel

### Load a model into the V-REP simulator

`m = V.loadmodel(file, options)` loads the model file `file` with extension .ttm into the simulator and returns a VREP\_obj object that mirrors it in MATLAB.

#### Options

‘local’    The file is loaded relative to the MATLAB client’s current folder, otherwise from the V-REP root folder.

#### Example

```
vrep.loadmodel('people/Walking_Bill');
```

## Notes

- If a relative filename is given in non-local (server) mode it is relative to the V-REP models folder.

## See also

[VREP.arm](#), [VREP.camera](#), [VREP.object](#)

---

# VREP.loadscene

## Load a scene into the V-REP simulator

**V.loadscene(file, options)** loads the scene file **file** with extension .ttt into the simulator.

## Options

- ‘local’ The file is loaded relative to the MATLAB client’s current folder, otherwise from the V-REP root folder.

## Example

```
vrep.loadscene('2IndustrialRobots');
```

## Notes

- If a relative filename is given in non-local (server) mode it is relative to the V-REP scenes folder.

## See also

[VREP.clearscene](#)

---

# VREP.mobile

## Return VREP\_mobile object

**V.mobile(name)** is a factory method that returns a VREP\_mobile object for the V-REP **mobile** base object named NAME.

**See also**

[VREP\\_mobile](#)

---

## VREP.object

**Return VREP\_obj object**

`V.objet(name)` is a factory method that returns a VREP\_obj object for the V-REP object or model named NAME.

**Example**

```
vrep.obj('Walking Bill');
```

**See also**

[VREP\\_obj](#)

---

## VREP.pausecomms

**Pause communications to the V-REP simulator**

`V.pausecomms(p)` pauses communications to the V-REP simulation engine if **p** is true else resumes it. Useful to ensure an atomic update of simulator state.

---

## VREP.setjoint

**Set value of V-REP joint object**

`V.setjoint(H, q)` sets the position of joint object with integer handle **H** to the value **q**.

---

## VREP.setjointtarget

**Set target value of V-REP joint object**

`V.setjointtarget(H, q)` sets the target position of joint object with integer handle **H** to the value **q**.

---

## VREP.setjointvel

### Set velocity of V-REP joint object

`V.setjointvel(H, qd)` sets the target velocity of joint object with integer handle **H** to the value **qd**.

---

## VREP.setobjparam\_bool

### Set boolean parameter of a V-REP object

`V.setobjparam_bool(H, param, val)` sets the boolean parameter with identifier **param** of object **H** to value **val**.

---

## VREP.setobjparam\_float

### Set float parameter of a V-REP object

`V.setobjparam_float(H, param, val)` sets the float parameter with identifier **param** of object **H** to value **val**.

---

## VREP.setobjparam\_int

### Set Integer parameter of a V-REP object

`V.setobjparam_int(H, param, val)` sets the integer parameter with identifier **param** of object **H** to value **val**.

---

## VREP.setorient

### Set orientation of V-REP object

`V.setorient(H, R)` sets the orientation of V-REP object with integer handle **H** to that given by rotation matrix **R** ( $3 \times 3$ ).

`V.setorient(H, T)` sets the orientation of V-REP object with integer handle **H** to rotational component of homogeneous transformation matrix **T** ( $4 \times 4$ ).

`V.setorient(H, E)` sets the orientation of V-REP object with integer handle **H** to ZYZ Euler angles ( $1 \times 3$ ).

`V.setorient(H, x, hr)` as above but orientation is set relative to the orientation of object with integer handle `hr`.

**See also**

[VREP.getorient](#), [VREP.setpos](#), [VREP.setpose](#)

---

## **VREP.setparam\_bool**

### **Set boolean parameter of the V-REP simulator**

`V.setparam_bool(name, val)` sets the boolean parameter with name `name` to value `val` within the V-REP simulation engine.

