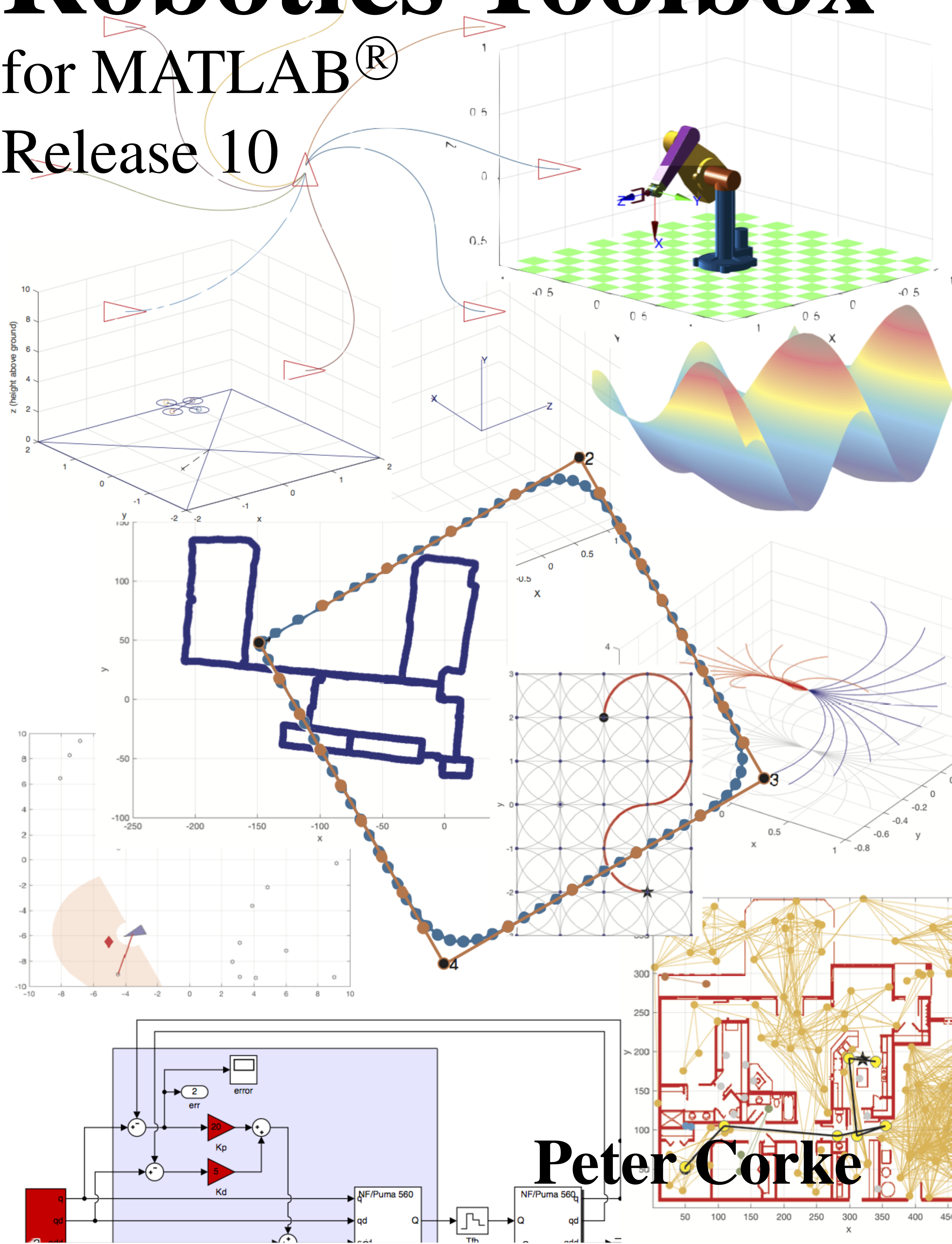
Robotics Toolbox

for MATLAB®

Release 10

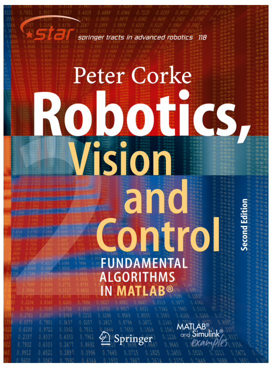


Peter Corke

Release	10.4
Release date	October 2019

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

Preface


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

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

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

The Toolbox makes strong use of classes to represent robots and such things as sensors and maps. It includes Simulink® models to describe the evolution of arm or mobile robot state over time for a number of classical control strategies. The Toolbox also provides functions for manipulating and converting between datatypes such as vectors, rotation matrices, unit-quaternions, quaternions, homogeneous transformations and twists which are necessary to represent position and orientation in 2- and 3-dimensions.

The code is written in a straightforward manner which allows for easy understanding, perhaps at the expense of computational efficiency. If you feel strongly about computational efficiency then you can always rewrite the function to be more efficient, compile the M-file using the MATLAB compiler, or create a MEX version.

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

Functions by category

Contents

Chapter 1

Introduction

1.1 Changes in RTB 10

RTB 10 is largely backward compatible with RTB 9.

1.1.1 Incompatible changes

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

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

Table 1.1: Function and method name changes

These changes are summarized in Table 1.1.

1.1.2 New features

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

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

Since its inception RTB has used matrices[1] to represent rotations and transformations in 2D and 3D. A trajectory, or sequence, was represented by a 3-dimensional matrix, eg. $4 \times 4 \times N$. In RTB10 a set of classes have been introduced to represent orientation and pose in 2D and 3D: `SO2`, `SE2`, `SO3`, `SE3`, `Twist` and `UnitQuaternion`. These classes are fairly polymorphic, that is, they share many methods and operators[2]. All have a number of static methods that serve as constructors from particular representations. A trajectory is represented by a vector of these objects which makes code easier to read and understand. Overloaded operators are used so the classes behave in a similar way to native matrices[3]. The relationship between the classical Toolbox functions and the new classes are shown in Fig 1.1.

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

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

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

1.1.3 Enhancements

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

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

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

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

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

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

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

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

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

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

1.2 Changes in RTB 10.3

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

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

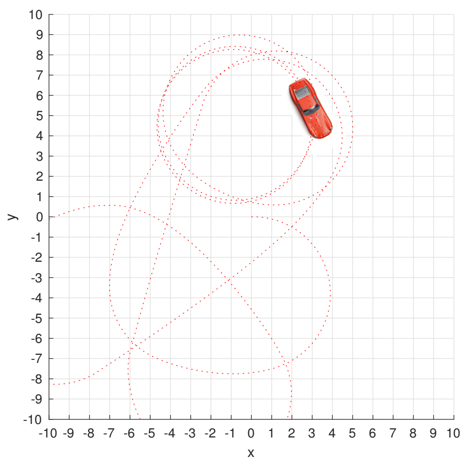


Figure 1.2: Car animation drawn with `demos/car_anim` using `plot_vehicle`

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

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

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

1.3 Changes in RTB 10.2

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

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

1.4 How to obtain the Toolbox

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

`http://www.petercorke.com`

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

1.4.1 From .mltbx file

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

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

1.4.2 From .zip file

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

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

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

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

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

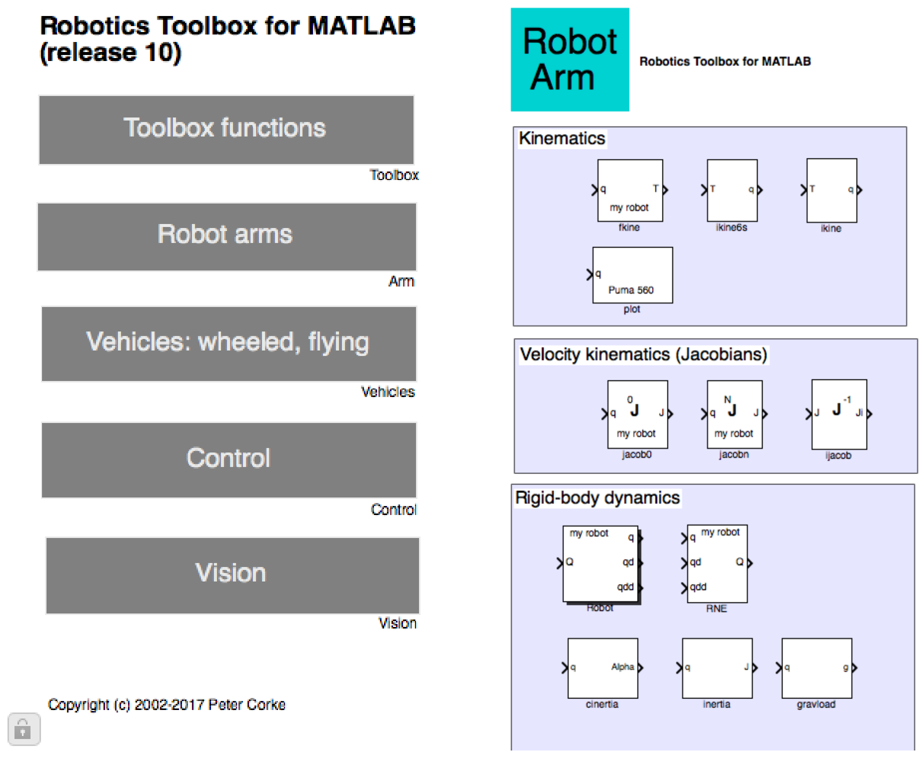
You need to setup the path every time you start MATLAB but you can automate this by setting up environment variables, editing your `startup.m` script, using `pathtool` and saving the path, or by pressing the “Update Toolbox Path Cache" button under MATLAB General preferences. You can check the path using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

A menu-driven demonstration can be invoked by

```
>> rtbdemo
```

1.4.3 MATLAB Online$^{\text{TM}}$

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



(a) (b)

Figure 1.3: The Robotics Toolbox blockset.

1.4.4 Simulink®

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

```
>> roblocks
```

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

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

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

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

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

A number of models are also provided:

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

1.4.5 Notes on implementation and versions

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

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

1.4.6 Documentation

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

1.5 Compatible MATLAB versions

The Toolbox has been tested under R2018a and R2018bPRE. Compatibility problems are increasingly likely the older your version of MATLAB is.

1.6 Use in teaching

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

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

1.7 Use in research

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

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

or

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

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

1.8 Support

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

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

You might instead like to communicate with other users via the Google Group called "Robotics and Machine Vision Toolbox" http://tiny.cc/rvcforum

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

1.9 Related software

1.9.1 Robotics System Toolbox™

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

1.9.2 Octave

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

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

1.9.3 Machine Vision toolbox

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

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

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

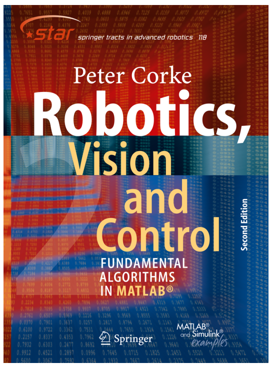
1.10 Contributing to the Toolboxes

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

1.11 Acknowledgements

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

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


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

Chapter 2

Functions and classes

Bicycle

Car-like vehicle class

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

Methods

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

Plotting/display methods

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

Properties (read/write)

x	true vehicle state: x, y, theta (3×1)
V	odometry covariance (2×2)
odometry	distance moved in the last interval (2×1)
rdim	dimension of the robot (for drawing)
L	length of the vehicle (wheelbase)
alphalim	steering wheel limit
maxspeed	maximum vehicle speed
T	sample interval
verbose	verbosity
x_hist	history of true vehicle state ($N \times 3$)
driver	reference to the driver object
x0	initial state, restored on init()

Examples

Odometry covariance (per timstep) is

```
V = diag([0.02, 0.5*pi/180].^2);
```

Create a vehicle with this noisy odometry

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

and display its initial state

```
v
```

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

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

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

```
v
```

We can add a driver object

```
v.add_driver( RandomPath(10) )
```

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

```
v.run(1000)
```

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

Notes

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

Reference

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

See also

RandomPath, EKF

Bicycle.Bicycle

Vehicle object constructor

v = **Bicycle**(**options**) creates a **Bicycle** object with the kinematics of a bicycle (or Ackerman) vehicle.

Options

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

Notes

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

Notes

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

Bicycle.char

Convert to a string

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

See also

Bicycle.display

Bicycle.deriv

Time derivative of state

dx = V.**deriv**(**T**, **x**, **u**) is the time derivative of state (3×1) at the state **x** (3×1) with input **u** (2×1).

Notes

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

Bicycle.f

Predict next state based on odometry

xn = V.**f**(**x**, **odo**) is the predicted next state **xn** (1×3) based on current state **x** (1×3) and odometry **odo** (1×2) = [distance, heading_change].

xn = V.**f**(**x**, **odo**, **w**) as above but with odometry noise **w**.

Notes

- Supports vectorized operation where **x** and **xn** ($N \times 3$).

Bicycle.Fv

Jacobian df/dv

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

See also

Bicycle.F, Vehicle.Fx

Bicycle.Fx

Jacobian df/dx

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

See also

Bicycle.f, Vehicle.Fv

Bicycle.update

Update the vehicle state

odo = V.**update**(**u**) is the true odometry value for motion with **u**=[speed,steer].

Notes

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

bresenham

Generate a line

p = **bresenham**(**x1**, **y1**, **x2**, **y2**) is a list of integer coordinates ($2 \times N$) for points lying on the line segment joining the integer coordinates (**x1**,**y1**) and (**x2**,**y2**).

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

Notes

- Endpoint coordinates must be integer values.

Author

- Based on code by Aaron Wetzler

See also

icanvas

Bug2

Bug navigation class

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

Methods

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

Example

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

Reference

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

See also

Navigation, DXform, Dstar, PRM

Bug2.Bug2

Construct a Bug2 navigation object

b = **Bug2**(**map**, **options**) is a bug2 navigation object, and **map** is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied).

Options

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

See also

Navigation.Navigation

Bug2.query

Find a path

B.**query**(**start**, **goal**, **options**) is the path ($N \times 2$) from **start** (1×2) to **goal** (1×2). Row are the coordinates of successive points along the path. If either **start** or **goal** is []

the grid map is displayed and the user is prompted to select a point by clicking on the plot.

Options

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

Notes

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

See also

animate

ccodefunctionstring

Converts a symbolic expression into a C-code function

[**funstr**, **hdrstr**] = **ccodefunctionstring**(**symexpr**, **arglist**) returns a string representing a C-code implementation of a symbolic expression **symexpr**. The C-code implementation has a signature of the form:

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

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

Options

'funname', name	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 accoringly 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

chi2inv_rtb

Inverse chi-squared function

x = **chi2inv_rtb**(**p**, **n**) is the inverse chi-squared CDF function of **n**-degrees of freedom.

Notes

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

See also

chi2inv

---

circle

Compute points on a circle

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

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

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

Options

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

CodeGenerator

Class for code generation

Objects of the CodeGenerator class automatcally 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
robjpath	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 (1×1).

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 down 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 genccode 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)

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 wirtten 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.genccodefriction

Generate C-code for the joint friction

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

Notes

- Is called by CodeGenerator.genfriction if cGen has active flag genccode or genmex
- The generated .c and .h files are wirtten 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.genccodegravload

Generate C-code for the vector of

gravitational load torques/forces

cGen.**genccodegravload**() 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 wirtten 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.genmexinertia

---

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.genccodefdyn

---

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
  - 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.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
  - 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.geninertia, CodeGenerator.genfkine

CodeGenerator.genfkine

Generate code for forward kinematics

T = cGen.**genfkine**() generates a symbolic homogeneous transform matrix (4×4) 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
  - 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.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
  - 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.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 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.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
  - 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.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
  - 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.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 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.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 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.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 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.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.genmexinertia

Generate C-MEX-function for robot inertia matrix

cGen.**genmexinertia**() 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 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.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 genmfun
- 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.genmfunfkine

Generate M-function for forward kinematics

cGen.**genmfunfkine**() 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 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.gengravload

CodeGenerator.genmfungravload

Generate M-functions for gravitational load

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

Notes

- Is called by CodeGenerator.gengravload 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.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 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)

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)

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 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

ctraj

Cartesian trajectory between two poses

tc = **ctraj**(**T0**, **T1**, **n**) is a Cartesian trajectory ($4 \times 4 \times \mathbf{n}$) from pose **T0** to **T1** with **n** points that follow a trapezoidal velocity profile along the path. The Cartesian trajectory is a homogeneous transform sequence and the last subscript being the point index, that is, T(:,:,i) is the i'th point along the path.

tc = **ctraj**(**T0**, **T1**, **s**) as above but the elements of **s** ($\mathbf{n} \times 1$) specify the fractional distance along the path, and these values are in the range [0 1]. The i'th point corresponds to a distance **s**(i) along the path.

Notes

- If **T0** or **T1** is equal to [] it is taken to be the identity matrix.
- In the second case **s** could be generated by a scalar trajectory generator such as TPOLY or LSPB (default).
- Orientation interpolation is performed using quaternion interpolation.

