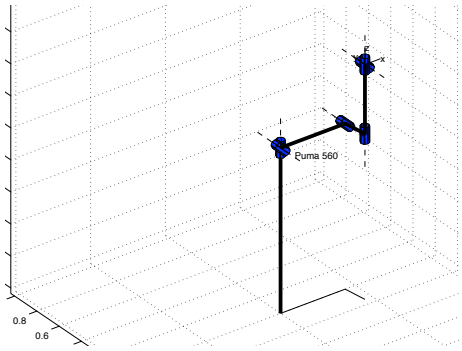
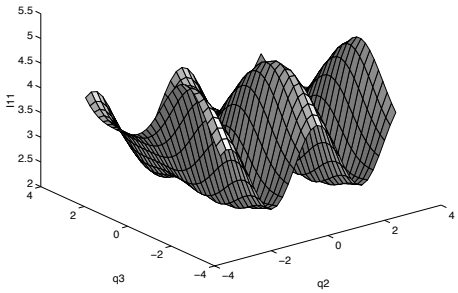
Robotics
TOOLBOX

for MATLAB

(Release 7.1)





Peter I. Corke
Peter.Corke@csiro.au
http://www.cat.csiro.au/cmst/staff/pic/robot

April 2002

Peter I. Corke
CSIRO
Manufacturing Science and Technology
Pullenvale, AUSTRALIA, 4069.
2002
http://www.cat.csiro.au/cmst/staff/pic/robot

©2002 by Peter I. Corke.

1 Preface

1 Introduction

This, the seventh release of the Toolbox, represents nearly a decade of tinkering and a substantial level of maturity. It finally includes many of the features I've been planning to add for some time, particularly MEX files, Simulink support and modified Denavit-Hartenberg support. The previous release has had thousands of downloads and the mailing list has hundreds of subscribers.

The Toolbox provides many functions that are useful in robotics including such things as kinematics, dynamics, and trajectory generation. The Toolbox is useful for simulation as well as analyzing results from experiments with real robots.

The Toolbox is based on a very general method of representing the kinematics and dynamics of serial-link manipulators. These parameters are encapsulated in Matlab objects. Robot objects can be created by the user for any serial-link manipulator and a number of examples are provided for well know robots such as the Puma 560 and the Stanford arm. The Toolbox also provides functions for manipulating and converting between datatypes such as vectors, homogeneous transformations and unit-quaternions which are necessary to represent 3-dimensional position and orientation.

The routines are written in a straightforward manner which allows for easy understanding, perhaps at the expense of computational efficiency. My guide in all of this work has been the book of Paul[1], now out of print, but which I grew up with. 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.

1.1 What's new

This release has some significant new functionality as well as some bug fixes.

- Full support for modified (Craig's) Denavit-Hartenberg notation, forward and inverse kinematics, Jacobians and forward and inverse dynamics.
- Simulink blockset library and demonstrations included, see Section 2
- MEX implementation of recursive Newton-Euler algorithm written in portable C. Speed improvements of at least 1000. Tested on Solaris, Linux and Windows. See Section 1.9.
- Fixed still more bugs and missing files in quaternion code.
- Remove '@' notation from `fdyn` to allow operation under Matlab 5 and 6.
- Fairly major update of documentation to ensure consistency between code, online help and this manual.

All code is now under CVS control which should eliminate many of the versioning problems I had previously due to developing the code across multiple computers.

1.2 Contact

The Toolbox home page is at

http://www.cat.csiro.au/cmst/staff/pic/robot

This page will always list the current released version number as well as bug fixes and new code in between major releases.

A Mailing List is also available, subscriptions details are available off that web page.

1.3 How to obtain the Toolbox

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

http://www.cat.csiro.au/cmst/staff/pic/robot

The files are available in either gzipped tar format (.gz) or zip format (.zip). The web page requests some information from you regarding such as your country, type of organization and application. This is just a means for me to gauge interest and to help convince my bosses (and myself) that this is a worthwhile activity.

The file `robot.pdf` is a comprehensive manual with a tutorial introduction and details of each Toolbox function. A menu-driven demonstration can be invoked by the function `rtdemo`.

1.4 MATLAB version issues

The Toolbox works with MATLAB version 6 and greater and has been tested on a Sun with version 6.