**See also**

[VREP.getparam\\_bool](#)

---

## **VREP.setparam\_float**

### **Set float parameter of the V-REP simulator**

`V.setparam_float(name, val)` sets the float parameter with name `name` to value `val` within the V-REP simulation engine.

**See also**

[VREP.getparam\\_float](#)

---

## **VREP.setparam\_int**

### **Set integer parameter of the V-REP simulator**

`V.setparam_int(name, val)` sets the integer parameter with name `name` to value `val` within the V-REP simulation engine.

**See also**

[VREP.getparam\\_int](#)

---

## VREP.setparam\_str

### Set string parameter of the V-REP simulator

**V.setparam\_str(name, val)** sets the integer parameter with name **name** to value **val** within the V-REP simulation engine.

**See also**

[VREP.getparam\\_str](#)

---

## VREP.setpos

### Set position of V-REP object

**V.setpos(H, T)** sets the position of V-REP object with integer handle **H** to **T** ( $1 \times 3$ ).

**V.setpos(H, T, hr)** as above but position is set relative to the position of object with integer handle **hr**.

**See also**

[VREP.getpos](#), [VREP.setpose](#), [VREP.setorient](#)

---

## VREP.setpose

### Set pose of V-REP object

**V.setpos(H, T)** sets the pose of V-REP object with integer handle **H** according to homogeneous transform **T** ( $4 \times 4$ ).

**V.setpos(H, T, hr)** as above but pose is set relative to the pose of object with integer handle **hr**.

**See also**

[VREP.getpose](#), [VREP.setpos](#), [VREP.setorient](#)

---

## VREP.signal\_float

### Send a float signal to the V-REP simulator

`V.signal_float(name, val)` send a float signal with name **name** and value **val** to the V-REP simulation engine.

---

## VREP.signal\_int

### Send an integer signal to the V-REP simulator

`V.signal_int(name, val)` send an integer signal with name **name** and value **val** to the V-REP simulation engine.

---

## VREP.signal\_str

### Send a string signal to the V-REP simulator

`V.signal_str(name, val)` send a string signal with name **name** and value **val** to the V-REP simulation engine.

---

## VREP.simpause

### Pause V-REP simulation

`V.simpause()` pauses the V-REP simulation engine. Use `V.simstart()` to resume the simulation.

---

**See also**

[VREP.simstart](#)

---

## VREP.simstart

### Start V-REP simulation

V.**simstart()** starts the V-REP simulation engine.

#### See also

[VREP.simstop](#), [VREP.simpause](#)

---

## VREP.simstop

### Stop V-REP simulation

V.**simstop()** stops the V-REP simulation engine.

#### See also

[VREP.simstart](#)

---

## VREP.youbot

### Return VREP\_youbot object

V.**youbot(name)** is a factory method that returns a VREP\_youbot object for the V-REP YouBot object named NAME.

#### See also

[vrep\\_youbot](#)

---

## VREP\_arm

### Mirror of V-REP robot arm object

Mirror objects are MATLAB objects that reflect the state of objects in the V-REP environment. Methods allow the V-REP state to be examined or changed.

This is a concrete class, derived from VREP\_mirror, for all V-REP robot arm objects and allows access to joint variables.

Methods throw exception if an error occurs.

## Example

```
vrep = VREP();
arm = vrep.arm('IRB140');
q = arm.getq();
arm.setq(zeros(1,6));
arm.setpose(T); % set pose of base
```

## Methods

getq	get joint coordinates
setq	set joint coordinates
setjointmode	set joint control parameters
animate	animate a joint coordinate trajectory
teach	graphical teach pendant

## Superclass methods (VREP\_obj)

getpos	get position of object
setpos	set position of object
getorient	get orientation of object
setorient	set orientation of object
getpose	get pose of object given
setpose	set pose of object

can be used to set/get the pose of the robot base.