Reference

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

See also

lspb, mstraj, trinterp, UnitQuaternion.interp, SE3.ctraj

delta2tr

Convert differential motion to a homogeneous transform

T = **delta2tr**(**d**) is a homogeneous transform (4×4) representing differential translation and rotation. The vector **d**=(dx, dy, dz, dRx, dRy, dRz) represents an infinitessimal motion, and is an approximation to the spatial velocity multiplied by time.

See also

tr2delta, SE3.delta

DHFactor

Simplify symbolic link transform expressions

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

For example:

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

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

Methods

base	the base transform as a Java string
tool	the tool transform as a Java string
command	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 reported.

Reference

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

See also

SerialLink

---

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

---

distancexform

Distance transform

d = distancexform(**im**, **options**) is the distance transform of the binary image **im**. The elements of **d** have a value equal to the shortest distance from that element to a non-zero pixel in the input image **im**.

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

Options:

'euclidean'	Use Euclidean (L2) distance metric (default)
'cityblock'	Use cityblock or Manhattan (L1) distance metric

'animate'	Show the iterations of the computation
'delay', **d**	Delay of **d** seconds between animation frames (default 0.2s)
'movie', M	Save animation to a movie file or folder

'noipt'	Don't use Image Processing Toolbox, even if available
'novlfeat'	Don't use VLFeat, even if available
'nofast'	Don't use IPT, VLFeat or imorph, even if available.

'delay'

Notes

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

See also

imorph, DXform, animate

---

distributeblocks

Distribute blocks in Simulink block library

distributeblocks(**model**) equidistantly distributes blocks in a Simulink block library named **model**.

Notes

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

Author

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

See also

symexpr2slblock, doesblockexist

---

doesblockexist

Check existence of block in Simulink model

res = **doesblockexist**(**mdlname**, **blockaddress**) is a logical result that indicates whether or not the block **blockaddress** exists within the Simulink model **mdlname**.

Author

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

See also

symexpr2slblock, distributeblocks

Dstar

D* navigation class

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

Methods

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

Properties (read only)

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

Example

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

Notes

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

References

- The D* algorithm for real-time planning of optimal traverses, A. Stentz, Tech. Rep. CMU-RI-TR-94-37, The Robotics Institute, Carnegie-Mellon University, 1994. https://www.ri.cmu.edu/pub_files/pub3/stentz_anthony__tony__1994_2/stentz_anthony__tony__19
- Robotics, Vision & Control, Sec 5.2.2, Peter Corke, Springer, 2011.

See also

Navigation, DXform, PRM

Dstar.Dstar

D* constructor

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

Options

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

Other options are supported by the Navigation superclass.

See also

Navigation.Navigation

Dstar.char

Convert navigation object to string

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

See also

Dstar.display, Navigation.char

Dstar.modify_cost

Modify cost map

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

Notes

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

See also

Dstar.set_cost

Dstar.plan

Plan path to goal

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

DS.**plan**(**goal**,**options**) as above but goal given explicitly.

Options

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

Note

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

- The reset method causes a fresh **plan**, rather than replan.

See also

Dstar.reset

Dstar.plot

Visualize navigation environment

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

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

See also

Navigation.plot

Dstar.reset

Reset the planner

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

Dstar.set_cost

Set the current costmap

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

Notes

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

See also

Dstar.modify_cost

DXform

Distance transform navigation class

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

Methods

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

Properties (read only)

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

Example

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

Notes

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

References

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

See also

Navigation, Dstar, PRM, distancexform

DXform.DXform

Distance transform constructor

dx = **DXform**(**map**, **options**) is a distance transform navigation object, and **map** is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied).

Options

'goal', G	Specify the goal point (2×1)
'metric', M	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**(**goal**, **options**) plans a path to the goal given to the constructor, updates the internal distancemap where the value of each element is the minimum distance from the corresponding point to the goal.

DX.**plan**(**goal**, **options**) as above but goal is specified explicitly

Options

'animate'	Plot the distance transform as it evolves

Notes

- This may take many seconds.

See also

Navigation.path

DXform.plot

Visualize navigation environment

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

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

Notes

- See Navigation.**plot** for options.

See also

Navigation.plot

DXform.plot3d

3D costmap view

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

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

DX.**plot3d**(**p**, **ls**) as above but plot the line with the MATLAB linestyle **ls**.

See also

Navigation.plot

---

edgelist

Return list of edge pixels for region

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

eg = **edgelist**(**im**, **seed**, **direction**) as above, but the direction of edge following is specified. **direction** == 0 (default) means clockwise, non zero is counter-clockwise. Note that direction is with respect to y-axis upward, in matrix coordinate frame, not image frame.

[**eg**,**d**] = **edgelist**(**im**, **seed**, **direction**) as above but also returns a vector of edge segment directions which have values 1 to 8 representing W SW S SE E NW N NW respectively.

Notes

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

Reference

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

See also

ilabel

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

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

Methods

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

Properties

x_est	estimated state
P	estimated covariance
V_est	estimated odometry covariance
W_est	estimated sensor covariance
landmarks	maps sensor landmark id to filter state element
robot	reference to the Vehicle object
sensor	reference to the Sensor subclass object
history	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

```
ekf.plot_ellipse('g');
```

We can plot the covariance against time as

```
clf
ekf.plot_P();
```

Map-based vehicle localization

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

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

We run the simulation for 1000 time steps

```
ekf.run(1000);
```

then plot the map and the true vehicle path

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

and overlay the estimatd path

```
ekf.plot_xy('r');
```

and overlay uncertainty ellipses

```
ekf.plot_ellipse('g');
```

We can plot the covariance against time as

```
clf
ekf.plot_P();
```

Vehicle-based map making

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

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

We run the simulation for 1000 time steps

```
ekf.run(1000);
```

Then plot the true map

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

and overlay the estimated map with 97% confidence ellipses

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

Simultaneous localization and mapping (SLAM)

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

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

We run the simulation for 1000 time steps

```
ekf.run(1000);
```

then plot the map and the true vehicle path

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

and overlay the estimated path

```
ekf.plot_xy('r');
```

and overlay uncertainty ellipses

```
ekf.plot_ellipse('g');
```

We can plot the covariance against time as

```
clf
ekf.plot_P();
```

Then plot the true map

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

and overlay the estimated map with 3 sigma ellipses

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

References

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

Stochastic processes and filtering theory, AH Jazwinski Academic Press 1970

Acknowledgement

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

See also

Vehicle, RandomPath, RangeBearingSensor, pointmap, ParticleFilter

---

EKF.EKF

EKF object constructor

E = EKF(**vehicle**, **v_est**, **p0**, **options**) is an EKF that estimates the state of the **vehicle** (subclass of Vehicle) with estimated odometry covariance **v_est** (2×2) and initial covariance (3×3).

E = EKF(**vehicle**, **v_est**, **p0**, **sensor**, **w_est**, **map**, **options**) as above but uses information from a **vehicle** mounted sensor, estimated sensor covariance **w_est** and a **map** (LandmarkMap class).

Options

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

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

Notes

- If **map** is [] then it will be estimated.
- If **v_est** and **p0** are [] the vehicle is assumed error free and the filter will only estimate the landmark positions (map).
- If **v_est** and **p0** are finite the filter will estimate the vehicle pose and the landmark positions (map).
- EKF subclasses Handle, so it is a reference object.
- Dimensions of workspace are normally taken from the map if given.

See also

Vehicle, Bicycle, Unicycle, Sensor, RangeBearingSensor, LandmarkMap

---

EKF.char

Convert to string

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

See also

EKF.display

EKF.display

Display status of EKF object

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

Notes

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

See also

EKF.char

EKF.get_map

Get landmarks

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

See also

EKF.plot_map, EKF.plot_ellipse

EKF.get_P

Get covariance magnitude

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

EKF.get_xy

Get vehicle position

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

See also

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

EKF.init

Reset the filter

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

EKF.plot_ellipse

Plot vehicle covariance as an ellipse

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

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

Options

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

See also

plot_ellipse

EKF.plot_error

Plot vehicle position

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

Options

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

Notes

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

See also

EKF.plot_xy, EKF.plot_ellipse, EKF.plot_P

EKF.plot_map

Plot landmarks

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

E.**plot_map**(**ls**, **options**) as above but pass line style arguments **ls** to plot_ellipse.

Options

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

See also

EKF.get_map, EKF.plot_ellipse

EKF.plot_P

Plot covariance magnitude

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

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

EKF.plot_xy

Plot vehicle position

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

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

See also

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

EKF.run

Run the filter

E.**run**(**n**, **options**) runs the filter for **n** time steps and shows an animation of the vehicle moving.

Options

'plot'	Plot an animation of the vehicle moving

Notes

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

ETS2

Elementary transform sequence in 2D

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

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

Operation methods

fkine	forward kinematics

Information methods

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

structure a string listing the joint types

Display methods

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

Conversion methods

char	convert to string
string	convert to string with symbolic variables

Operators

*	compound two elementary transforms
+	compound two elementary transforms

Notes

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

See also

ETS3

ETS2.ETS2

Create an ETS2 object

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

E = **ETS2**(**e1**) is a new **ETS2** object that is a clone of the **ETS2** object **e1**.

See also

ETS2.Rz, ETS2.Tx, ETS2.Ty

ETS2.char

Convert to string

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

See also

ETS2.display

ETS2.display

Display parameters

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

Notes

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

See also

ETS2.char

ETS2.find

Find joints in transform sequence

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

ETS2.fkine

Forward kinematics

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

ETS.**fkine**(**q**, **n**, **options**) as above but process only the first **n** elements of the transform sequence.

Options

'deg'	Angles are given in degrees.

ETS2.isjoint

Test if transform is a joint

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

ETS2.isprismatic

Test if transform is prismatic joint

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

ETS2.mtimes

Compound transforms

E1 * E2 is a sequence of two elementary transform.

See also

ETS2.plus

ETS2.n

Number of joints in transform sequence

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

Notes

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

See also

ETS2.n

ETS2.njoints

Number of joints in transform sequence

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

See also

ETS2.n

ETS2.plot

Graphical display and animation

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

Options

'workspace', W	Size of robot 3D workspace, W = [xmn, xmx ymn ymx zmn zmx]
'floorlevel', L	Z-coordinate of floor (default -1)
'delay', D	Delay betwen frames for animation (s)
'fps', fps	Number of frames per second for display, inverse of 'delay' option
'[no]loop'	Loop over the trajectory forever
'[no]raise'	Autoraise the figure
'movie', M	Save an animation to the movie M
'trail', L	Draw a line recording the tip path, with line style L
'scale', S	Annotation scale factor
'zoom', Z	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.
'top'	View from the top.
'[no]shading'	Enable Gouraud shading (default true)
'lightpos', L	Position of the light source (default [0 0 20])
'[no]name'	Display the robot's name
'[no]wrist'	Enable display of wrist coordinate frame
'xyz'	Wrist axis label is XYZ
'noa'	Wrist axis label is NOA
'[no]arrow'	Display wrist frame with 3D arrows
'[no]tiles'	Enable tiled floor (default true)
'tilesize', S	Side length of square tiles on the floor (default 0.2)
'tile1color', C	Color of even tiles [r g b] (default [0.5 1 0.5] light green)
'tile2color', C	Color of odd tiles [r g b] (default [1 1 1] white)
'[no]shadow'	Enable display of shadow (default true)
'shadowcolor', C	Colorspec of shadow, [r g b]
'shadowwidth', W	Width of shadow line (default 6)
'[no]jaxes'	Enable display of joint axes (default false)
'[no]jvec'	Enable display of joint axis vectors (default false)
'[no]joints'	Enable display of joints
'jointcolor', C	Colorspec for joint cylinders (default [0.7 0 0])
'jointcolor', C	Colorspec for joint cylinders (default [0.7 0 0])
'jointdiam', D	Diameter of joint cylinder in scale units (default 5)
'linkcolor', C	Colorspec of links (default 'b')
'[no]base'	Enable display of base 'pedestal'
'basecolor', C	Color of base (default 'k')
'basewidth', W	Width of base (default 3)

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

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

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

Graphical annotations and options

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

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

which are controlled by **options**.

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

Figure behaviour

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

Notes

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

See also

ETS2.teach, SerialLink.plot3d

ETS2.plus

Compound transforms

E1 + E2 is a sequence of two elementary transform.

See also

ETS2.mtimes

ETS2.string

Convert to string with symbolic variables

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

See also

trchain

ETS2.structure

Show joint type structure

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

Notes

- The string will be E.njoints long.

See also

SerialLink.config

ETS2.teach

Graphical teach pendant

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

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

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

Options

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

GUI

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

Notes

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

See also

ETS2.plot

ETS3

Elementary transform sequence in 3D

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

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

Operation methods

fkine

Information methods

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

structure a string listing the joint types

Display methods

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

Conversion methods

char	convert to string
string	convert to string with symbolic variables

Operators

*	compound two elementary transforms
+	compound two elementary transforms

Notes

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

See also

ETS2

ETS3.ETS3

Create an ETS3 object

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

E = **ETS3**(**e1**) is a new **ETS3** object that is a clone of the **ETS3** object **e1**.

See also

ETS2.Rz, ETS2.Tx, ETS2.Ty

ETS3.char

Convert to string

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

See also

ETS3.display

ETS3.display

Display parameters

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

Notes

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

See also

ETS3.char

ETS3.find

Find joints in transform sequence

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

ETS3.fkine

Forward kinematics

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

ETS.**fkine**(**q**, **n**, **options**) as above but process only the first **n** elements of the transform sequence.

Options

'deg'	Angles are given in degrees.

ETS3.isjoint

Test if transform is a joint

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

ETS3.isprismatic

Test if transform is prismatic joint

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

ETS3.mtimes

Compound transforms

E1 * E2 is a sequence of two elementary transform.

See also

ETS3.plus

ETS3.n

Number of joints in transform sequence

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

Notes

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

See also

ETS3.n

ETS3.njoints

Number of joints in transform sequence

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

See also

ETS2.n

ETS3.plot

Graphical display and animation

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

Options

'workspace', W	Size of robot 3D workspace, W = [xmn, xmx ymn ymx zmn zmx]
'floorlevel', L	Z-coordinate of floor (default -1)
'delay', D	Delay betwen frames for animation (s)
'fps', fps	Number of frames per second for display, inverse of 'delay' option
'[no]loop'	Loop over the trajectory forever
'[no]raise'	Autoraise the figure
'movie', M	Save an animation to the movie M
'trail', L	Draw a line recording the tip path, with line style L
'scale', S	Annotation scale factor
'zoom', Z	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.
'top'	View from the top.
'[no]shading'	Enable Gouraud shading (default true)
'lightpos', L	Position of the light source (default [0 0 20])
'[no]name'	Display the robot's name
'[no]wrist'	Enable display of wrist coordinate frame
'xyz'	Wrist axis label is XYZ
'noa'	Wrist axis label is NOA
'[no]arrow'	Display wrist frame with 3D arrows
'[no]tiles'	Enable tiled floor (default true)
'tilesize', S	Side length of square tiles on the floor (default 0.2)
'tile1color', C	Color of even tiles [r g b] (default [0.5 1 0.5] light green)
'tile2color', C	Color of odd tiles [r g b] (default [1 1 1] white)
'[no]shadow'	Enable display of shadow (default true)
'shadowcolor', C	Colorspec of shadow, [r g b]
'shadowwidth', W	Width of shadow line (default 6)
'[no]jaxes'	Enable display of joint axes (default false)
'[no]jvec'	Enable display of joint axis vectors (default false)
'[no]joints'	Enable display of joints
'jointcolor', C	Colorspec for joint cylinders (default [0.7 0 0])
'jointcolor', C	Colorspec for joint cylinders (default [0.7 0 0])
'jointdiam', D	Diameter of joint cylinder in scale units (default 5)
'linkcolor', C	Colorspec of links (default 'b')
'[no]base'	Enable display of base 'pedestal'
'basecolor', C	Color of base (default 'k')
'basewidth', W	Width of base (default 3)

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

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

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

Graphical annotations and options

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

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

which are controlled by **options**.

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

Figure behaviour

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

Notes

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

See also

ETS3.teach, SerialLink.plot3d

ETS3.plus

Compound transforms

E1 + E2 is a sequence of two elementary transform.

See also

ETS3.mtimes

ETS3.string

Convert to string with symbolic variables

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

See also

trchain

ETS3.structure

Show joint type structure

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

Notes

- The string will be E.njoints long.

See also

SerialLink.config

ETS3.teach

Graphical teach pendant

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

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

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

Options

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

GUI

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

Notes

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

See also

ETS3.plot

---

jsingu

Show the linearly dependent joints in a Jacobian matrix

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

See also

SerialLink.jacobn

---

jtraj

Compute a joint space trajectory

[**q**,**qd**,**qdd**] = **jtraj**(**q0**, **qf**, **m**) is a joint space trajectory **q** (**m** $\times N$) where the joint coordinates vary from **q0** ($1 \times N$) to **qf** ($1 \times N$). A quintic (5th order) polynomial is used with default zero boundary conditions for velocity and acceleration. Time is assumed to vary from 0 to 1 in **m** steps. Joint velocity and acceleration can be optionally returned as **qd** (**m** $\times N$) and **qdd** (**m** $\times N$) respectively. The trajectory **q**, **qd** and **qdd** are **m** $\times N$ matrices, with one row per time step, and one column per joint.