The Toolbox does not function under MATLAB v3.x or v4.x since those versions do not support objects. An older version of the Toolbox, available from the Matlab4 ftp site is workable but lacks some features of this current Toolbox release.

1.5 Acknowledgements

I am grateful for the support of my employer, CSIRO, for supporting me in this activity and providing me with the Matlab tools and web server.

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.

1.6 Support, use in teaching, bug fixes, etc.

I'm always happy to correspond with people who have found genuine bugs or deficiencies in the Toolbox, or who have suggestions about ways to improve its functionality. However I draw the line at providing help for people with their assignments and homework!

Many people are using the Toolbox for teaching and this is something that I would encourage. If you plan to duplicate the documentation for class use then every copy must include the front page.

If you want to cite the Toolbox please use

```
@ARTICLE{Corke96b,
        AUTHOR          = {P.I. Corke},
        JOURNAL         = {IEEE Robotics and Automation Magazine},
        MONTH           = mar,
        NUMBER          = {1},
        PAGES           = {24-32},
        TITLE           = {A Robotics Toolbox for {MATLAB}},
        VOLUME          = {3},
        YEAR            = {1996}
}
```

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

1.7 A note on kinematic conventions

Many people are not aware that there are two quite different forms of Denavit-Hartenberg representation for serial-link manipulator kinematics:

1. Classical as per the original 1955 paper of Denavit and Hartenberg, and used in textbooks such as by Paul[1], Fu etal[2], or Spong and Vidyasagar[3].
2. Modified form as introduced by Craig[4] in his text book.

Both notations represent a joint as 2 translations (A and D) and 2 rotation angles (α and θ). However the expressions for the link transform matrices are quite different. In short, you must know which kinematic convention your Denavit-Hartenberg parameters conform to.

Unfortunately many sources in the literature do not specify this crucial piece of information. Most textbooks cover only one and do not even allude to the existence of the other. These issues are discussed further in Section 3.

The Toolbox has full support for both the classical and modified conventions.

1.8 Creating a new robot definition

Let's take a simple example like the two-link planar manipulator from Spong & Vidyasagar[3] (Figure 3-6, p73) which has the following (standard) Denavit-Hartenberg link parameters

Link	a_i	α_i	d_i	θ_i
1	1	0	0	θ_1^*
2	1	0	0	θ_2^*

where we have set the link lengths to 1. Now we can create a pair of link objects:

```
>> L1=link([0 1 0 0 0], 'standard')

 L1 =

    0.000000      1.000000        0.000000        0.000000        R    (std)

>> L2=link([0 1 0 0 0], 'standard')

 L2 =

    0.000000      1.000000        0.000000        0.000000        R    (std)

>> r=robot({L1 L2})

r =

noname (2 axis, RR)
                grav = [0.00 0.00 9.81]         standard D&H parameters
  alpha           A               theta           D             R/P
0.000000       1.000000        0.000000        0.000000        R    (std)
0.000000       1.000000        0.000000        0.000000        R    (std)

>>
```

The first few lines create link objects, one per robot link. Note the second argument to `link` which specifies that the standard D&H conventions are to be used (this is actually the default). The arguments to the link object can be found from

```
>> help link
    .
    .
        LINK([alpha A theta D sigma], CONVENTION)
    .
    .
```

which shows the order in which the link parameters must be passed (which is different to the column order of the table above). The fifth element of the first argument, `sigma`, is a flag that indicates whether the joint is revolute (`sigma` is zero) or primsmatic (`sigma` is non zero).

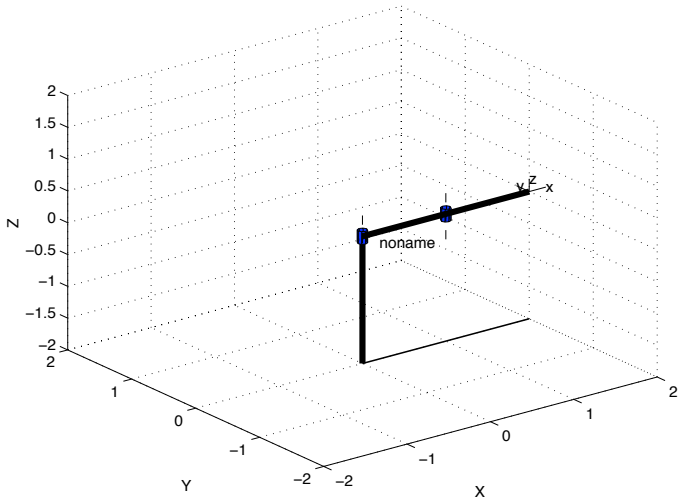


Figure 1: Simple two-link manipulator model.