## Superclass methods (VREP\_mirror)

getname	get object name
setparam_bool	set object boolean parameter
setparam_int	set object integer parameter
setparam_float	set object float parameter
getparam_bool	get object boolean parameter
getparam_int	get object integer parameter
getparam_float	get object float parameter

## Properties

n Number of joints

### See also

[VREP\\_mirror](#), [VREP\\_obj](#), [VREP\\_arm](#), [VREP\\_camera](#), [VREP\\_hokuyo](#)

---

## VREP\_arm.VREP\_arm

### Create a robot arm mirror object

**arm = VREP\_arm(name, options)** is a mirror object that corresponds to the robot arm named **name** in the V-REP environment.

### Options

‘fmt’, F Specify format for joint object names (default ‘%s.joint%d’)

### Notes

- The number of joints is found by searching for objects with names systematically derived from the root object name, by default named NAME\_N where N is the joint number starting at 0.

### See also

[VREP.arm](#)

---

## VREP\_arm.animate

### Animate V-REP robot

R.**animate(qt, options)** animates the corresponding V-REP robot with configurations taken from consecutive rows of **qt** ( $M \times N$ ) which represents an M-point trajectory and N is the number of robot joints.

### Options

‘delay’, D Delay (s) between frames for animation (default 0.1)  
‘fps’, fps Number of frames per second for display, inverse of ‘delay’ option  
‘[no]loop’ Loop over the trajectory forever

**See also**

[SerialLink.plot](#)

---

## VREP\_arm.getq

### Get joint angles of V-REP robot

ARM.**getq()** is the vector of joint angles ( $1 \times N$ ) from the corresponding robot arm in the V-REP simulation.

**See also**

[VREP\\_arm.setq](#)

---

## VREP\_arm.setjointmode

### Set joint mode

ARM.**setjointmode(m, C)** sets the motor enable **m** (0 or 1) and motor control **C** (0 or 1) parameters for all joints of this robot arm.

---

## VREP\_arm.setq

### Set joint angles of V-REP robot

ARM.**setq(q)** sets the joint angles of the corresponding robot arm in the V-REP simulation to **q** ( $1 \times N$ ).

**See also**

[VREP\\_arm.getq](#)

---

## VREP\_arm.setqt

### Set joint angles of V-REP robot

ARM.**setq(q)** sets the joint angles of the corresponding robot arm in the V-REP simulation to  $\mathbf{q}$  ( $1 \times N$ ).

---

## VREP\_arm.teach

### Graphical teach pendant

R.**teach(options)** drive a V-REP robot by means of a graphical slider panel.

### Options

'degrees'	Display angles in degrees (default radians)
'q0', q	Set initial joint coordinates

### Notes

- The slider limits are all assumed to be [-pi, +pi]

### See also

[SerialLink.plot](#)

---

## VREP\_camera

### Mirror of V-REP vision sensor object

Mirror objects are MATLAB objects that reflect the state of objects in the V-REP environment. Methods allow the V-REP state to be examined or changed.

This is a concrete class, derived from VREP\_mirror, for all V-REP vision sensor objects and allows access to images and image parameters.

Methods throw exception if an error occurs.

## Example

```
vrep = VREP();
camera = vrep.camera('Vision_sensor');
im = camera.grab();
camera.setpose(T);
R = camera.getorient();
```

## Methods

grab	return an image from simulated camera
setangle	set field of view
setresolution	set image resolution
setclipping	set clipping boundaries

## Superclass methods (VREP\_obj)

getpos	get position of object
setpos	set position of object
getorient	get orientation of object
setorient	set orientation of object
getpose	get pose of object
setpose	set pose of object

can be used to set/get the pose of the robot base.

## Superclass methods (VREP\_mirror)

getname	get object name
setparam_bool	set object boolean parameter
setparam_int	set object integer parameter
setparam_float	set object float parameter
getparam_bool	get object boolean parameter
getparam_int	get object integer parameter
getparam_float	get object float parameter

## See also

[VREP\\_mirror](#), [VREP\\_obj](#), [VREP\\_arm](#), [VREP\\_camera](#), [VREP\\_hokuyo](#)

---

## VREP\_camera.VREP\_camera

### Create a camera mirror object

C = **VREP\_camera**(name, options) is a mirror object that corresponds to the vision sensor named name in the V-REP environment.