[**q**,**qd**,**qdd**] = **jtraj**(**q0**, **qf**, **m**, **qd0**, **qdf**) as above but also specifies initial **qd0** ($1 \times N$) and final **qdf** ($1 \times N$) joint velocity for the trajectory.

[**q**,**qd**,**qdd**] = **jtraj**(**q0**, **qf**, **T**) as above but the number of steps in the trajectory is defined by the length of the time vector **T** (**m** $\times 1$).

[**q**,**qd**,**qdd**] = **jtraj**(**q0**, **qf**, **T**, **qd0**, **qdf**) as above but specifies initial and final joint velocity for the trajectory and a time vector.

Notes

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

See also

qplot, ctraj, SerialLink.jtraj

LandmarkMap

Map of planar point landmarks

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

Methods

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

Properties

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

Examples

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

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

which can be displayed by

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

Reference

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

See also

RangeBearingSensor, EKF

LandmarkMap.LandmarkMap

Create a map of point landmark landmarks

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

Options

'verbose'	Be verbose

LandmarkMap.char

Convert map parameters to a string

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

LandmarkMap.display

Display map parameters

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

Notes

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

See also

map.char

LandmarkMap.landmark

Get landmarks from map

f = M.**landmark**(**k**) is the coordinate (2×1) of the **k**'th **landmark** (**landmark**).

LandmarkMap.plot

Plot the map

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

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

Notes

- The **plot** is left with HOLD ON.

LandmarkMap.show

Show the landmark map

Notes

- Deprecated, use **plot** method.

LandmarkMap.verbosity

Set verbosity

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

Lattice

Lattice planner navigation class

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

Methods

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

Properties (read only)

graph	A PGraph object describign the tree

Example

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

References

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

See also

Navigation, DXform, Dstar, pgraph

Lattice.Lattice

Create a Lattice navigation object

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

Options

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

Other **options** are supported by the Navigation superclass.

Notes

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

See also

Navigation.Navigation

Lattice.char

Convert to string

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

See also

Lattice.display

Lattice.plan

Create a lattice plan

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

Options

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

Default parameter values come from the constructor

Lattice.plot

Visualize navigation environment

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

Options

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

Lattice.query

Find a path between two poses

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

Link

manipulator Link class

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

Constructors

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

Information/display methods

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

Conversion methods

char	convert to string

Operation methods

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

Testing methods

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

Overloaded operators

+	concatenate links, result is a SerialLink object

Properties (read/write)

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

Examples

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

Notes

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

References

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

See also

Link, Revolute, Prismatic, SerialLink, RevoluteMDH, PrismaticMDH

Link.Link

Create robot link object

This the class constructor which has several call signatures.

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

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

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

Options

'theta', TH	joint angle, if not specified joint is revolute
'd', D	joint extension, if not specified joint is prismatic
'a', A	joint offset (default 0)
'alpha', A	joint twist (default 0)
'standard'	defined using standard D&H parameters (default).
'modified'	defined using modified D&H parameters.
'offset', O	joint variable offset (default 0)
'qlim', **L**	joint limit (default [])
'I', I	link inertia matrix (3×1, 6×1 or 3×3)
'r', R	link centre of gravity (3×1)
'm', M	link mass (1×1)
'G', G	motor gear ratio (default 1)
'B', B	joint friction, motor referenced (default 0)
'Jm', J	motor inertia, motor referenced (default 0)
'Tc', T	Coulomb friction, motor referenced (1×1 or 2×1), (default [0 0])
'revolute'	for a revolute joint (default)
'prismatic'	for a prismatic joint 'p'
'standard'	for standard D&H parameters (default).
'modified'	for modified D&H parameters.
'sym'	consider all parameter values as symbolic not numeric

Notes

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

Old syntax

L = Link(**dh**, **options**) is a link object using the specified kinematic convention and with parameters:

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

Options

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

Notes

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

Examples

A standard Denavit-Hartenberg link

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

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

Using the old syntax

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

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

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

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

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

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

For a modified Denavit-Hartenberg revolute joint

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

Notes

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

See also

Revolute, Prismatic, RevoluteMDH, PrismaticMDH

Link.A

Link transform matrix

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

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

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

Notes

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

See also

SerialLink.fkine

Link.char

Convert to string

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

See also

Link.display

Link.display

Display parameters

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

Notes

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

See also

Link.char, Link.dyn, SerialLink.showlink

Link.dyn

Show inertial properties of link

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

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

See also

SerialLink.dyn

Link.friction

Joint friction force

f = L.**friction**(**qd**) is the joint **friction** force/torque ($1 \times N$) for joint velocity **qd** ($1 \times N$). The **friction** model includes:

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

Notes

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

See also

Link.nofriction

Link.horzcat

Concatenate link objects

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

Notes

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

See also

Link.plus

Link.islimit

Test joint limits

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

Note

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

Link.isprismatic

Test if joint is prismatic

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

See also

Link.isrevolute

Link.isrevolute

Test if joint is revolute

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

See also

Link.isprismatic

Link.issym

Check if link is a symbolic model

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

See also

Link.sym

Link.nofriction

Remove friction

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

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

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

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

Notes

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

See also

Link.friction, SerialLink.nofriction, SerialLink.fdyn

Link.plus

Concatenate link objects into a robot

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

Notes

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

See also

SerialLink, SerialLink.plus, Link.horzcat

Link.set.I

Set link inertia

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

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

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

Link.set.r

Set centre of gravity

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

Link.set.Tc

Set Coulomb friction

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

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

Notes

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

See also

Link.friction

Link.sym

Convert link parameters to symbolic type

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

See also

Link.issym

Link.type

Joint type

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

See also

SerialLink.config

---

lspb

Linear segment with parabolic blend

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

[**s**,**sd**,**sdd**] = **lspb**(**s0**, **sf**, **m**, **v**) as above but specifies the velocity of the linear segment which is normally computed automatically.

[**s**,**sd**,**sdd**] = **lspb**(**s0**, **sf**, **T**) as above but specifies the trajectory in terms of the length of the time vector **T** (**m** $\times$ 1).

[**s**,**sd**,**sdd**] = **lspb**(**s0**, **sf**, **T**, **v**) as above but specifies the velocity of the linear segment which is normally computed automatically and a time vector.

lspb(**s0**, **sf**, **m**, **v**) as above but plots **s**, **sd** and **sdd** versus time in a single figure.

Notes

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

References

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

See also

tpoly, jtraj

makemap

Make an occupancy map

map = **makemap**(**n**) is an occupancy grid **map** (**n** $\times$ **n**) created by a simple interactive editor. The **map** is initially unoccupied and obstacles can be added using geometric primitives.

map = **makemap**() as above but **n**=128.

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

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

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

See also

dxform, PRM, RRT

mdl_ball

Create model of a ball manipulator

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

mdl_ball(**n**) as above but creates a manipulator with **n** joints.

Also define the workspace vectors:

q joint angle vector for default ball configuration

Reference

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

Notes

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

See also

mdl_coil, SerialLink

mdl_baxter

Kinematic model of Baxter dual-arm robot

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

Also define the workspace vectors:

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

Notes

- SI units of metres are used.

References

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

See also

mdl_nao, SerialLink

mdl_cobra600

Create model of Adept Cobra 600 manipulator

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

Also define the workspace vectors:

qz	zero joint angle configuration

Notes

- SI units are used.

See also

serialrevolute, mdl_puma560akb, mdl_stanford

mdl_coil

Create model of a coil manipulator

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

mdl_ball(**n**) as above but creates a manipulator with **n** joints.

Also defines the workspace vectors:

q joint angle vector for default helical configuration

Reference

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

Notes

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

See also

mdl_ball, SerialLink

mdl_fanuc10L

Create kinematic model of Fanuc AM120iB/10L robot

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

Also defines the workspace vector:

q0	mastering position.

Notes

- SI units of metres are used.

Author

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

See also

mdl_irb140, mdl_m16, mdl_motomanHP6, mdl_puma560, SerialLink

mdl_hyper2d

Create model of a hyper redundant planar manipulator

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

mdl_hyper2d(**n**) as above but creates a manipulator with **n** joints.

Also define the workspace vectors:

qz joint angle vector for zero angle configuration

R = **mdl_hyper2d**(**n**) functional form of the above, returns the SerialLink object.

[**R**,**qz**] = **mdl_hyper2d**(**n**) as above but also returns a vector of zero joint angles.

Notes

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

See also

mdl_hyper3d, mdl_coil, mdl_ball, mdl_twolink, SerialLink

---

mdl_hyper3d

Create model of a hyper redundant 3D manipulator

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

mdl_hyper3d(**n**) as above but creates a manipulator with **n** joints.

Also define the workspace vectors:

qz joint angle vector for zero angle configuration

R = **mdl_hyper3d**(**n**) functional form of the above, returns the SerialLink object.

[**R**,**qz**] = **mdl_hyper3d**(**n**) as above but also returns a vector of zero joint angles.

Notes

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

See also

mdl_hyper2d, mdl_ball, mdl_coil, SerialLink

mdl_irb140

Create model of ABB IRB 140 manipulator

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

Also define the workspace vectors:

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

Reference

- “IRB 140 data sheet”, ABB Robotics.
- "Utilizing the Functional Work Space Evaluation Tool for Assessing a System Design and Reconfiguration Alternatives" A. Djuric and R. J. Urbanic

Notes

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

See also

mdl_fanuc10l, mdl_m16, mdl_motormanhp6, mdl_S4ABB2p8, mdl_puma560, SerialLink

mdl_irb140_mdh

Create model of the ABB IRB 140 manipulator

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

Also define the workspace vectors:

qz	zero joint angle configuration

Reference

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

See also

mdl_irb140, mdl_puma560, mdl_stanford, mdl_twolink, SerialLink

Notes

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

mdl_jaco

Create model of Kinova Jaco manipulator

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

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY' configuration

Reference

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

Notes

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

See also

mdl_mico, mdl_puma560, SerialLink

mdl_KR5

Create model of Kuka KR5 manipulator

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

Also define the workspace vectors:

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

Notes

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

Author

- Gautam Sinha, Indian Institute of Technology, Kanpur.

See also

mdl_irb140, mdl_fanuc10l, mdl_motomanHP6, mdl_S4ABB2p8, mdl_puma560, SerialLink

---

mdl_LWR

Create model of Kuka LWR manipulator

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

Also define the workspace vectors:

qz	all zero angles

Notes

- SI units of metres are used.

Reference

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

See also

mdl_kr5, mdl_irb140, mdl_puma560, SerialLink

---

mdl_M16

Create model of Fanuc M16 manipulator

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

Also define the workspace vectors:

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

References

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

Notes

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

See also

mdl_irb140, mdl_fanuc10l, mdl_motomanHP6, mdl_S4ABB2p8, mdl_puma560, SerialLink

mdl_mico

Create model of Kinova Mico manipulator

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

Also define the workspace vectors:

qz	zero joint angle configuration
qr	vertical 'READY' configuration

Reference

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

Notes

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

See also

Revolute, mdl_jaco, mdl_puma560, mdl_twolink, SerialLink

---

mdl_motomanHP6

Create kinematic data of a Motoman HP6 manipulator

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

Also defines the workspace vector:

q0	mastering position.

Author

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

Notes

- SI units of metres are used.

See also

mdl_irb140, mdl_m16, mdl_fanuc10l, mdl_S4ABB2p8, mdl_puma560, SerialLink

---

mdl_nao

Create model of Aldebaran NAO humanoid robot

MDL_NAO is a script that creates several workspace variables

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

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

Reference

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

Notes

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

See also

mdl_baxter, SerialLink

mdl_offset6

A minimalistic 6DOF robot arm with shoulder offset

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

Also define the workspace vectors:

qz	zero joint angle configuration

Notes

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

See also

mdl_simple6, mdl_puma560, mdl_twolink, SerialLink

mdl_onelink

Create model of a simple 1-link mechanism

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

Also defines the vector:

qz	corresponds to the zero joint angle configuration.

Notes

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

References

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

See also

mdl_twolink, mdl_planar1, SerialLink

---

mdl_p8

Create model of Puma robot on an XY base

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

Also define the workspace vectors:

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

Notes

- SI units of metres are used.

References

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

See also

mdl_puma560, SerialLink

---

mdl_panda

Create model of Franka-Emika PANDA robot

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

Also define the workspace vectors:

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

Reference

- http://www.diag.uniroma1.it/ deluca/rob1_en/WrittenExamsRob1/Robotics1_18.01.11.pdf

Notes

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

See also

mdl_sawyer, SerialLink

mdl_phantomx

Create model of PhantomX pincher manipulator

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

Also define the workspace vectors:

qz	zero joint angle configuration

Notes

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

Reference

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

mdl_planar1

Create model of a simple planar 1-link mechanism

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

Also defines the vector:

qz	corresponds to the zero joint angle configuration.

Notes

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

See also

mdl_planar2, mdl_planar3, SerialLink

mdl_planar2

Create model of a simple planar 2-link mechanism

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

Also defines the vector:

qz	corresponds to the zero joint angle configuration.

Notes

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

See also

mdl_twolink, mdl_planar1, mdl_planar3, SerialLink

---

mdl_planar2_sym

Create model of a simple planar 2-link mechanism

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

Also defines the vector:

qz	corresponds to the zero joint angle configuration.

Also defines the vector:

qz	corresponds to the zero joint angle configuration.

Notes

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

See also

mdl_twolink, mdl_planar1, mdl_planar3, SerialLink

---

mdl_planar3

Create model of a simple planar 3-link mechanism

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

Also defines the vector:

qz	corresponds to the zero joint angle configuration.

Notes

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

See also

mdl_twolink, mdl_planar1, mdl_planar2, SerialLink

mdl_puma560

Create model of Puma 560 manipulator

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

Also define the workspace vectors:

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

Notes

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

Reference

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

See also

serialrevolute, mdl_puma560akb, mdl_stanford

mdl_puma560akb

Create model of Puma 560 manipulator

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

Also defines the workspace vectors:

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

Notes

- SI units are used.

References

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

See also

mdl_puma560, mdl_stanford_mdh, SerialLink

mdl_quadrotor

Dynamic parameters for a quadrotor.

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

Properties

This is a structure with the following elements:

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

Notes

- SI units are used.

References

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

- This is a heavy lift quadrotor

See also

sl_quadrotor

---

mdl_S4ABB2p8

Create kinematic model of ABB S4 2.8robot

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

Also defines the workspace vector:

q0	mastering position.

Author

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

See also

mdl_fanuc10l, mdl_m16, mdl_motormanhp6, mdl_irb140, mdl_puma560, SerialLink

---

mdl_sawyer

Create model of Rethink Robotics Sawyer robot

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

Also define the workspace vectors:

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

Reference

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

Notes

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

See also

mdl_baxter, SerialLink

---

mdl_simple6

A minimalistic 6DOF robot arm

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

Also define the workspace vectors:

qz	zero joint angle configuration

Notes

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

See also

mdl_offset6, mdl_puma560, SerialLink

---

mdl_stanford

Create model of Stanford arm

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

Also defines the vectors:

qz	zero joint angle configuration.

Note

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

References

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

See also

mdl_puma560, mdl_puma560akb, SerialLink

---

mdl_stanford_mdh

Create model of Stanford arm using MDH conventions

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

Also defines the vectors:

qz	zero joint angle configuration.

Notes

- SI units are used.

References

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

See also

mdl_puma560, mdl_puma560akb, SerialLink

mdl_twolink

Create model of a 2-link mechanism

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

Also defines the vector:

qz	corresponds to the zero joint angle configuration.

Notes

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

References

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

See also

mdl_twolink_sym, mdl_planar2, SerialLink

mdl_twolink_mdh

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

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

Also defines the vector:

qz	corresponds to the zero joint angle configuration.

Notes

- SI units of metres are used.
- It is a planar mechanism operating in the xz-plane (vertical) and is therefore not affected by gravity.

References

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

See also

mdl_twolink, mdl_onelink, mdl_planar2, SerialLink

mdl_twolink_sym

Create symbolic model of a simple 2-link mechanism

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

- link lengths: a1, a2
- link masses: m1, m2
- link CoMs in the link frame x-direction: c1, c2
- gravitational acceleration: g
- joint angles: q1, q2
- joint angle velocities: qd1, qd2
- joint angle accelerations: qdd1, qdd2

Notes

- It is a planar mechanism operating in the vertical plane and is therefore affected by gravity (unlike mdl_planar2 in the horizontal plane).
- Gear ratio is 1 and motor inertia is 0.
- Link inertias Iyy1, Iyy2 are 0.
- Viscous and Coulomb friction is 0.

References

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

See also

mdl_puma560, mdl_stanford, SerialLink

mdl_ur10

Create model of Universal Robotics UR10 manipulator

MDL_UR5 is a script that creates the workspace variable ur10 which describes the kinematic characteristics of a Universal Robotics UR10 manipulator using standard DH conventions.