The link objects are passed as a cell array to the `robot()` function which creates a robot object which is in turn passed to many of the other Toolbox functions.

Note that displays of link data include the kinematic convention in brackets on the far right. `(std)` for standard form, and `(mod)` for modified form.

The robot just created can be displayed graphically by

```
>> plot(r, [0 0])
```

which will create the plot shown in Figure 1.

1.9 Using MEX files

The Robotics Toolbox Release 7 includes portable C source code to generate a MEX file version of the `rne` function.

The MEX file runs upto 500 times faster than the interpretted version `rne.m` and this is critical for calculations involving forward dynamics. The forward dynamics requires the calculation of the manipulator inertia matrix at each integration time step. The Toolbox uses a computationally simple but inefficient method that requires evaluating the `rne` function $n+1$ times, where n is the number of robot axes. For forward dynamics the `rne` is the bottleneck.

The Toolbox stores all robot kinematic and inertial parameters in a `robot` object, but accessing these parameters from a C language MEX file is somewhat cumbersome and must be done on each call. Therefore the speed advantage increases with the number of rows in the `q`, `qd` and `qdd` matrices that are provided. In other words it is better to call `rne` with a trajectory, than for each point on a trajectory.

To build the MEX file:

1. Change directory to the `mex` subdirectory of the Robotics Toolbox.

2. On a Unix system just type `make`. For other platforms follow the Mathworks guidelines. You need to compile and link three files with a command something like `mex frne.c ne.c vmath.c`.

3. If successful you now have a file called `frne.ext` where `ext` is the file extension and depends on the architecture (`mexsol` for Solaris, `mexlx` for Linux).

4. From within Matlab `cd` into this same directory and run the test script

```
>> cd ROBOTDIR/mex
>> check
************************************************************
********************** Puma 560 ****************************
************************************************************
********************** normal case *****************************
DH:  Fast RNE: (c) Peter Corke 2002
Speedup is         17, worst case error is 0.000000
MDH: Speedup is       1565, worst case error is 0.000000
********************** no gravity ******************************
DH:  Speedup is       1501, worst case error is 0.000000
MDH: Speedup is       1509, worst case error is 0.000000
********************** ext force *******************************
DH:  Speedup is       1497, worst case error is 0.000000
MDH: Speedup is        637, worst case error is 0.000000

************************************************************
********************** Stanford arm ****************************
************************************************************
********************** normal case *****************************
DH:  Speedup is       1490, worst case error is 0.000000
MDH: Speedup is       1519, worst case error is 0.000000
********************** no gravity ******************************
DH:  Speedup is       1471, worst case error is 0.000000
MDH: Speedup is       1450, worst case error is 0.000000
********************** ext force *******************************
DH:  Speedup is        417, worst case error is 0.000000
MDH: Speedup is       1458, worst case error is 0.000000
>>
```

   This will run the M-file and MEX-file versions of the `rne` function for various robot models and options with various options. For each case it should report a speedup greater than one, and an error of zero. The results shown above are for a Sparc Ultra 10.

5. Copy the MEX-file `frne.ext` into the Robotics Toolbox main directory with the name `rne.ext`. Thus all future references to `rne` will now invoke the MEX-file instead of the M-file. The first time you run the MEX-file in any Matlab session it will print a one-line identification message.

2 Using the Toolbox with Simulink

2 Introduction

Simulink is the block diagram editing and simulation environment for Matlab. Until its most recent release Simulink has not been able to handle matrix valued signals, and that has made its application to robotics somewhat clumsy. This shortcoming has been rectified with Simulink Release 4. Robot Toolbox Release 7 and higher includes a library of blocks for use in constructing robot kinematic and dynamic models.

To use this new feature it is neccessary to include the Toolbox Simulink block directory in your Matlab path:

```
>> addpath ROBOTDIR/simulink
```

To bring up the block library

```
>> roblocks
```

which will create a display like that shown in Figure 2.