### Options

'fov', A	Specify field of view in degrees (default 60)
'resolution', N	Specify resolution. If scalar $N \times N$ else $N(1) \times N(2)$
'clipping', Z	Specify near Z(1) and far Z(2) clipping boundaries

### Notes

- Default parameters are set in the V-REP environment
- Can be applied to “DefaultCamera” which controls the view in the simulator GUI.

### See also

[VREP\\_obj](#)

---

## VREP\_camera.char

### Convert to string

V.**char**() is a string representation the VREP parameters in human readable format.

### See also

[VREP.display](#)

---

## VREP\_camera.getangle

### Get field of view for V-REP vision sensor

fov = C.**getangle**(fov) is the field-of-view angle to fov in radians.

**See also**

[VREP\\_camera.setangle](#)

---

## VREP\_camera.getclipping

### Get clipping boundaries for V-REP vision sensor

**C.getclipping()** is the near and far clipping boundaries ( $1 \times 2$ ) in the Z-direction as a 2-vector [NEAR,FAR].

**See also**

[VREP\\_camera.setclipping](#)

---

## VREP\_camera.getresolution

### Get resolution for V-REP vision sensor

**R = C.getresolution()** is the image resolution ( $1 \times 2$ ) of the vision sensor **R(1)xR(2)**.

**See also**

[VREP\\_camera.setresolution](#)

---

## VREP\_camera.grab

### Get image from V-REP vision sensor

**im = C.grab(options)** is an image ( $W \times H$ ) returned from the V-REP vision sensor.

**C.grab(options)** as above but the image is displayed using idisp.

#### Options

‘grey’ Return a greyscale image (default color).

## Notes

- V-REP simulator must be running.
- Color images can be quite dark, ensure good light sources.
- Uses the signal ‘handle\_rgb\_sensor’ to trigger a single image generation.

## See also

[idisp](#), [VREP.simstart](#)

---

# VREP\_camera.setangle

## Set field of view for V-REP vision sensor

C.**setangle**(fov) set the field-of-view angle to **fov** in radians.

## See also

[VREP\\_camera.getangle](#)

---

# VREP\_camera.setclipping

## Set clipping boundaries for V-REP vision sensor

C.**setclipping**(near, far) set clipping boundaries to the range of Z from **near** to **far**. Objects outside this range will not be rendered.

## See also

[VREP\\_camera.getclipping](#)

---

# VREP\_camera.setresolution

## Set resolution for V-REP vision sensor

C.**setresolution**(R) set image resolution to **R** × **R** if **R** is a scalar or **R**(1)x**R**(2) if it is a 2-vector.

## Notes

- By default V-REP cameras seem to have very low ( $32 \times 32$ ) resolution.
- Frame rate will decrease as frame size increases.

## See also

[VREP\\_camera.getresolution](#)

---

---

---

# VREP\_mirror

## V-REP mirror object class

Mirror objects are MATLAB objects that reflect the state of objects in the V-REP environment. Methods allow the V-REP state to be examined or changed.

This abstract class is the root class for all V-REP mirror objects.

Methods throw exception if an error occurs.

## Methods

getname	get object name
setparam_bool	set object boolean parameter
setparam_int	set object integer parameter
setparam_float	set object float parameter
getparam_bool	get object boolean parameter
getparam_int	get object integer parameter
getparam_float	get object float parameter
remove	remove object from scene
display	display object info
char	convert to string

## Properties (read only)

h	V-REP integer handle for the object
name	Name of the object in V-REP
vrep	Reference to the V-REP connection object

## Notes

- This has nothing to do with mirror objects in V-REP itself which are shiny reflective surfaces.