Also define the workspace vectors:

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

Reference

- https://www.universal-robots.com/how-tos-and-faqs/faq/ur-faq/actual-center-of-mass-for-robot-17264/

Notes

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

See also

mdl_ur3, mdl_ur5, mdl_puma560, SerialLink

mdl_ur3

Create model of Universal Robotics UR3 manipulator

MDL_UR5 is a script that creates the workspace variable ur3 which describes the kinematic characteristics of a Universal Robotics UR3 manipulator using standard DH conventions.

Also define the workspace vectors:

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

Reference

- https://www.universal-robots.com/how-tos-and-faqs/faq/ur-faq/actual-center-of-mass-for-robot-17264/

Notes

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

See also

mdl_ur5, mdl_ur10, mdl_puma560, SerialLink

---

mdl_ur5

Create model of Universal Robotics UR5 manipulator

MDL_UR5 is a script that creates the workspace variable ur5 which describes the kinematic characteristics of a Universal Robotics UR5 manipulator using standard DH conventions.

Also define the workspace vectors:

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

Reference

- https://www.universal-robots.com/how-tos-and-faqs/faq/ur-faq/actual-center-of-mass-for-robot-17264/

Notes

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

See also

mdl_ur3, mdl_ur10, mdl_puma560, SerialLink

---

models

Summarise and search available robot models

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

models(**query**) lists those **models** that match the keyword **query**. Case is ignored in the comparison.

m = **models**(**query**) as above but returns a cell array ($N \times 1$) of the names of the **m**-files that define the **models**.

Examples

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

Notes

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

mplot

Plot time-series data

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

mplot(**y**, **options**) plots the time series data **y**($N \times M$) in multiple subplots. The first column is assumed to be time, so M-1 plots are produced.

mplot(**T**, **y**, **options**) plots the time series data **y**($N \times M$) in multiple subplots. Time is provided explicitly as the first argument so M plots are produced.

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

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

mplot(**R**, **options**) as above but **R** is a Simulink.SimulationOutput object returned by the Simulink sim() function.

Options

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

Notes

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

See also

plot2, plotp

mstraj

Multi-segment multi-axis trajectory

traj = mstraj(**wp**, **qdmax**, **tseg**, **q0**, **dt**, **tacc**, **options**) is a trajectory ($K \times N$) for N axes moving simultaneously through M segment. Each segment is linear motion and polynomial blends connect the segments. The axes start at **q0** ($1 \times N$) and pass through M-1 via points defined by the rows of the matrix **wp** ($M \times N$), and finish at the point defined by the last row of **wp**. The trajectory matrix has one row per time step, and one column per axis. The number of steps in the trajectory K is a function of the number of via points and the time or velocity limits that apply.

- **wp** ($M \times N$) is a matrix of via points, 1 row per via point, one column per axis. The last via point is the destination.
- **qdmax** ($1 \times N$) are axis speed limits which cannot be exceeded,
- **tseg** ($1 \times M$) are the durations for each of the K segments
- **q0** ($1 \times N$) are the initial axis coordinates
- **dt** is the time step
- **tacc** (1×1) is the acceleration time used for all segment transitions

- **tacc** $(1 \times M)$ is the acceleration time per segment, **tacc**(i) is the acceleration time for the transition from segment i to segment i+1. **tacc**(1) is also the acceleration time at the start of segment 1.

traj = **mstraj**(**wp**, **qdmax**, **tseg**, [], **dt**, **tacc**, **options**) as above but the initial coordinates are taken from the first row of **wp**.

traj = **mstraj**(**wp**, **qdmax**, **q0**, **dt**, **tacc**, **qd0**, **qdf**, **options**) as above but additionally specifies the initial and final axis velocities $(1 \times N)$.

Options

'verbose'	Show details.

Notes

- Only one of **qdmax** or **tseg** can be specified, the other is set to [].
- If no output arguments are specified the trajectory is plotted.
- The path length K is a function of the number of via points, **q0**, **dt** and **tacc**.
- The final via point P(end,:) is the destination.
- The motion has M segments from **q0** to P(1,:) to P(2,:) ... to P(end,:).
- All axes reach their via points at the same time.
- Can be used to create joint space trajectories where each axis is a joint coordinate.
- Can be used to create Cartesian trajectories where the "axes" correspond to translation and orientation in RPY or Euler angle form.
- If qdmax is a scalar then all axes are assumed to have the same maximum speed.

See also

mtraj, lspb, ctraj

mtraj

Multi-axis trajectory between two points

[**q**,**qd**,**qdd**] = **mtraj**(**tfunc**, **q0**, **qf**, **m**) is a multi-axis trajectory ($\mathbf{m} \times N$) 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** which is applied to each axis:

```
[S,SD,SDD] = TFUNC(S0, SF, M);
```

and possible values of **tfunc** include @lspb for a trapezoidal trajectory, or @tpoly for a polynomial trajectory.

[**q**,**qd**,**qdd**] = **mtraj**(**tfunc**, **q0**, **qf**, **T**) as above but **T** ($\mathbf{m} \times 1$) 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

multidfprintf

Print formatted text to multiple streams

COUNT = MULTIDFPRINTF(IDVEC, FORMAT, A, ...) performs formatted output to multiple streams such as console and files. FORMAT is the format string as used by sprintf and fprintf. A is the array of elements, to which the format will be applied similar to sprintf and fprint.

IDVEC is a vector ($1 \times N$) of file descriptors and COUNT is a vector ($1 \times N$) of the number of bytes written to each file.

Notes

- To write to the consolde use the file identifier 1.

Example

```
% Create and open a new example file:
fid = fopen('exampleFile.txt','w+');
% Write something to the file and the console simultaneously:
multidfprintf([1 FID],'% s % d % d % d% Close the file:
fclose(FID);
```

Authors

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

See also

fprintf, sprintf

Navigation

Navigation superclass

An abstract superclass for implementing planar grid-based navigation classes.

Methods

Navigation	Superclass constructor
plan	Find a path to goal
query	Return/animate a path from start to goal
plot	Display the occupancy grid
display	Display the parameters in human readable form
char	Convert to string
isoccupied	Test if cell is occupied
rand	Uniformly distributed random number
randn	Normally distributed random number
randi	Uniformly distributed random integer
progress_init	Create a progress bar
progress	Update progress bar
progress_delete	Remove progress bar

Properties (read only)

occgrid	Occupancy grid representing the navigation environment
goal	Goal coordinate
start	Start coordinate
seed0	Random number state

Methods that must be provided in subclass

plan	Generate a plan for motion to goal
next	Returns coordinate of next point along path

Methods that may be overriden in a subclass

goal_set	The goal has been changed by nav.goal = (a,b)
navigate_init	Start of path planning.

Notes

- Subclasses the MATLAB handle class which means that pass by reference semantics apply.
- A grid world is assumed and vehicle position is quantized to grid cells.
- Vehicle orientation is not considered.
- The initial random number state is captured as seed0 to allow rerunning an experiment with an interesting outcome.

See also

Bug2, Dstar, dxform, PRM, Lattice, RRT

Navigation.Navigation

Create a Navigation object

n = Navigation(**occgrid**, **options**) is a Navigation object that holds an occupancy grid **occgrid**. A number of options can be be passed.

Options

'goal', G	Specify the goal point (2 × 1)
'inflate', K	Inflate all obstacles by K cells.
'private'	Use private random number stream.
'reset'	Reset random number stream.
'verbose'	Display debugging information
'seed', S	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.
- Inflation requires either MVTB or IPT installed.
- The 'private' option creates a private random number stream for the methods rand, randn and randi. If not given the global stream is used.

See also

randstream

Navigation.char

Convert to string

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

Navigation.display

Display status of navigation object

N.**display**() displays the state of the navigation object in human-readable form.

Notes

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

See also

Navigation.char

Navigation.goal_change

Notify change of goal

Invoked when the goal property of the object is changed. Typically this is overriden in a subclass to take particular action such as invalidating a costmap.

Navigation.isoccupied

Test if grid cell is occupied

N.**isoccupied**(**pos**) is true if there is a valid grid map and the coordinates given by the columns of **pos** ($2 \times N$) are occupied.

N.**isoccupied**(**x**,**y**) as above but the coordinates given separately.

Notes:

- **x** and **y** are Cartesian rather than MATLAB row-column coordinates.

Navigation.message

Print debug message

N.**message**(**s**) displays the string **s** if the verbose property is true.

N.**message**(**fmt**, **args**) as above but accepts printf() like semantics.

Navigation.navigate_init

Notify start of path

N.**navigate_init**(**start**) is called when the query() method is invoked. Typically overriden in a subclass to take particular action such as computing some path parameters. **start** (2×1) is the initial position for this path, and nav.goal (2×1) is the final position.

See also

Navigate.query

Navigation.plot

Visualize navigation environment

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

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

Options

'distance', D	Display a distance field D behind the obstacle map. D is a matrix of the same size as the occupancy grid.
'colormap', @f	Specify a colormap for the distance field as a function handle, eg. @hsv
'beta', B	Brighten the distance field by factor B.
'inflated'	Show the inflated occupancy grid rather than original

Notes

- The distance field at a point encodes its distance from the goal, small distance is dark, a large distance is bright. Obstacles are encoded as red.
- Beta value -1<B<0 to darken, 0<B<+1 to lighten.

See also

Navigation.plot_fg, Navigation.plot_bg

Navigation.plot_bg

Visualization background

N.**plot_bg**(**options**) displays the occupancy grid with occupied cells shown as red and an optional distance field.

N.**plot_bg**(**p**,**options**) as above but overlays the points along the path ($2 \times M$) matrix.

Options

'distance', D	Display a distance field D behind the obstacle map. D is a matrix of the same size as the occupancy grid.
'colormap', @f	Specify a colormap for the distance field as a function handle, eg. @hsv
'beta', B	Brighten the distance field by factor B.
'inflated'	Show the inflated occupancy grid rather than original
'pathmarker', M	Options to draw a path point
'startmarker', M	Options to draw the start marker
'goalmarker', M	Options to draw the goal marker

Notes

- The distance field at a point encodes its distance from the goal, small distance is dark, a large distance is bright. Obstacles are encoded as red.
- Beta value -1<B<0 to darken, 0<B<+1 to lighten.

See also

Navigation.plot, Navigation.plot_fg, brighten

Navigation.plot_fg

Visualization foreground

N.**plot_fg**(**options**) displays the start and goal locations if specified. By default the goal is a pentagram and start is a circle.

N.**plot_fg**(**p**, **options**) as above but overlays the points along the path ($2 \times M$) matrix.

Options

'pathmarker', M	Options to draw a path point
'startmarker', M	Options to draw the start marker
'goalmarker', M	Options to draw the goal marker

Notes

- In all cases M is a single string eg. 'r*' or a cell array of MATLAB LineSpec options.
- Typically used after a call to plot_bg().

See also

Navigation.plot_bg

Navigation.query

Find a path from start to goal using plan

N.**query**(**start**, **options**) animates the robot moving from **start** (2×1) to the goal (which is a property of the object) using a previously computed plan.

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

The method performs the following steps:

- Initialize navigation, invoke method N.navigate_init()
- Visualize the environment, invoke method N.plot()
- Iterate on the next() method of the subclass until the goal is achieved.

Options

'animate'	Show the computed path as a series of green dots.

Notes

- If **start** given as [] then the user is prompted to click a point on the map.

See also

Navigation.navigate_init, Navigation.plot, Navigation.goal

Navigation.rand

Uniformly distributed random number

R = N.**rand**() return a uniformly distributed random number from a private random number stream.

R = N.**rand**(**m**) as above but return a matrix ($\mathbf{m} \times \mathbf{m}$) of random numbers.

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

i = N.**randn**(**rm**, **L**,**m**) as above but returns a matrix ($\mathbf{L} \times \mathbf{m}$) of random integers.

Notes

- Accepts the same arguments as randi().
- Seed is provided to Navigation constructor.

- Provides an independent sequence of random numbers that does not interfere with any other randomised algorithms that might be used.

See also

Navigation.rand, Navigation.randn, randi, randstream

Navigation.randn

Normally distributed random number

R = N.**randn**() returns a normally distributed random number from a private random number stream.

R = N.**randn**(**m**) as above but returns a matrix (**m** × **m**) of random numbers.

R = N.**randn**(**L**,**m**) as above but returns a matrix (**L** × **m**) of random numbers.

Notes

- Accepts the same arguments as **randn**().
- Seed is provided to Navigation constructor.
- Provides an independent sequence of random numbers that does not interfere with any other randomised algorithms that might be used.

See also

Navigation.rand, Navigation.randi, randn, randstream

Navigation.spinner

Update progress spinner

N.**spinner**() displays a simple ASCII progress **spinner**, a rotating bar.

Navigation.verbosity

Set verbosity

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

ParticleFilter

Particle filter class

Monte-carlo based localisation for estimating vehicle pose based on odometry and observations of known landmarks.

Methods

run	run the particle filter
plot_xy	display estimated vehicle path
plot_pdf	display particle distribution

Properties

robot	reference to the robot object
sensor	reference to the sensor object
history	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 = PointMap(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 is the covariance of the random noise added to the particle states at each iteration to represent uncertainty in configuration.

```
Q = diag([0.1, 0.1, 1*pi/180]).^2;
```

and the covariance of the likelihood function applied to innovation

```
L = diag([0.1 0.1]);
```

Now construct the particle filter

```
pf = ParticleFilter(veh, sensor, Q, L, 1000);
```

which is configured with 1000 particles. The particles are initially uniformly distributed over the 3-dimensional configuration space.

We run the simulation for 1000 time steps

```
pf.run(1000);
```

then plot the map and the true vehicle path

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

and overlay the mean of the particle cloud

```
pf.plot_xy('r');
```

We can plot the standard deviation against time

```
plot(pf.std(1:100,:))
```

The particles are a sampled approximation to the PDF and we can display this as

```
pf.plot_pdf()
```

Acknowledgement

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

Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

See also

Vehicle, RandomPath, RangeBearingSensor, pointmap, EKF

ParticleFilter.ParticleFilter

Particle filter constructor

pf = ParticleFilter(**vehicle**, **sensor**, **q**, **L**, **np**, **options**) is a particle filter that estimates the state of the **vehicle** with a landmark sensor **sensor**. **q** is the covariance of the noise added to the particles at each step (diffusion), **L** is the covariance used in the sensor likelihood model, and **np** is the number of particles.

Options

'verbose'	Be verbose.
'private'	Use private random number stream.
'reset'	Reset random number stream.
'seed', S	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, pointmap

ParticleFilter.char

Convert to string

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

See also

ParticleFilter.display

ParticleFilter.display

Display status of particle filter object

PF.**display**() displays the state of the **ParticleFilter** object in human-readable form.

Notes

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

See also

ParticleFilter.char

ParticleFilter.init

Initialize the particle filter

PF.**init**() initializes the particle distribution and clears the history.

Notes

- If initial particle states were given to the constructor the states are set to this value, else a random distribution over the map is used.
- Invoked by the run() method.

ParticleFilter.plot_pdf

Plot particles as a PDF

PF.**plot_pdf**() plots a sparse PDF as a series of vertical line segments of height equal to particle weight.

ParticleFilter.plot_xy

Plot vehicle position

PF.**plot_xy**() plots the estimated vehicle path in the xy-plane.

PF.**plot_xy**(**ls**) as above but the optional line style arguments **ls** are passed to plot.

ParticleFilter.run

Run the particle filter

PF.**run**(**n**, **options**) runs the filter for **n** time steps.

Options

'noplot'	Do not show animation.
'movie', M	Create an animation movie file M

Notes

- All previously estimated states and estimation history is cleared.

---

pickregion

Pick a rectangular region of a figure using mouse

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

Options

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

Notes

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

Author

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

---

plot_vehicle

Draw mobile robot pose

plot_vehicle(**x**,**options**) draws an oriented triangle to represent the pose of a mobile robot moving in a planar world. The pose **x** (1×3) = [x,y,theta]. If **x** is a matrix $(N \times 3)$ then an animation of the robot motion is shown and animated at the specified frame rate.

Image mode

Create a structure with the following elements and pass it with the 'model' option.

image	an RGB image $(H \times W \times 3)$
alpha	an alpha plane $(H \times W)$ with pixels 0 if transparent
rotation	image rotation in degrees required for front to pointing to the right
centre	image coordinate (U,V) of the centre of the back axle
length	length of the car in real-world units

Animation mode

H = **plot_vehicle**(**x**,**options**) as above draws the robot and returns a handle.

plot_vehicle(**x**, 'handle', **H**) updates the pose **x** (1×3) of the previously drawn robot.