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.

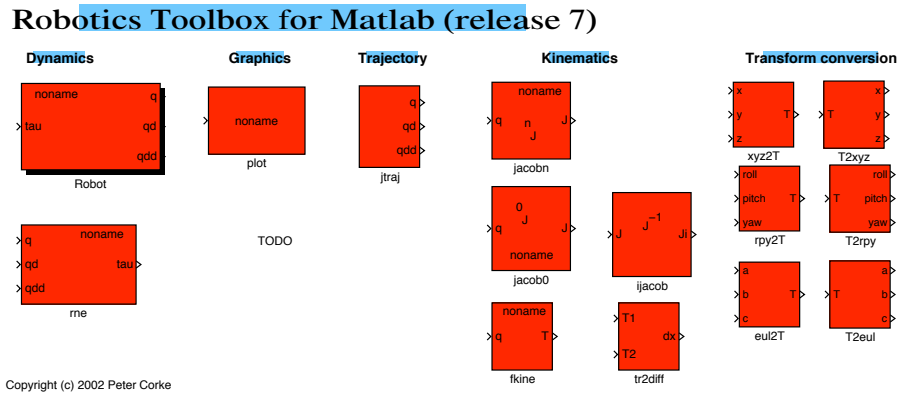


Figure 2: The Robotics Toolbox blockset.

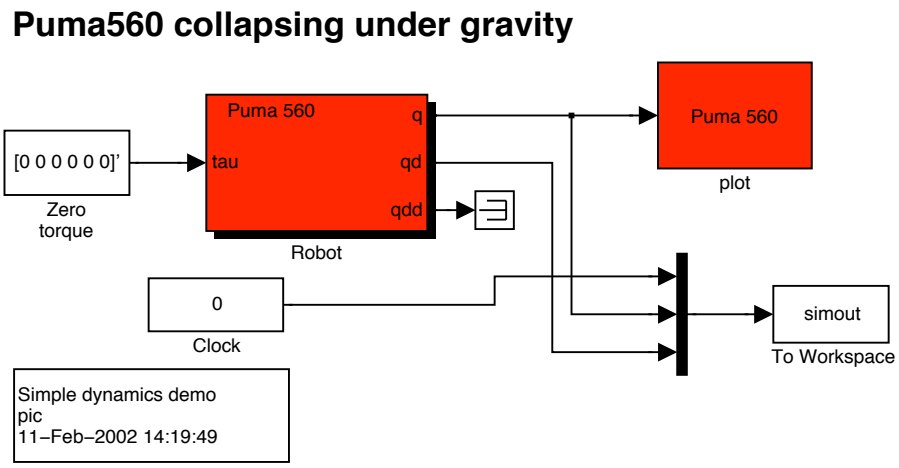


Figure 3: Robotics Toolbox example `demo1`, Puma robot collapsing under gravity.

3 Examples

3.1 Dynamic simulation of Puma 560 robot collapsing under gravity

The Simulink model, `demo1`, is shown in Figure 3, and the two blocks in this model would be familiar to Toolbox users. The `Robot` block is similar to the `fdyn()` function and represents the forward dynamics of the robot, and the `plot` block represents the `plot` function. Note the parameters of the `Robot` block contain the robot object to be simulated and the initial joint angles. The `plot` block has one parameter which is the robot object to be displayed graphically and should be consistent with the robot being simulated. Display options are taken from the `plotbotopt.m` file, see help for `robot/plot` for details.

To run this demo first create a robot object in the workspace,typically by using the `puma560` command, then start the simulation using Simulation/Start option from the model toolbar.

```
>> puma560
>> demo1
```

3.2 Dynamic simulation of a simple robot with flexible transmission

The Simulink model, `demo2`, is shown in Figure 4, and represents a simple 2-link robot with flexible or compliant transmission. The first joint receives a step position demand change at time 1s. The resulting oscillation and dynamic coupling between the two joints can be seen clearly. Note that the drive model comprises spring plus damper, and that the joint position control loops are simply unity negative feedback.