## See also

[VREP\\_obj](#), [VREP\\_arm](#), [VREP\\_camera](#), [VREP\\_hokuyo](#)

---

# VREP\_mirror.VREP\_mirror

## Construct VREP\_mirror object

**obj = VREP\_mirror(name)** is a V-REP mirror object that represents the object named **name** in the V-REP simulator.

---

# VREP\_mirror.char

## Convert to string

OBJ.**char()** is a string representation the VREP parameters in human readable format.

## See also

[VREP.display](#)

---

# VREP\_mirror.display

## Display parameters

OBJ.**display()** displays the VREP parameters in compact format.

## Notes

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

**See also**

[VREP.char](#)

---

## VREP\_mirror.getname

### Get object name

OBJ.**getname()** is the name of the object in the VREP simulator.

---

## VREP\_mirror.getparam\_bool

### Get boolean parameter of V-REP object

OBJ.**getparam\_bool(id)** is the boolean parameter with **id** of the corresponding V-REP object.

See also [VREP\\_mirror.setparam\\_bool](#), [VREP\\_mirror.getparam\\_int](#), [VREP\\_mirror.getparam\\_float](#).

---

## VREP\_mirror.getparam\_float

### Get float parameter of V-REP object

OBJ.**getparam\_float(id)** is the float parameter with **id** of the corresponding V-REP object.

See also [VREP\\_mirror.setparam\\_bool](#), [VREP\\_mirror.getparam\\_bool](#), [VREP\\_mirror.getparam\\_int](#).

---

## VREP\_mirror.getparam\_int

### Get integer parameter of V-REP object

OBJ.**getparam\_int(id)** is the integer parameter with **id** of the corresponding V-REP object.

See also [VREP\\_mirror.setparam\\_int](#), [VREP\\_mirror.getparam\\_bool](#), [VREP\\_mirror.getparam\\_float](#).

---

## VREP\_mirror.setparam\_bool

### Set boolean parameter of V-REP object

OBJ.**setparam\_bool**(**id**, **val**) sets the boolean parameter with **id** to value **val** within the V-REP simulation engine.

See also **VREP\_mirror.getparam\_bool**, **VREP\_mirror.setparam\_int**, **VREP\_mirror.setparam\_float**.

---

## VREP\_mirror.setparam\_float

### Set float parameter of V-REP object

OBJ.**setparam\_float**(**id**, **val**) sets the float parameter with **id** to value **val** within the V-REP simulation engine.

See also **VREP\_mirror.getparam\_float**, **VREP\_mirror.setparam\_bool**, **VREP\_mirror.setparam\_int**.

---

## VREP\_mirror.setparam\_int

### Set integer parameter of V-REP object

OBJ.**setparam\_int**(**id**, **val**) sets the integer parameter with **id** to value **val** within the V-REP simulation engine.

See also **VREP\_mirror.getparam\_int**, **VREP\_mirror.setparam\_bool**, **VREP\_mirror.setparam\_float**.

---

---

---

## VREP\_obj

### V-REP mirror of simple object

Mirror objects are MATLAB objects that reflect objects in the V-REP environment. Methods allow the V-REP state to be examined or changed.

This is a concrete class, derived from VREP\_mirror, for all V-REP objects and allows access to pose and object parameters.

## Example

```
vrep = VREP();
bill = vrep.object('Bill'); % get the human figure Bill
bill.setpos([1,2,0]);
bill.setorient([0 pi/2 0]);
```

Methods throw exception if an error occurs.

## Methods

getpos	get position of object
setpos	set position of object
getorient	get orientation of object
setorient	set orientation of object
getpose	get pose of object
setpose	set pose of object

## Superclass methods (VREP\_mirror)

getname	get object name
setparam_bool	set object boolean parameter
<b>setparam_int</b>	set object integer parameter
setparam_float	set object float parameter
getparam_bool	get object boolean parameter
getparam_int	get object integer parameter
getparam_float	get object float parameter
display	print the link parameters in human readable form
char	convert to string

## See also

[VREP\\_mirror](#), [VREP\\_obj](#), [VREP\\_arm](#), [VREP\\_camera](#), [VREP\\_hokuyo](#)

---

# VREP\_obj.VREP\_obj

## VREP\_obj mirror object constructor

**v = VREP\_base(name)** creates a V-REP mirror object for a simple V-REP object type.

---

## VREP\_obj.getorient

### Get orientation of V-REP object

`V.getorient()` is the orientation of the corresponding V-REP object as a rotation matrix ( $3 \times 3$ ).