Options

'scale', S	draw vehicle with length S x maximum axis dimension (default 1/60)
'size', S	draw vehicle with length S
'fillcolor', F	the color of the circle's interior, MATLAB color spec
'alpha', A	transparency of the filled circle: 0=transparent, 1=solid
'box'	draw a box shape (default is triangle)
'fps', F	animate at F frames per second (default 10)
'image', I	use an image to represent the robot pose
'retain'	when **x** ($N \times 3$) then retain the robots, leaving a trail
'model', M	animate an image of the vehicle. M is a structure with elements: image, alpha, rotation (deg), centre (pix), length (m).
'axis', h	handle of axis or UIAxis to draw into (default is current axis)
'movie', M	create a movie file in file M

Example

```
[car.image,~,car.alpha] = imread('car2.png');  % image and alpha layer
car.rotation = 180;  % image rotation to align front with world x-axis
car.centre = [648; 173]; % image coordinates of centre of the back wheels
car.length = 4.2; % real world length for scaling (guess)
h = plot_vehicle(x, 'model', car)  % draw car at configuration x
plot_vehicle(x, 'handle', h)  % animate car to configuration x
```

Notes

- The vehicle is drawn relative to the size of the axes, so set them first using axis().
- For backward compatibility, 'fill', is a synonym for 'fillcolor'
- For the 'model' option, you provide a monochrome or color image of the vehicle. Optionally you can provide an alpha mask (0=transparent). Specify the reference point on the vehicle as the (x,y) pixel coordinate within the image. Specify the rotation, in degrees, so that the car's front points to the right. Finally specify a length of the car, the image is scaled to be that length in the plot.
- Set 'fps' to Inf to have zero pause

See also Vehicle.plot, Animate, plot_poly, demos/car_animation

plotbotopt

Define default options for robot plotting

A user provided function that returns a cell array of default plot options for the SerialLink.plot method.

See also

SerialLink.plot

---

plotp

Plot trajectory

Convenience function to plot points stored columnwise.

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

plotp(**p**, **ls**) as above but the line style arguments **ls** are passed to plot.

See also

plot, plot2

---

polydiff

Differentiate a polynomial

pd = **polydiff**(**p**) is a vector of coefficients of a polynomial ($1 \times N$-1) which is the derivative of the polynomial **p** ($1 \times N$).

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

6      2
```

See also

polyval

Polygon

Polygon class

A general class for manipulating polygons and vectors of polygons.

Methods

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

Properties

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

Notes

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

Acknowledgement

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

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

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

Polygon.Polygon

Polygon class constructor

p = **Polygon**(**v**) is a polygon with vertices given by **v**, one column per vertex.

p = **Polygon**(**C**, **wh**) is a rectangle centred at **C** with dimensions **wh**=[WIDTH, HEIGHT].

Polygon.area

Area of polygon

a = P.**area**() is the **area** of the polygon.

See also

Polygon.moments

Polygon.centroid

Centroid of polygon

x = P.**centroid**() is the **centroid** of the polygon.

See also

Polygon.moments

Polygon.char

String representation

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

Polygon.difference

Difference of polygons

d = P.**difference**(**q**) is polygon P minus polygon **q**.

Notes

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

Polygon.display

Display polygon

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

See also

Polygon.char

Polygon.inside

Test if points are inside polygon

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

Polygon.intersect

Intersection of polygon with list of polygons

i = P.**intersect**(**plist**) indicates whether or not the **Polygon** P intersects with

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

Polygon.intersect_line

Intersection of polygon and line segment

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

Polygon.intersection

Intersection of polygons

i = P.**intersection**(**q**) is a **Polygon** representing the **intersection** of polygons P and **q**.

Notes

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

Polygon.moments

Moments of polygon

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

See also

Polygon.area, Polygon.centroid, mpq_poly

Polygon.perimeter

Perimeter of polygon

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

Polygon.plot

Draw polygon

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

P.**plot**(**ls**) as above but pass the arguments **ls** to **plot**.

Notes

- The polygon is added to the current **plot**.

Polygon.transform

Transform polygon vertices

p2 = P.**transform**(**T**) is a new **Polygon** object whose vertices have been transformed by the SE(2) homgoeneous transformation **T** (3×3).

Polygon.union

Union of polygons

i = P.**union**(**q**) is a polygon representing the **union** of polygons P and **q**.

Notes

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

Polygon.xor

Exclusive or of polygons

i = P.**union**(**q**) is a polygon representing the exclusive-or of polygons P and **q**.

Notes

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

PoseGraph

Pose graph

PoseGraph.PoseGraph

the file data

we assume g2o format

```
VERTEX* vertex_id X Y THETA
EDGE* startvertex_id endvertex_id X Y THETA IXX IXY IYY IXT IYT ITT
```

vertex numbers start at 0

PoseGraph.linear_factors

the ids of the vertices connected by the kth edge

id_i=eids(1,k); id_j=eids(2,k);

extract the poses of the vertices and the mean of the edge

```
v_i=vmeans(:,id_i);
v_j=vmeans(:,id_j);
z_ij=emeans(:,k);
```

Prismatic

Robot manipulator prismatic link class

A subclass of the Link class for a prismatic joint defined using standard Denavit-Hartenberg parameters: holds all information related to a robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

Constructors

Prismatic	construct a prismatic joint+link using standard DH

Information/display methods

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

Conversion methods

char	convert to string

Operation methods

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

Testing methods

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

Overloaded operators

+	concatenate links, result is a SerialLink object

Properties (read/write)

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

Notes

- Methods inherited from the Link superclass.
- This is reference class object
- Link class objects can be used in vectors and arrays

References

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

See also

Link, Revolute, SerialLink

Prismatic.Prismatic

Create prismatic robot link object

L = Prismatic(**options**) is a prismatic link object with the kinematic and dynamic parameters specified by the key/value pairs using the standard Denavit-Hartenberg conventions.

Options

'theta', TH	joint angle
'a', A	joint offset (default 0)
'alpha', A	joint twist (default 0)
'standard'	defined using standard D&H parameters (default).
'modified'	defined using modified D&H parameters.
'offset', O	joint variable offset (default 0)
'qlim', **L**	joint limit (default [])
'I', I	link inertia matrix (3×1, 6×1 or 3×3)
'r', R	link centre of gravity (3×1)
'm', M	link mass (1×1)
'G', G	motor gear ratio (default 1)
'B', B	joint friction, motor referenced (default 0)
'Jm', J	motor inertia, motor referenced (default 0)
'Tc', T	Coulomb friction, motor referenced (1×1 or 2×1), (default [0 0])
'sym'	consider all parameter values as symbolic not numeric

Notes

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

See also

Link, Prismatic, RevoluteMDH

PrismaticMDH

Robot manipulator prismatic link class for MDH convention

A subclass of the Link class for a prismatic joint defined using modified Denavit-Hartenberg parameters: holds all information related to a robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

Constructors

PrismaticMDH	construct a prismatic joint+link using modified DH

Information/display methods

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

Conversion methods

char	convert to string

Operation methods

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

Testing methods

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

Overloaded operators

+	concatenate links, result is a SerialLink object

Properties (read/write)

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

Notes

- Methods inherited from the Link superclass.
- This is reference class object
- Link class objects can be used in vectors and arrays
- Modified Denavit-Hartenberg parameters are used

References

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

See also

Link, Prismatic, RevoluteMDH, SerialLink

PrismaticMDH.PrismaticMDH

Create prismatic robot link object using MDH notaton

L = **PrismaticMDH**(**options**) is a prismatic link object with the kinematic and dynamic parameters specified by the key/value pairs using the modified Denavit-Hartenberg conventions.

Options

'theta', TH	joint angle
'a', A	joint offset (default 0)
'alpha', A	joint twist (default 0)
'standard'	defined using standard D&H parameters (default).
'modified'	defined using modified D&H parameters.
'offset', O	joint variable offset (default 0)
'qlim', **L**	joint limit (default [])
'I', I	link inertia matrix (3×1, 6×1 or 3×3)
'r', R	link centre of gravity (3×1)
'm', M	link mass (1×1)
'G', G	motor gear ratio (default 1)
'B', B	joint friction, motor referenced (default 0)
'Jm', J	motor inertia, motor referenced (default 0)
'Tc', T	Coulomb friction, motor referenced (1×1 or 2×1), (default [0 0])
'sym'	consider all parameter values as symbolic not numeric

Notes

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

See also

Link, Prismatic, RevoluteMDH

PRM

Probabilistic RoadMap navigation class

A concrete subclass of the abstract Navigation class that implements the probabilistic roadmap navigation algorithm over an occupancy grid. This performs goal independent planning of roadmaps, and at the query stage finds paths between specific start and goal points.

Methods

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

Example

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

References

- Probabilistic roadmaps for path planning in high dimensional configuration spaces, L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, IEEE Transactions on Robotics and Automation, vol. 12, pp. 566-580, Aug 1996.
- Robotics, Vision & Control, Section 5.2.4, P. Corke, Springer 2011.

See also

Navigation, DXform, Dstar, pgraph

PRM.PRM

Create a PRM navigation object

p = PRM(**map**, **options**) is a probabilistic roadmap navigation object, and **map** is an occupancy grid, a representation of a planar world as a matrix whose elements are 0 (free space) or 1 (occupied).

Options

'npoints', N	Number of sample points (default 100)
'distthresh', D	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.plan

Create a probabilistic roadmap

P.**plan**(**options**) creates the probabilistic roadmap by randomly sampling the free space in the map and building a graph with edges connecting close points. The resulting graph is kept within the object.

Options

'npoints', N	Number of sample points (default is set by constructor)
'distthresh', D	Distance threshold, edges only connect vertices closer than D (default set by constructor)
'movie', M	make a movie of the PRM planning

PRM.plot

Visualize navigation environment

P.**plot**() displays the roadmap and the occupancy grid.

Options

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

Notes

- If a query has been made then the path will be shown.
- Goal and start locations are kept within the object.

PRM.query

Find a path between two points

P.**query**(**start**, **goal**) finds a path ($M \times 2$) from **start** to **goal**.

---

qplot

Plot robot joint angles

qplot(**q**) is a convenience function to plot joint angle trajectories ($M \times 6$) for a 6-axis robot, where each row represents one time step.

The first three joints are shown as solid lines, the last three joints (wrist) are shown as dashed lines. A legend is also displayed.

qplot(**T**, **q**) as above but displays the joint angle trajectory versus time given the time vector **T** ($M \times 1$).

See also

jtraj, plotp, plot

---

randinit

Reset random number generator

RANDINIT resets the defaul random number stream.

See also

randstream

RandomPath

Vehicle driver class

Create a "driver" object capable of steering a Vehicle subclass object through random waypoints within a rectangular region and at constant speed.

The driver object is connected to a Vehicle object by the latter's add_driver() method. The driver's demand() method is invoked on every call to the Vehicle's step() method.

Methods

init	reset the random number generator
demand	speed and steer angle to next waypoint
display	display the state and parameters in human readable form
char	convert to string

plot

Properties

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

Example