`V.getorient('euler', OPTIONS)` as above but returns ZYZ Euler angles.

`V.getorient(base)` is the orientation of the corresponding V-REP object relative to the **VREP\_obj** object **base**.

`V.getorient(base, 'euler', OPTIONS)` as above but returns ZYZ Euler angles.

### Options

See `tr2eul`.

### See also

[VREP\\_obj.setorient](#), [VREP\\_obj.getpos](#), [VREP\\_obj.getpose](#)

---

## VREP\_obj.getpos

### Get position of V-REP object

`V.getpos()` is the position ( $1 \times 3$ ) of the corresponding V-REP object.

`V.getpos(base)` as above but position is relative to the **VREP\_obj** object **base**.

### See also

[VREP\\_obj.setpos](#), [VREP\\_obj.getorient](#), [VREP\\_obj.getpose](#)

---

## VREP\_obj.getpose

### Get pose of V-REP object

`V.getpose()` is the pose ( $4 \times 4$ ) of the the corresponding V-REP object.

`V.getpose(base)` as above but pose is relative to the pose the **VREP\_obj** object **base**.

**See also**

[VREP\\_obj.setpose](#), [VREP\\_obj.getorient](#), [VREP\\_obj.getpos](#)

---

## VREP\_obj.setorient

### Set orientation of V-REP object

**V.setorient(R)** sets the orientation of the corresponding V-REP to rotation matrix **R** ( $3 \times 3$ ).

**V.setorient(T)** sets the orientation of the corresponding V-REP object to rotational component of homogeneous transformation matrix **T** ( $4 \times 4$ ).

**V.setorient(E)** sets the orientation of the corresponding V-REP object to ZYZ Euler angles ( $1 \times 3$ ).

**V.setorient(x, base)** as above but orientation is set relative to the orientation of **VREP\_obj** object **base**.

**See also**

[VREP\\_obj.getorient](#), [VREP\\_obj.setpos](#), [VREP\\_obj.setpose](#)

---

## VREP\_obj.setpos

### Set position of V-REP object

**V.setpos(T)** sets the position of the corresponding V-REP object to **T** ( $1 \times 3$ ).

**V.setpos(T, base)** as above but position is set relative to the position of the **VREP\_obj** object **base**.

**See also**

[VREP\\_obj.getpos](#), [VREP\\_obj.setorient](#), [VREP\\_obj.setpose](#)

---

## VREP\_obj.setpose

### Set pose of V-REP object

**V.setpose(T)** sets the pose of the corresponding V-REP object to **T** ( $4 \times 4$ ).

**V.setpose(T, base)** as above but pose is set relative to the pose of the **VREP\_obj** object base.

### See also

[VREP\\_obj.getpose](#), [VREP\\_obj.setorient](#), [VREP\\_obj.setpos](#)

---

---

## wtrans

### Transform a wrench between coordinate frames

**wt = wtrans(T, w)** is a wrench ( $6 \times 1$ ) in the frame represented by the homogeneous transform **T** ( $4 \times 4$ ) corresponding to the world frame wrench **w** ( $6 \times 1$ ).

The wrenches **w** and **wt** are 6-vectors of the form [Fx Fy Fz Mx My Mz]’.

### See also

[tr2delta](#), [tr2jac](#)

---

## 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](#)

---

## **xyzlabel**

### **Label X, Y and Z axes**

**XYZLABEL** label the x-, y- and z-axes with ‘X’, ‘Y’, and ‘Z’ respectively

---

## **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](#)

---