```
veh = Bicycle(V);
veh.add_driver( RandomPath(20, 2) );
```

Notes

- It is possible in some cases for the vehicle to move outside the desired region, for instance if moving to a waypoint near the edge, the limited turning circle may cause the vehicle to temporarily move outside.

- The vehicle chooses a new waypoint when it is closer than property closeenough to the current waypoint.
- Uses its own random number stream so as to not influence the performance of other randomized algorithms such as path planning.

Reference

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

See also

Vehicle, Bicycle, Unicycle

RandomPath.RandomPath

Create a driver object

d = **RandomPath**(**d**, **options**) returns a "driver" object capable of driving a Vehicle subclass object through random waypoints. The waypoints are positioned inside a rectangular region of dimension **d** interpreted as:

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

Options

'speed', S	Speed along path (default 1m/s).
'dthresh', **d**	Distance from goal at which next goal is chosen.

See also

Vehicle

RandomPath.char

Convert to string

s = R.**char**() is a string showing driver parameters and state in in a compact human readable format.

RandomPath.demand

Compute speed and heading to waypoint

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

See also

Vehicle

RandomPath.display

Display driver parameters and state

R.**display**() displays driver parameters and state in compact human readable form.

Notes

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

See also

RandomPath.char

RandomPath.init

Reset random number generator

R.**init**() resets the random number generator used to create the waypoints. This enables the sequence of random waypoints to be repeated.

Notes

- Called by Vehicle.run.

See also

randstream

RangeBearingSensor

Range and bearing sensor class

A concrete subclass of the Sensor class that implements a range and bearing angle sensor that provides robot-centric measurements of landmark points in the world. To enable this it holds a references to a map of the world (LandmarkMap object) and a robot (Vehicle subclass object) that moves in SE(2).

The sensor observes landmarks within its angular field of view between the minimum and maximum range.

Methods

reading	range/bearing observation of random landmark
h	range/bearing observation of specific landmark
Hx	Jacobian matrix with respect to vehicle pose dh/dx
Hp	Jacobian matrix with respect to landmark position dh/dp
Hw	Jacobian matrix with respect to noise dh/dw
g	feature position given vehicle pose and observation
Gx	Jacobian matrix with respect to vehicle pose dg/dx
Gz	Jacobian matrix with respect to observation dg/dz

Properties (read/write)

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

landmarklog time history of observed landmarks

Reference

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

See also

Sensor, Vehicle, LandmarkMap, EKF

RangeBearingSensor.RangeBearingSensor

Range and bearing sensor constructor

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

The sensor has specified angular field of view and minimum and maximum range.

Options

'covar', W	covariance matrix (2×2)
'range', xmax	maximum range of sensor
'range', [xmin xmax]	minimum and maximum range of sensor
'angle', TH	angular field of view, from -TH to +TH
'angle', [THMIN THMAX]	detection for angles betwen THMIN and THMAX
'skip', K	return a valid reading on every K'th call
'fail', [TMIN TMAX]	sensor simulates failure between timesteps TMIN and TMAX
'animate'	animate sensor readings

See also

options for sensor constructor

See also

RangeBearingSensor.reading, Sensor.Sensor, Vehicle, LandmarkMap, EKF

RangeBearingSensor.g

Compute landmark location

p = S.g(**x**, **z**) is the world coordinate (2×1) of a feature given the observation **z** (1×2) from a vehicle state with **x** (3×1).

See also

RangeBearingSensor.Gx, RangeBearingSensor.Gz

RangeBearingSensor.Gx

Jacobian dg/dx

J = S.Gx(**x**, **z**) is the Jacobian dg/dx (2×3) at the vehicle state **x** (3×1) for sensor observation **z** (2×1).

See also

RangeBearingSensor.g

RangeBearingSensor.Gz

Jacobian dg/dz

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

See also

RangeBearingSensor.g

RangeBearingSensor.h

Landmark range and bearing

z = S.h(**x**, **k**) is a sensor observation (1×2), range and bearing, from vehicle at pose **x** (1×3) to the **k**'th landmark.

z = S.h(**x**, **p**) as above but compute range and bearing to a landmark at coordinate **p**.

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

Notes

- Noise with covariance W (propertyW) is added to each row of **z**.
- Supports vectorized operation where XV ($N \times 3$) and **z** ($N \times 2$).
- The landmark is assumed visible, field of view and range liits are not applied.

See also

RangeBearingSensor.reading, RangeBearingSensor.Hx, RangeBearingSensor.Hw, RangeBearingSensor.Hp

---

RangeBearingSensor.Hp

Jacobian dh/dp

J = S.**Hp**(**x**, **k**) is the Jacobian dh/dp (2×2) at the vehicle state **x** (3×1) for map landmark **k**.

J = S.**Hp**(**x**, **p**) as above but for a landmark at coordinate **p** (1×2).

See also

RangeBearingSensor.h

RangeBearingSensor.Hw

Jacobian dh/dw

J = S.**Hw**(**x**, **k**) is the Jacobian dh/dw (2×2) at the vehicle state **x** (3×1) for map landmark **k**.

See also

RangeBearingSensor.h

RangeBearingSensor.Hx

Jacobian dh/dx

J = S.**Hx**(**x**, **k**) returns the Jacobian dh/dx (2×3) at the vehicle state **x** (3×1) for map landmark **k**.

J = S.**Hx**(**x**, **p**) as above but for a landmark at coordinate **p**.

See also

RangeBearingSensor.h

RangeBearingSensor.reading

Choose landmark and return observation

[**z**,**k**] = S.**reading**() is an observation of a random visible landmark where **z**=[R,THETA] is the range and bearing with additive Gaussian noise of covariance W (property W). **k** is the index of the map feature that was observed.

The landmark is chosen randomly from the set of all visible landmarks, those within the angular field of view and range limits. If no valid measurement, ie. no features within range, interval subsampling enabled or simulated failure the return is **z**=[] and **k**=0.

Notes

- Noise with covariance W (property W) is added to each row of **z**.
- If 'animate' option set then show a line from the vehicle to the landmark
- If 'animate' option set and the angular and distance limits are set then display that region as a shaded polygon.
- Implements sensor failure and subsampling if specified to constructor.

See also

RangeBearingSensor.h

Revolute

Robot manipulator Revolute link class

A subclass of the Link class for a revolute joint defined using standard Denavit-Hartenberg parameters: holds all information related to a revolute robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

Constructors

Revolute	construct a revolute joint+link using standard DH

Information/display methods

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

Conversion methods

char	convert to string

Operation methods

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

Testing methods

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

Overloaded operators

+	concatenate links, result is a SerialLink object

Properties (read/write)

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

Notes

- Methods inherited from the Link superclass.
- This is reference class object
- Link class objects can be used in vectors and arrays

References

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

See also

Link, Prismatic, RevoluteMDH, SerialLink

Revolute.Revolute

Create revolute robot link object

L = Revolute(**options**) is a revolute link object with the kinematic and dynamic parameters specified by the key/value pairs using the standard Denavit-Hartenberg conventions.

Options

'd', D	joint extension
'a', A	joint offset (default 0)
'alpha', A	joint twist (default 0)
'standard'	defined using standard D&H parameters (default).
'modified'	defined using modified D&H parameters.
'offset', O	joint variable offset (default 0)
'qlim', **L**	joint limit (default [])
'I', I	link inertia matrix (3×1, 6×1 or 3×3)
'r', R	link centre of gravity (3×1)
'm', M	link mass (1×1)
'G', G	motor gear ratio (default 1)
'B', B	joint friction, motor referenced (default 0)
'Jm', J	motor inertia, motor referenced (default 0)
'Tc', T	Coulomb friction, motor referenced (1×1 or 2×1), (default [0 0])
'sym'	consider all parameter values as symbolic not numeric

Notes

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

See also

Link, Prismatic, RevoluteMDH

RevoluteMDH

Robot manipulator Revolute link class for MDH convention

A subclass of the Link class for a revolute joint defined using modified Denavit-Hartenberg parameters: holds all information related to a revolute robot link such as kinematics parameters, rigid-body inertial parameters, motor and transmission parameters.

Constructors

RevoluteMDH	construct a revolute joint+link using modified DH

Information/display methods

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

Conversion methods

char	convert to string

Operation methods

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

Testing methods

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

Overloaded operators

+	concatenate links, result is a SerialLink object

Properties (read/write)

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

Notes

- Methods inherited from the Link superclass.
- This is reference class object
- Link class objects can be used in vectors and arrays
- Modified Denavit-Hartenberg parameters are used

References

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

See also

Link, PrismaticMDH, Revolute, SerialLink

RevoluteMDH.RevoluteMDH

Create revolute robot link object using MDH notation

L = **RevoluteMDH**(**options**) is a revolute link object with the kinematic and dynamic parameters specified by the key/value pairs using the modified Denavit-Hartenberg conventions.

Options

'd', D	joint extension
'a', A	joint offset (default 0)
'alpha', A	joint twist (default 0)
'standard'	defined using standard D&H parameters (default).
'modified'	defined using modified D&H parameters.
'offset', O	joint variable offset (default 0)
'qlim', **L**	joint limit (default [])
'I', I	link inertia matrix (3×1, 6×1 or 3×3)
'r', R	link centre of gravity (3×1)
'm', M	link mass (1×1)
'G', G	motor gear ratio (default 1)
'B', B	joint friction, motor referenced (default 0)
'Jm', J	motor inertia, motor referenced (default 0)
'Tc', T	Coulomb friction, motor referenced (1×1 or 2×1), (default [0 0])
'sym'	consider all parameter values as symbolic not numeric

Notes

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

See also

Link, Prismatic, RevoluteMDH

RRT

Class for rapidly-exploring random tree navigation

A concrete subclass of the abstract Navigation class that implements the rapidly exploring random tree (RRT) algorithm. This is a kinodynamic planner that takes into account the motion constraints of the vehicle.

Methods

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

Properties (read only)

graph	A PGraph object describign the tree

Example

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

References

- Randomized kinodynamic planning, S. LaValle and J. Kuffner, International Journal of Robotics Research vol. 20, pp. 378-400, May 2001.
- Probabilistic roadmaps for path planning in high dimensional configuration spaces, L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, IEEE Transactions on Robotics and Automation, vol. 12, pp. 566-580, Aug 1996.
- Robotics, Vision & Control, Section 5.2.5, P. Corke, Springer 2011.

See also

Navigation, PRM, DXform, Dstar, pgraph

RRT.RRT

Create an RRT navigation object

R = RRT.RRT(**veh**, **options**) is a rapidly exploring tree navigation object for a vehicle kinematic model given by a Vehicle subclass object **veh**.

R = RRT.RRT(**veh**, **map**, **options**) as above but for a region with obstacles defined by the occupancy grid **map**.

Options

'npoints', N	Number of nodes in the tree (default 500)
'simtime', T	Interval over which to simulate kinematic model toward random point (default 0.5s)
'goal', P	Goal position (1×2) or pose (1×3) in workspace
'speed', S	Speed of vehicle [m/s] (default 1)
'root', **R**	Configuration of tree root (3×1) (default [0,0,0])
'revcost', C	Cost penalty for going backwards (default 1)
'range', **R**	Specify rectangular bounds of robot's workspace:

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

Other options are provided by the Navigation superclass.

Notes

- 'range' option is ignored if an occupacy grid is provided.

Reference

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

See also

Vehicle, Bicycle, Unicycle

RRT.char

Convert to string

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

RRT.plan

Create a rapidly exploring tree

R.**plan**(**options**) creates the tree roadmap by driving the vehicle model toward random goal points. The resulting graph is kept within the object.

Options

'goal', P	Goal pose (1×3)
'ntrials', N	Number of path trials (default 50)
'noprogress'	Don't show the progress bar
'samples'	Show progress in a plot of the workspace

- '.' for each random point x_rand
- 'o' for the nearest point which is added to the tree
- red line for the best path

Notes

- At each iteration we need to find a vehicle path/control that moves it from a random point towards a point on the graph. We sample ntrials of random steer angles and velocities and choose the one that gets us closest (computationally slow, since each path has to be integrated over time).

RRT.plot

Visualize navigation environment

R.**plot**() displays the navigation tree in 3D, where the vertical axis is vehicle heading angle. If an occupancy grid was provided this is also displayed.

RRT.query

Find a path between two points

x = R.**path**(**start**, **goal**) finds a **path** ($N \times 3$) from pose **start** (1×3) to pose **goal** (1×3). The pose is expressed as [**x**,Y,THETA].

R.**path**(**start**, **goal**) as above but plots the **path** in 3D, where the vertical axis is vehicle heading angle. The nodes are shown as circles and the line segments are blue for forward motion and red for backward motion.

Notes

- The **path** starts at the vertex closest to the **start** state, and ends at the vertex closest to the **goal** state. If the tree is sparse this might be a poor approximation to the desired start and end.

See also

RRT.plot

rtbdemo

Robot toolbox demonstrations

rtbdemo displays a menu of toolbox demonstration scripts that illustrate:

- fundamental datatypes
  - rotation and homogeneous transformation matrices
  - quaternions
  - trajectories
- serial link manipulator arms
  - forward and inverse kinematics
  - robot animation
  - forward and inverse dynamics
- mobile robots
  - kinematic models and control
  - **path** planning (D*, PRM, Lattice, RRT)
  - localization (EKF, particle filter)
  - SLAM (EKF, pose graph)
  - quadrotor control

rtbdemo(**T**) as above but waits for **T** seconds after every statement, no need to push the enter key periodically.

Notes

- By default the scripts require the user to periodically hit <Enter> in order to move through the explanation.
- Some demos require Simulink
- To quit, close the **rtbdemo** window

RTBPlot

Plot utilities for Robotics Toolbox

RTBPlot.box

Draw a box

BPX(**ax**, **R**, **extent**, **color**, **offset**, **options**) draws a cylinder parallel to axis **ax** ('x', 'y' or 'z') of side length **R** between **extent**(1) and **extent**(2).

RTBPlot.cyl

Draw a cylinder

CYL(**ax**, **R**, **extent**, **color**, **offset**, **options**) draws a cylinder parallel to axis **ax** ('x', 'y' or 'z') of radius **R** between **extent**(1) and **extent**(2).

options are passed through to surf.

See also

surf, RTBPlot.box

RTBPlot.install_teach_panel

robot like object, has n fkine animate methods

---

runscript

Run an M-file in interactive fashion

runscript(**script**, **options**) runs the M-file **script** and pauses after every executable line in the file until a key is pressed. Comment lines are shown without any delay between lines.

Options

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

Notes

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

See also

eval

Sensor

Sensor superclass

An abstract superclass to represent robot navigation sensors.

Methods

plot	plot a line from robot to map feature
display	print the parameters in human readable form
char	convert to string

Properties

robot	The Vehicle object on which the sensor is mounted
map	The PointMap object representing the landmarks around the robot

Reference

Robotics, Vision & Control, Peter Corke, Springer 2011

See also

RangeBearingSensor, EKF, Vehicle, LandmarkMap

Sensor.Sensor

Sensor object constructor

s = Sensor(**vehicle**, **map**, **options**) is a sensor mounted on a vehicle described by the Vehicle subclass object **vehicle** and observing landmarks in a map described by the LandmarkMap class object **map**.

Options

'animate'	animate the action of the laser scanner
'ls', LS	laser scan lines drawn with style ls (default 'r-')
'skip', I	return a valid reading on every I'th call
'fail', T	sensor simulates failure between timesteps T=[TMIN,TMAX]

Notes

- Animation shows a ray from the vehicle position to the selected landmark.

Sensor.char

Convert sensor parameters to a string

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

Sensor.display

Display status of sensor object

S.**display**() displays the state of the sensor object in human-readable form.

Notes

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

See also

Sensor.char

Sensor.plot

Plot sensor reading

S.**plot**(**J**) draws a line from the robot to the **J**'th map feature.

Notes

- The line is drawn using the linestyle given by the property ls
- There is a delay given by the property delay

SerialLink

Serial-link robot class

A concrete class that represents a serial-link arm-type robot. Each link and joint in the chain is described by a Link-class object using Denavit-Hartenberg parameters (standard or modified).

Constructor methods

SerialLink	general constructor
L1+L2	construct from Link objects

Display/plot methods

animate	animate robot model
display	print the link parameters in human readable form
dyn	display link dynamic parameters
edit	display and edit kinematic and dynamic parameters
getpos	get position of graphical robot
plot	display graphical representation of robot
plot3d	display 3D graphical model of robot
teach	drive the graphical robot

Testing methods

islimit	test if robot at joint limit
isconfig	test robot joint configuration
issym	test if robot has symbolic parameters
isprismatic	index of prismatic joints
isrevolute	index of revolute joints
isspherical	test if robot has spherical wrist
isdh	test if robot has standard DH model
ismdh	test if robot has modified DH model

Conversion methods

char	convert to string
sym	convert to symbolic parameters
todegrees	convert joint angles to degrees
toradians	convert joint angles to radians

SerialLink.SerialLink

Create a SerialLink robot object

R = SerialLink(**links**, **options**) is a robot object defined by a vector of Link class objects which includes the subclasses 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 as 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) or prismatic (sigma=1). An optional sixth column is the joint offset.

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 **options** for .plot() to P
‘plotopt3d’, P	set default **options** for .plot3d() to P
‘nofast’	don’t use RNE MEX file
‘configs’, P	provide a cell array of predefined configurations, as name, value pairs

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**, **q**) is an SE3 object (4×4) that transforms between link frames for the **J**'th joint. **q** is a vector $(1 \times N)$ of joint variables. For:

- standard DH parameters, this is from frame $\{\mathbf{J}\text{-}1\}$ to frame $\{\mathbf{J}\}$.
- modified DH parameters, this is from frame $\{\mathbf{J}\}$ to frame $\{\mathbf{J}+1\}$.

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(3:6, q)
```

where q is 1×6 and the elements q(3) .. q(6) are used.

Notes

- Base and tool transforms are not applied.

See also

Link.A

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 $(1 \times N)$ to the manipulator robot R in state **q** $(1 \times N)$ and **qd** $(1 \times N)$, 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 Journa of Dynamic Systems, Measurement and Control, vol. 104, no. 3, pp. 205-211, 1982.

See also

SerialLink.fdyn, 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 with an associated 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 = length(**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 4x4xMxP.

Notes

- Requires the pHRIWARE package which defines CollisionModel class. Available from: https://github.com/bryan91/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 interpretted 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.DH

Convert modified DH model to standard

rmdh = R.**DH**() is a **SerialLink** object that represents the same kinematics as R but expressed using standard **DH** parameters.

Notes

- can only be applied to a model expressed with modified DH parameters.

See also: MDH

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

R.edit displays the kinematic parameters of the robot as an editable table in a new figure.

R.edit('**dyn**') as above but also includes the dynamic parameters in the table.

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**(**tmax**, **ftfun**) integrates the dynamics of the robot over the time interval 0 to **tmax** and returns vectors of time **T** ($K \times 1$), joint position **q** ($K \times N$) and joint velocity **qd** ($K \times N$). The initial joint position and velocity are zero. The torque applied to the joints is computed by the user-supplied control function **ftfun**:

```
TAU = FTFUN(ROBOT, T, Q, QD)
```

where **q** ($1 \times N$) and **qd** ($1 \times N$) are the manipulator joint coordinate and velocity state respectively, and **T** is the current time.

[**ti**,**q**,**qd**] = R.**fdyn**(**T**, **ftfun**, **q0**, **qd0**) as above but allows the initial joint position **q0** ($1 \times N$) and velocity **qd0** (1x) to be specified.

[**T**,**q**,**qd**] = R.**fdyn**(T1, **ftfun**, **q0**, **qd0**, ARG1, ARG2, ...) allows optional arguments to be passed through to the user-supplied control function:

```
TAU = FTFUN(ROBOT, 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 = myftfun(q, qd, qstar, P, D)
tau = (qstar-q)*P + qd*D;  % P, D are 6x6
end
```

and then integrate the robot dynamics with the control:

```
[t,q] = robot.fdyn(10, @(robot, t, q, qd) myftfun(q, qd, qstar, P, D) );
```

where the lambda function ignores the passed values of robot and t but adds qstar, P and D to argument list for myftfun.

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 **ftfun** is not specified, or is given as [], then zero torque is applied to the manipulator joints.
- The MATLAB builtin integration function ode45() is used.

See also

SerialLink.accel, SerialLink.nofriction, SerialLink.rne, ode45

SerialLink.fellipse

Force ellipsoid for seriallink manipulator

R.**fellipse**(**q**, **options**) displays the force ellipsoid for the robot R at pose **q**. The ellipsoid is centered at the tool tip position.

Options

'2d'	Ellipse for translational xy motion, for planar manipulator
'trans'	Ellipsoid for translational motion (default)
'rot'	Ellipsoid for rotational motion

Display options as per plot_ellipse to control ellipsoid face and edge color and transparency.

Example

To interactively update the force ellipsoid while using sliders to change the robot's pose:

```
robot.teach('callback', @(r,q) r.fellipse(q))
```

Notes

- The ellipsoid is tagged with the name of the robot prepended to "**.fellipse**".
- Calling the function with a different pose will update the ellipsoid.

See also

SerialLink.jacob0, SerialLink.vellipse, plot_ellipse

SerialLink.fkine

Forward kinematics

T = R.**fkine**(**q**, **options**) is the pose of the robot end-effector as an SE3 object 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 an array of SE3 objects (K) where the subscript is the index along the path.

[**T**,**all**] = R.**fkine**(**q**) as above but **all** (N) is a vector of SE3 objects describing the pose of the link frames 1 to N.

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.

- If the result is symbolic then each element is simplified.

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**).

Notes

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

See also

Link.friction

SerialLink.gencoords

Vector of symbolic generalized coordinates

q = R.**gencoords**() is a vector $(1 \times N)$ of symbols [q1 q2 ... qN].

[**q**,**qd**] = R.**gencoords**() as above but **qd** is a vector ($1 \times N$) of symbols [qd1 qd2 ... qdN].

[**q**,**qd**,**qdd**] = R.**gencoords**() as above but **qdd** is a vector ($1 \times N$) of symbols [qdd1 qdd2 ... qddN].

See also

SerialLink.genforces

SerialLink.genforces

Vector of symbolic generalized forces

q = R.**genforces**() is a vector ($1 \times N$) of symbols [Q1 Q2 ... QN].

See also

SerialLink.gencoords

SerialLink.get.config

Returnt the joint configuration string

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 **tau** ($1 \times N$) and the manipulator Jacobian in the base frame **jac0** ($6 \times N$) for robot pose **q** ($1 \times N$), where N is the number of robot joints.

[**tau**,**jac0**] = R.gravjac(**q**,**grav**) as above but gravitational acceleration is given explicitly by **grav** (3×1).

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 robot configuration. **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.gravjac, SerialLink.rne, SerialLink.itorque, SerialLink.coriolis

SerialLink.ikcon

Inverse kinematics by optimization with joint limits

q = R.**ikcon**(**T**, **options**) are the joint coordinates ($1 \times N$) corresponding to the robot end-effector pose **T** which is an SE3 object or homogenous transform matrix (4×4), and N is the number of robot joints. **options** is an optional list of name/value pairs than can be passed to fmincon.

q = robot.**ikunc**(**T**, **q0**, **options**) as above but specify the initial joint coordinates **q0** used for the minimisation.

[**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,OUTPUT] = robot.ikcon(**T**, ...) as above but also returns a structure with information such as total number of iterations, and final objective function value. See the documentation of fmincon for a complete list.

Trajectory operation

In all cases if **T** is a vector of SE3 objects ($1 \times M$) or a homogeneous transform sequence ($4 \times 4 \times M$) then 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 MATLAB 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

Inverse kinematics by optimization without joint limits

q = R.**ikine**(**T**) are the joint coordinates ($1 \times N$) corresponding to the robot end-effector pose **T** which is an SE3 object or homogenous transform matrix (4×4), and N is the number of robot joints.

This method can be used for robots with any number of degrees of freedom.

Options

'ilimit', L	maximum number of iterations (default 500)
'rlimit', L	maximum number of consecutive step rejections (default 100)
'tol', **T**	final error tolerance (default 1e-10)
'lambda', L	initial value of lambda (default 0.1)
'lambdamin', M	minimum allowable value of lambda (default 0)
'quiet'	be quiet
'verbose'	be verbose
'mask', M	mask vector (6×1) that correspond to translation in X, Y and Z, and rotation about X, Y and Z respectively.
'q0', **q**	initial joint configuration (default all zeros)
'search'	search over all configurations
'slimit', L	maximum number of search attempts (default 100)
'transpose', A	use Jacobian transpose with step size A, rather than Levenberg-Marquadt

Trajectory operation

In all cases if **T** is a vector of SE3 objects ($1 \times M$) or a homogeneous transform sequence ($4 \times 4 \times M$) then 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.

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.

In this case we specify the 'mask' option where the mask vector (1×6) 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 use the option: 'mask', [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 **T** in world coordinates and the achievable orientations are a function of the tool position.

References

- Robotics, Vision & Control, P. Corke, Springer 2011, Section 8.4.

Notes

- This has been completely reimplemented in RTB 9.11
- Does NOT require MATLAB Optimization Toolbox.
- Solution is computed iteratively.
- Implements a Levenberg-Marquadt variable step size solver.
- The tolerance is computed on the norm of the error between current and desired tool pose. This norm is computed from distances and angles without any kind of weighting.
- The inverse kinematic solution is generally not unique, and depends on the initial guess Q0 (defaults to 0).
- 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.

- 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 are not considered in this solution.
- The 'search' option peforms a brute-force search with initial conditions chosen from the entire configuration space.
- If the 'search' option is used any prismatic joint must have joint limits defined.

See also

SerialLink.ikcon, SerialLink.ikunc, SerialLink.fkine, SerialLink.ikine6s

---

SerialLink.ikine3

Inverse kinematics for 3-axis robot with no wrist

q = R.**ikine3**(**T**) is the joint coordinates (1×3) corresponding to the robot end-effector pose **T** represented by the homogenenous transform. This is a analytic solution for a 3-axis robot (such as the first three joints of a robot like the Puma 560).

q = R.**ikine3**(**T**, **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 **q**.

Trajectory operation

In all cases if **T** is a vector of SE3 objects ($1 \times M$) or a homogeneous transform sequence ($4 \times 4 \times M$) then returns the joint coordinates corresponding to each of the transforms in the sequence. **q** is $M \times 3$.

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

q = R.**ikine**(**T**) are the joint coordinates $(1 \times N)$ corresponding to the robot end-effector pose **T** which is an SE3 object or homogenenous transform matrix (4×4), and N is the number of robot joints. 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 **q** $(M \times N)$ with each row representing the joint angles at the corresponding pose.

q = R.**IKINE6S**(**T**, **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)

Trajectory operation

In all cases if **T** is a vector of SE3 objects $(1 \times M)$ or a homogeneous transform sequence $(4 \times 4 \times M)$ then R.ikcon() returns the joint coordinates corresponding to each of the transforms in the sequence.

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 kinematics for the various cases determined using ikine_sym.
- The inverse kinematic solution is generally not unique, and depends on the configuration string.
- Joint offsets, if defined, are added to the inverse kinematics to generate **q**.
- 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

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 solutions for each joint, ie. **q**$\{1\}$ is the solution for joint 1. A cell may contain an array of solutions. The solution is expressed in terms of other joint angles and elements of the desired end-point pose which is represented by the symbolic matrix (3×4) 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 is the number of joint angles solved for.

Options

'file', F	Write the solution to an m-file named F
'Tpost', T	Add a symbolic 4×4 matrix T to the end of the chain

Example

```
mdl_planar2
sol = p2.ikine_sym(2);
length(sol)

q1 = sol{1}    % are the solution for joint 1
q2 = sol{2}    % is the solution for joint 2
length(q1)
ans =

2      % there are 2 solutions for this joint

q1(1)        % one solution for q1
q1(2);       % the other solution for q1
```

Notes

- ignores tool and base transforms.

References

- Robot manipulators: mathematics, programming and control Richard Paul, MIT Press, 1981.
- The kinematics of manipulators under computer control, D.L. Pieper, Stanford report AI 72, October 1968.

Notes

- Requires the MATLAB Symbolic Math Toolbox.
- 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 homogenenous 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 contraint on joint angles, useful when N is large (default 0)
'qlimits'	Enforce joint limits
'ilimit', L	Iteration limit (default 1000)
'nolm'	Disable Levenberg-Marquadt

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 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 contraints if ‘qlimits’ is set.

See also

fminsearch, fmincon, SerialLink.fkine, SerialLink.ikine, tr2angvec

SerialLink.ikunc

Inverse manipulator by optimization without joint limits

q = R.**ikunc**(**T**, **options**) are the joint coordinates $(1 \times N)$ corresponding to the robot end-effector pose **T** which is an SE3 object or homogenenous transform matrix (4×4), and N is the number of robot joints. **options** is an optional list of name/value pairs than can be passed to fminunc.

q = robot.**ikunc**(**T**, **q0**, **options**) as above but specify the initial joint coordinates **q0** used for the minimisation.

[**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.

Trajectory operation

In all cases if **T** is a vector of SE3 objects $(1 \times M)$ or a homogeneous transform sequence $(4 \times 4 \times M)$ then 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 fminunc from the MATLAB 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 **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.ikcon, fmincon, SerialLink.ikine, SerialLink.fkine

SerialLink.inertia

Manipulator inertia matrix

i = R.**inertia**(**q**) is the symmetric joint **inertia** matrix ($N \times N$) which relates joint torque to joint acceleration for the robot at joint configuration **q**.

If **q** is a matrix ($K \times N$), each row is interpretted 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 **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.isdh

Test if SerialLink object has a standard DH model

v = R.**isdh**() is true if the **SerialLink** manipulator R has a standard DH model

See also: ismdh

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 not used by many methods, exceptions being:
  - ikcon() to specify joint constraints for inverse kinematics.
  - by plot() for prismatic joints to help infer the size of the workspace

See also

Link.islimit

SerialLink.ismdh

Test if SerialLink object has a modified DH model

v = R.**ismdh**() is true if the **SerialLink** manipulator R has a modified DH model

See also: isdh

SerialLink.isprismatic

identify prismatic joints

X = R.isprismatic is a list of logical variables, one per joint, true if the corresponding joint is prismatic, otherwise false.

See also

Link.isprismatic, SerialLink.isrevolute

SerialLink.isrevolute

identify revolute joints

X = R.isrevolute is a list of logical variables, one per joint, true if the corresponding joint is revolute, otherwise false.

See also

Link.isrevolute, SerialLink.isprismatic

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

Test 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)

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 interpretted 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 XYZ roll-pitch-yaw angles
'eul'	Compute analytical Jacobian with rotation rates in terms of Euler angles
'exp'	Compute analytical Jacobian with rotation rates in terms of exponential coordinates
'trans'	Return translational submatrix of Jacobian
'rot'	Return rotational submatrix of Jacobian

Note

- End-effector spatial velocity is a vector (6×1): the first 3 elements are translational velocity, the last 3 elements are rotational velocity as angular velocity (default), RPY angle rate or Euler angle rate.
- This Jacobian accounts for a base and/or tool transform if set.
- 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.

See also

SerialLink.jacobe, jsingu, deltatr, tr2delta, jsingu

SerialLink.jacob_dot

Derivative of Jacobian

jdq = R.**jacob_dot**(**q**, **qd**) is the product (6×1) of the derivative of the Jacobian (in the world frame) and the joint rates.

Notes

- This term appears in the formulation 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. Angleles, Springer 2003.
- A unified approach for motion and force control of robot manipulators: The operational space formulation

O Khatib, IEEE Journal on Robotics and Automation, 1987.

See also

SerialLink.jacob0, diff2tr, tr2diff

SerialLink.jacobe

Jacobian in end-effector frame

je = R.jacobe(**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 = **je***QD in the end-effector frame.

Options

'trans'	Return translational submatrix of Jacobian
'rot'	Return rotational submatrix of Jacobian

Notes

- Was joacobn() is earlier version of the Toolbox.
- This Jacobian accounts for a tool transform if one is set.
- This Jacobian is often referred to as the geometric Jacobian.
- Prior to release 10 this function was named jacobn.

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.jointdynamics

Transfer function of joint actuator

tf = R.**jointdynamic**(**q**) is a vector of N continuous-time transfer function objects that represent the transfer function 1/(Js+B) for each joint based on the dynamic parameters of the robot and the configuration **q** $(1 \times N)$. N is the number of robot joints.

% **tf** = R.**jointdynamic**(**q**, QD) as above but include the linearized effects of Coulomb friction when operating at joint velocity QD $(1 \times N)$.

Notes

- Coulomb friction is ignoredf.

See also

tf, SerialLink.rne

SerialLink.jtraj

Joint space trajectory

q = R.**jtraj**(**T1**, **t2**, **k**, **options**) is a joint space trajectory $(\mathbf{k} \times N)$ where the joint coordinates reflect motion from end-effector pose **T1** to **t2** in **k** steps, where N is the number of robot joints. **T1** and **t2** are SE3 objects or homogeneous transformation matrices (4×4). The trajectory **q** has one row per time step, and one column per joint.

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().

Notes

- Zero boundary conditions for velocity and acceleration are assumed.
- Additional options are passed as trailing arguments to the inverse kinematic function, eg. configuration options like 'ru'.

See also

jtraj, SerialLink.ikine, SerialLink.ikine6s

SerialLink.maniplty

Manipulability measure

m = R.**maniplty**(**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 $(\mathbf{m} \times N)$ then **m** $(\mathbf{m} \times 1)$ is a vector of manipulability indices for each joint configuration specified by a row of **q**.

[**m**,**ci**] = R.**maniplty**(**q**, **options**) as above, but for the case of the Asada measure returns the Cartesian inertia matrix **ci**.

R.**maniplty**(**q**) displays the translational and rotational manipulability.

Two measures can be computed:

- Yoshikawa's manipulability measure is based on the shape of the velocity ellipsoid and depends only on kinematic parameters (default).
- 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

'trans'	manipulability for transational motion only (default)
'rot'	manipulability for rotational motion only
'all'	manipulability for all motions
'dof', D	D is a vector (1×6) 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 (1st edition) 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.
- Robotics, Vision & Control, P. Corke, Springer 2011.

See also

SerialLink.inertia, SerialLink.jacob0

SerialLink.MDH

Convert standard DH model to modified

rmdh = R.**MDH**() is a **SerialLink** object that represents the same kinematics as R but expressed using modified DH parameters.

Notes

- can only be applied to a model expressed with standard DH parameters.

See also: DH

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 general 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** (1×6) 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, 'e' 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 **f** = 'e'.
- Must have a constant wrench - no trajectory support for this yet.

Author

Bryan Moutrie

See also

SerialLink.paycap, SerialLink.jacob0, SerialLink.jacobe

SerialLink.paycap

Static payload capacity of a robot

[**wmax**,**J**] = R.**paycap**(**q**, **w**, **f**, **tlim**) returns the maximum permissible payload wrench **wmax** (1×6) 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 (1×6), **f** the wrench reference frame (either '0' or 'e') 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 **f** = 'e'.

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.

R.**payload**(0) removes added **payload**

Notes

- An added **payload** will affect the inertia, Coriolis and gravity terms.
- Sets, rather than adds, the **payload**. Mass and CoM of the last link is overwritten.

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);
```

See also

SerialLink.rne

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 betwen 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 an animation to the movie M
'trail', L	Draw a line recording the tip path, with line style L. L can be a cell array, eg. {'r', 'LineWidth', 2}
'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.
'top'	View from the top.
'[no]shading'	Enable Gouraud shading (default true)
'lightpos', L	Position of the light source (default [0 0 20])
'[no]name'	Display the robot's name
'[no]wrist'	Enable display of wrist coordinate frame
'xyz'	Wrist axis label is XYZ
'noa'	Wrist axis label is NOA
'[no]arrow'	Display wrist frame with 3D arrows
'[no]tiles'	Enable tiled floor (default true)
'tilesize', S	Side length of square tiles on the floor
'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])
'pjointcolor', C	Colorspec for prismatic joint boxes (default [0.4 1 .03])
'jointdiam', D	Diameter of joint cylinder in scale units (default 5)
'linkcolor', C	Colorspec of links (default 'b')
'[no]base'	Enable display of base 'pedestal'
'basecolor', C	Color of base (default 'k')
'basewidth', W	Width of base (default 3)

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

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

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

Graphical annotations and options

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

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

which are controlled by **options**.

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

Figure behaviour

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

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

The 'movie' **options** saves the animation as a movie file or separate frames in a folder

- 'movie','file.mp4' saves as an MP4 movie called file.mp4
- 'movie','folder' saves as files NNNN.png into the specified folder
  - The specified folder will be created
  - NNNN are consecutive numbers: 0000, 0001, 0002 etc.
  - 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 neccessary to clear the figure.
- The link segments do not neccessarily represent the links of the robot, they are a pipe network that joins the origins of successive link coordinate frames.
- Delay betwen 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.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 transparant, 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 betwen 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. L can be a cell array, eg. {'r', 'LineWidth', 2}
'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 files (ascii or binary) with extension .stl.
- The solid models live in RVCTOOLS/robot/data/meshes.
- 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 overridden 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

Making an animation

The 'movie' **options** saves the animation as a movie file or separate frames in a folder

- 'movie','file.mp4' saves as an MP4 movie called file.mp4
- 'movie','folder' saves as files NNNN.png into the specified folder
  - The specified folder will be created
  - NNNN are consecutive numbers: 0000, 0001, 0002 etc.
  - To convert frames to a movie use a command like:

```
ffmpeg -r 10 -i %04d.png out.avi
```

Authors

- Peter Corke, based on existing code for plot().
- Bryan Moutrie, demo code on the Google Group for connecting ARTE and RTB.

Acknowledgments

- STL files are from ARTE: A ROBOTICS TOOLBOX FOR EDUCATION by Arturo Gil (https://arvc.umh.es/arte) are included, with permission.
- The various authors of STL reading code on file exchange, see stlRead.m

See also

SerialLink.plot, plotbotopt3d, SerialLink.animate, SerialLink.teach, stlRead

---

SerialLink.plus

Append a link objects to a robot

R+L is a **SerialLink** object formed appending a deep copy of the Link L to the **SerialLink** robot R.

Notes

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

See also

Link.plus

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 MATLAB 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**, **options**) 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(**x**, **options**) as above where **x=[q,qd,qdd]** $(1 \times 3N)$.

[**tau**,**wbase**] = R.rne(**x**, **grav**, **fext**) as above but the extra output is the wrench on the base.

Options

'gravity', G	specify gravity acceleration (default [0,0,9.81])
'fext', W	specify wrench acting on the end-effector W=[Fx Fy Fz Mx My Mz]
'slow'	do not use MEX file

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 'slow' option is not given, and
- 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 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.
- If a model has no dynamic parameters set the result is zero.

See also

SerialLink.accel, SerialLink.gravload, SerialLink.inertia

SerialLink.simplify_powers

a list of simplifications

substitute S^2 = 1-C^2, S^4=(1-C^2)^2

SerialLink.teach

Graphical teach pendant

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

R.**teach**(**options**) adds a slider panel to a current robot plot.

R.**teach**(**q**, **options**) as above but the robot joint angles are set to **q** $(1 \times N)$.

Options

'eul'	Display tool orientation in Euler angles (default)
'rpy'	Display tool orientation in roll/pitch/yaw angles
'approach'	Display tool orientation as approach vector (z-axis)
'[no]deg'	Display angles in degrees (default true)
'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 removes the **teach** panel from the robot plot.

Notes

- If the robot is displayed in several windows, only one has the **teach** panel added.
- All currently displayed robots move as the sliders are adjusted.
- The slider limits are derived from the joint limit properties. If not set then for
  - a revolute joint they are assumed to be [-pi, +pi]
  - a prismatic joint they are assumed unknown and an error occurs.

See also

SerialLink.plot, SerialLink.getpos

SerialLink.todegrees

Convert joint angles to degrees

q2 = R.**todegrees**(**q**) is a vector of joint coordinates where those elements corresponding to revolute joints are converted from radians to degrees. Elements corresponding to prismatic joints are copied unchanged.

See also

SerialiLink.toradians

SerialLink.toradians

Convert joint angles to radians

q2 = R.**toradians**(**q**) is a vector of joint coordinates where those elements corresponding to revolute joints are converted from degrees to radians. Elements corresponding to prismatic joints are copied unchanged.

See also

SerialiLink.todegrees

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, troty, trotz, transl, DHFactor

SerialLink.twists

Joint axis twists

[**tw**,**T0**] = R.**twists**(**q**) is a vector of Twist objects ($1 \times N$) that represent the axes of the joints for the robot with joint coordinates **q** ($1 \times N$). **T0** is an SE3 object representing the pose of the tool.

[**tw**,**T0**] = R.**twists**() as above but the joint coordinates are taken to be zero.

Notes

tw,**T0** is the product of exponential representation of the robot's forward kinematics: prod([**tw**.exp(**q**) **T0**])

See also

twist

SerialLink.vellipse

Velocity ellipsoid for seriallink manipulator

R.**vellipse**(**q**, **options**) displays the velocity ellipsoid for the robot R at pose **q**. The ellipsoid is centered at the tool tip position.

Options

'2d'	Ellipse for translational xy motion, for planar manipulator
'trans'	Ellipsoid for translational motion (default)
'rot'	Ellipsoid for rotational motion

Display options as per plot_ellipse to control ellipsoid face and edge color and transparency.

Example

To interactively update the velocity ellipsoid while using sliders to change the robot's pose:

```
robot.teach('callback', @(r,q) r.vellipse(q))
```

Notes

- The ellipsoid is tagged with the name of the robot prepended to "**.vellipse**".
- Calling the function with a different pose will update the ellipsoid.

See also

SerialLink.jacob0, SerialLink.fellipse, plot_ellipse

---

simulinkext

Return file extension of Simulink block diagrams.

str = **simulinkext**() is either

- '.mdl' if Simulink version number is less than 8
- '.slx' if Simulink version numberis larger or equal to 8

Notes

The file extension for Simulink block diagrams has changed from Matlab 2011b to Matlab 2012a. This function is used for backwards compatibility.

Author

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

See also

symexpr2slblock, doesblockexist, distributeblocks

---

startup_rtb

Initialize MATLAB paths for Robotics Toolbox

Adds demos, data, contributed code and examples to the MATLAB path, and adds also to Java class path.

Notes

- This sets the paths for the current session only.
- To make the settings persistent across sessions you can:
  - Add this script to your MATLAB startup.m script.
  - After running this script run PATHTOOL and save the path.

See also

path, addpath, pathtool, javaaddpath

stlRead

reads any STL file not depending on its format

[**v**, **f**, **n**, **name**] = **stlread**(**fileName**) reads the STL format file (ASCII or binary) and returns vertices V, faces F, normals N and NAME is the **name** of the STL object (NOT the **name** of the STL file).

Authors

- from MATLAB File Exchange by Pau Mico, https://au.mathworks.com/matlabcentral/fileexchange/51200-stltools
- Copyright (c) 2015, Pau Mico
- Copyright (c) 2013, Adam H. Aitkenhead
- Copyright (c) 2011, Francis Esmonde-White

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

tpoly

Generate scalar polynomial trajectory

[**s**,**sd**,**sdd**] = **tpoly**(**s0**, **sf**, **m**) is a scalar trajectory (**m** $\times$ 1) that varies smoothly from **s0** to **sf** in **m** steps using a quintic (5th order) polynomial. Velocity and acceleration can be optionally returned as **sd** (**m** $\times$ 1) and **sdd** (**m** $\times$ 1) respectively.

tpoly(**s0**, **sf**, **m**) as above but plots **s**, **sd** and **sdd** versus time in a single figure.

[**s**,**sd**,**sdd**] = **tpoly**(**s0**, **sf**, **T**) as above but the trajectory is computed at each point in the time vector **T** (**m** $\times$ 1).

[**s**,**sd**,**sdd**] = **tpoly**(**s0**, **sf**, **T**, **qd0**, **qd1**) as above but also specifies the initial and final velocity of the trajectory.

Notes

- If **m** is given
  - Velocity is in units of distance per trajectory step, not per second.
  - Acceleration is in units of distance per trajectory step squared, not per second squared.
- If **T** is given then results are scaled to units of time.
- The time vector **T** is assumed to be monotonically increasing, and time scaling is based on the first and last element.

Reference:

Robotics, Vision & Control Chap 3 Springer 2011

See also

lspb, jtraj

Unicycle

vehicle class

This concrete class models the kinematics of a differential steer vehicle (unicycle model) on a plane. For given steering and velocity inputs it updates the true vehicle state and returns noise-corrupted odometry readings.

Methods

init	initialize vehicle state
f	predict next state based on odometry
step	move one time step and return noisy odometry
control	generate the control inputs for the vehicle
update	update the vehicle state
run	run for multiple time steps
Fx	Jacobian of f wrt x
Fv	Jacobian of f wrt odometry noise
gstep	like step() but displays vehicle
plot	plot/animate vehicle on current figure
plot_xy	plot the true path of the vehicle
add_driver	attach a driver object to this vehicle
display	display state/parameters in human readable form
char	convert to string

Class methods

plotv	plot/animate a pose on current figure

Properties (read/write)

x	true vehicle state: x, y, theta (3×1)
V	odometry covariance (2×2)
odometry	distance moved in the last interval (2×1)
rdim	dimension of the robot (for drawing)
L	length of the vehicle (wheelbase)
alphalim	steering wheel limit
maxspeed	maximum vehicle speed
T	sample interval
verbose	verbosity
x_hist	history of true vehicle state ($N \times 3$)
driver	reference to the driver object
x0	initial state, restored on init()

Examples

Odometry covariance (per timstep) is

```
V = diag([0.02, 0.5*pi/180].^2);
```

Create a vehicle with this noisy odometry

```
v = Unicycle( 'covar', diag([0.1 0.01].^2) );
```

and display its initial state

```
v
```

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

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

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

```
v
```

We can add a driver object

```
v.add_driver( RandomPath(10) )
```

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

```
v.run(1000)
```

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

Notes

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

Reference

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

See also

RandomPath, EKF

Unicycle.Unicycle

Unicycle object constructor

v = **Unicycle**(**va**, **options**) creates a **Unicycle** object with actual odometry covariance **va** (2×2) matrix corresponding to the odometry vector [dx dtheta].

Options

'W', W	Wheel separation [m] (default 1)
'vmax', S	Maximum speed (default 5m/s)
'x0', x0	Initial state (default (0,0,0))
'dt', T	Time interval
'rdim', R	Robot size as fraction of plot window (default 0.2)
'verbose'	Be verbose

Notes

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

Unicycle.char

Convert to a string

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

See also

Unicycle.display

Unicycle.deriv

be called from a continuous time integrator such as ode45 or Simulink

---

Unicycle.f

Predict next state based on odometry

xn = V.**f**(**x**, **odo**) is the predicted next state **xn** (1×3) based on current state **x** (1×3) and odometry **odo** (1×2) = [distance, heading_change].

xn = V.**f**(**x**, **odo**, **w**) as above but with odometry noise **w**.

Notes

- Supports vectorized operation where **x** and **xn** ($N \times 3$).

Unicycle.Fv

Jacobian df/dv

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

See also

Unicycle.F, Vehicle.Fx

Unicycle.Fx

Jacobian df/dx

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

See also

Unicycle.f, Vehicle.Fv

Unicycle.update

Update the vehicle state

odo = V.**update**(**u**) is the true odometry value for motion with **u**=[speed,steer].

Notes

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

Vehicle

Abstract vehicle class

This abstract class models the kinematics of a mobile robot moving on a plane and with a pose in SE(2). For given steering and velocity inputs it updates the true vehicle state and returns noise-corrupted odometry readings.

Methods

Vehicle	constructor
add_driver	attach a driver object to this vehicle
control	generate the control inputs for the vehicle
f	predict next state based on odometry
init	initialize vehicle state
run	run for multiple time steps
run2	run with control inputs
step	move one time step and return noisy odometry
update	**update** the vehicle state

Plotting/display methods

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

Properties (read/write)

x	true vehicle state: x, y, theta (3×1)
V	odometry covariance (2×2)
odometry	distance moved in the last interval (2×1)
rdim	dimension of the robot (for drawing)
L	length of the vehicle (wheelbase)
alphalim	steering wheel limit
speedmax	maximum vehicle speed
T	sample interval
verbose	verbosity
x_hist	history of true vehicle state $(N \times 3)$
driver	reference to the driver object
x0	initial state, restored on init()

Examples

If veh is an instance of a Vehicle class then we can add a driver object

```
veh.add_driver( RandomPath(10) )
```

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

```
veh.run(1000)
```

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

Notes

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

Reference

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

See also

Bicycle, Unicycle, RandomPath, EKF

Vehicle.Vehicle

Vehicle object constructor

v = Vehicle(**options**) creates a Vehicle object that implements the kinematic model of a wheeled vehicle.

Options

'covar', C	specify odometry covariance (2×2) (default 0)
'speedmax', S	Maximum speed (default 1m/s)
'L', L	Wheel base (default 1m)
'x0', x0	Initial state (default (0,0,0))
'dt', T	Time interval (default 0.1)
'rdim', R	Robot size as fraction of plot window (default 0.2)
'verbose'	Be verbose

Notes

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

Vehicle.add_driver

Add a driver for the vehicle

V.add_driver(**d**) connects a driver object **d** to the vehicle. The driver object has one public method:

```
[speed, steer] = D.demand();
```

that returns a speed and steer angle.

Notes

- The Vehicle.step() method invokes the driver if one is attached.

See also

Vehicle.step, RandomPath

Vehicle.char

Convert to string

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

See also

Vehicle.display

Vehicle.control

Compute the control input to vehicle

u = V.**control**(**speed**, **steer**) is a **control** input (1×2) = [speed,steer] based on provided controls **speed**,**steer** to which speed and steering angle limits have been applied.

u = V.**control**() as above but demand originates with a "driver" object if one is attached, the driver's DEMAND() method is invoked. If no driver is attached then speed and steer angle are assumed to be zero.

See also

Vehicle.step, RandomPath

Vehicle.display

Display vehicle parameters and state

V.**display**() displays vehicle parameters and state in compact human readable form.

Notes

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

See also

Vehicle.char

Vehicle.init

Reset state

V.**init**() sets the state V.x := V.x0, initializes the driver object (if attached) and clears the history.

V.**init**(**x0**) as above but the state is initialized to **x0**.

Vehicle.path

Compute path for constant inputs

xf = V.**path**(**tf**, **u**) is the final state of the vehicle (3×1) from the initial state (0,0,0) with the control inputs **u** (vehicle specific). **tf** is a scalar to specify the total integration time.

xp = V.**path**(**tv**, **u**) is the trajectory of the vehicle ($N \times 3$) from the initial state (0,0,0) with the control inputs **u** (vehicle specific). T is a vector (N) of times for which elements of the trajectory will be computed.

xp = V.**path**(**T**, **u**, **x0**) as above but specify the initial state.

Notes

- Integration is performed using ODE45.
- The ODE being integrated is given by the deriv method of the vehicle object.

See also

ode45

Vehicle.plot

Plot vehicle

The vehicle is depicted graphically as a narrow triangle that travels "point first" and has a length V.rdim.

V.**plot**(**options**) plots the vehicle on the current axes at a pose given by the current robot state. If the vehicle has been previously plotted its pose is updated.

V.**plot**(**x**, **options**) as above but the robot pose is given by **x** (1×3).

H = V.**plotv**(**x**, **options**) draws a representation of a ground robot as an oriented triangle with pose **x** (1×3) [x,y,theta]. **H** is a graphics handle.

V.**plotv**(**H**, **x**) as above but updates the pose of the graphic represented by the handle **H** to pose **x**.

Options

'scale', S	Draw vehicle with length S x maximum axis dimension
'size', S	Draw vehicle with length S
'color', C	Color of vehicle.
'fill'	Filled
'trail', S	Trail with line style S, use line() name-value pairs

Example

```
veh.plot('trail', {'Color', 'r', 'Marker', 'o', 'MarkerFaceColor', 'r', 'MarkerEdgeColor', 'r', 'Mark
```

Vehicle.plot_xy

Plots true path followed by vehicle

V.**plot_xy**() plots the true xy-plane path followed by the vehicle.

V.**plot_xy**(**ls**) as above but the line style arguments **ls** are passed to plot.

Notes

- The path is extracted from the x_hist property.

Vehicle.plotv

Plot ground vehicle pose

H = **Vehicle**.**plotv**(**x**, **options**) draws a representation of a ground robot as an oriented triangle with pose **x** (1×3) [x,y,theta]. **H** is a graphics handle. If **x** ($N \times 3$) is a matrix it is considered to represent a trajectory in which case the vehicle graphic is animated.

Vehicle.**plotv**(**H**, **x**) as above but updates the pose of the graphic represented by the handle **H** to pose **x**.

Options

'scale', S	Draw vehicle with length S x maximum axis dimension
'size', S	Draw vehicle with length S
'fillcolor', C	Color of vehicle.
'fps', F	Frames per second in animation mode (default 10)

Example

Generate some path $3 \times N$

```
p = PRM.plan(start, goal);
```

Set the axis dimensions to stop them rescaling for every point on the path

```
axis([-5 5 -5 5]);
```

Now invoke the static method

```
Vehicle.plotv(p);
```

Notes

- This is a class method.

See also

Vehicle.plot

Vehicle.run

Run the vehicle simulation

V.**run**(**n**) runs the vehicle model for **n** timesteps and plots the vehicle pose at each step.

p = V.**run**(**n**) runs the vehicle simulation for **n** timesteps and return the state history (**n** $\times$ 3) without plotting. Each row is (x,y,theta).

See also

Vehicle.step, Vehicle.run2

Vehicle.run2

run the vehicle simulation with control inputs

p = V.**run2**(**T**, **x0**, **speed**, **steer**) runs the vehicle model for a time **T** with speed **speed** and steering angle **steer**. **p** ($N \times 3$) is the path followed and each row is (x,y,theta).

Notes

- Faster and more specific version of run() method.
- Used by the RRT planner.

See also

Vehicle.run, Vehicle.step, RRT

Vehicle.step

Advance one timestep

odo = V.**step**(**speed**, **steer**) updates the vehicle state for one timestep of motion at specified **speed** and **steer** angle, and returns noisy odometry.

odo = V.**step**() updates the vehicle state for one timestep of motion and returns noisy odometry. If a "driver" is attached then its DEMAND() method is invoked to compute speed and steer angle. If no driver is attached then speed and steer angle are assumed to be zero.

Notes

- Noise covariance is the property V.

See also

Vehicle.control, Vehicle.update, Vehicle.add_driver

Vehicle.update

Update the vehicle state

odo = V.**update**(**u**) is the true odometry value for motion with **u**=[speed,steer].

Notes

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

Vehicle.verbosity

Set verbosity

V.**verbosity**(**a**) set **verbosity** to **a**. **a**=0 means silent.

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 vosion 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 foramt.

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 (3×3).

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 (1×3) 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 (4×4).

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 communcations 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** (3×3).

V.**setorient**(**H**, **T**) sets the orientation of V-REP object with integer handle **H** to rotational component of homogeneous transformation matrix **T** (4×4).

V.**setorient**(**H**, **E**) sets the orientation of V-REP object with integer handle **H** to ZYZ Euler angles (1×3).

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** (1×3).

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** (4×4).

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) betwen 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 **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 senor named **name** in the V-REP environment.

Options

'fov', A	Specify field of view in degreees (default 60)
'resolution', N	Specify resolution. If scalar $N \times N$ else N(1)xN(2)
'clipping', Z	Specify near Z(1) and far Z(2) clipping boundaries

Notes

- Default parameters are set in the V-REP environmen
- 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 foramt.

See also

VREP.display

VREP_camera.getangle

Fet 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 (1×2) 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 (1×2) 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** $\times$ **R** if **R** is a scalar or **R**(1)xR(2) if it is a 2-vector.

Notes

- By default V-REP cameras seem to have very low (32×32) 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 foramt.

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
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
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 (3×3).

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.getopos, VREP_obj.getpose

VREP_obj.getpos

Get position of V-REP object

V.**getpos**() is the position (1×3) 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 (4×4) 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** (3×3).

V.**setorient**(**T**) sets the orientation of the corresponding V-REP object to rotational component of homogeneous transformation matrix **T** (4×4).

V.**setorient**(**E**) sets the orientation of the corresponding V-REP object to ZYZ Euler angles (1×3).

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** (1×3).

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** (4×4).

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 (6×1) in the frame represented by the homogeneous transform **T** (4×4) corresponding to the world frame wrench **w** (6×1).

The wrenches **w** and **wt** are 6-vectors of the form [Fx Fy Fz Mx My Mz]'.

See also

tr2delta, tr2jac

---

xaxis

Set X-axis scaling

xaxis(**max**) set x-axis scaling from 0 to **max**.

xaxis(**min**, **max**) set x-axis scaling from **min** to **max**.

xaxis([**min max**]) as above.

xaxis restore automatic scaling for x-axis.

See also

yaxis

---

yaxis

Y-axis scaling

yaxis(**max**) set y-axis scaling from 0 to **max**.

yaxis(**min**, **max**) set y-axis scaling from **min** to **max**.

yaxis([**min max**]) as above.

yaxis restore automatic scaling for y-axis.

See also

yaxis