To run this demo first create a 2-link robot object in the workspace,typically by using the `twolink` command, then start the simulation using Simulation/Start option from the model toolbar.

```
>> twolink
>> demo2
```

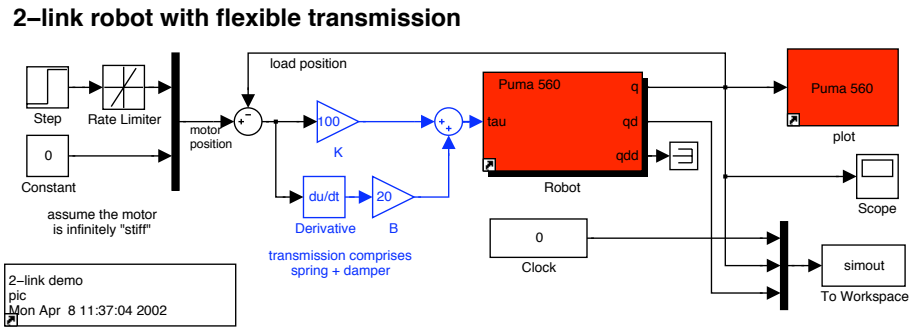


Figure 4: Robotics Toolbox example `demo2`, simple flexible 2-link manipulator.

3.3 Computed torque control

The Simulink model, `demo3`, shown in Figure 5, is for a Puma560 with a computed torque control structure. This is a "classical" dynamic control technique in which the rigid-body dynamic model is inverted to compute the demand torque for the robot based on current joint angles and joint angle rates and demand joint angle acceleration. This model introduces the `rne` block which computes the inverse dynamics using the recursive Newton-Euler algorithm (see `rne` function), and the `jtraj` block which computes a vector quintic polynomial. `jtraj` has parameters which include the initial and final values of the each output element as well as the overall motion time. Initial and final velocity are assumed to be zero.

In practice of course the dynamic model of the robot is not exactly known, we can only invert our best estimate of the rigid-body dynamics. In the simulation we can model this by using the `perturb` function to alter the parameters of the dynamic model used in the `rne` block — note the 'P/' prefix on the model name displayed by that block. This means that the inverse dynamics are computed for a slightly different dynamic model to the robot under control and shows the effect of model error on control performance.

To run this demo first create a robot object in the workspace,typically by using the `puma560` command, then start the simulation using Simulation/Start option from the model toolbar.

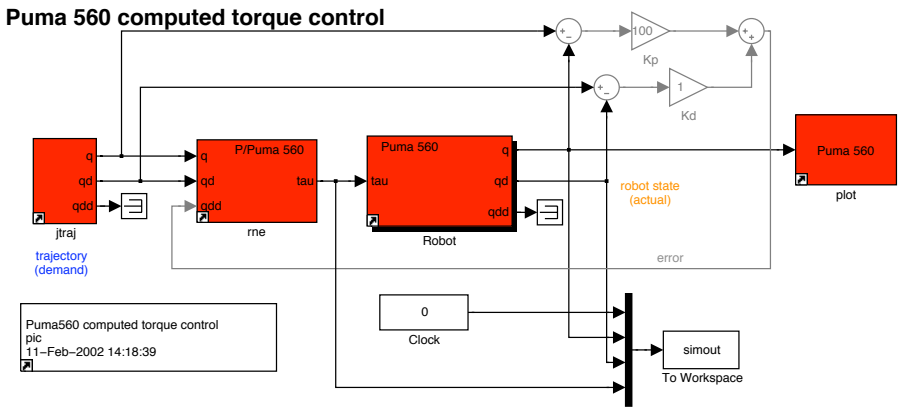


Figure 5: Robotics Toolbox example `demo3`, computed torque control.

```
>> puma560
>> demo3
```

3.4 Torque feedforward control

The Simulink model `demo4` demonstrates torque feedforward control, another "classical" dynamic control technique in which the demanded torque is computed using the `rne` block and added to the error torque computed from position and velocity error. It is instructive to compare the structure of this model with `demo3`. The inverse dynamics are not in the forward path and since the robot configuration changes relatively slowly, they can be computed at a low rate (this is illustrated by the zero-order hold block sampling at 20Hz).

To run this demo first create a robot object in the workspace,typically by using the `puma560` command, then start the simulation using Simulation/Start option from the model toolbar.

```
>> puma560
>> demo4
```

3.5 Cartesian space control

The Simulink model, `demo5`, shown in Figure 6, demonstrates Cartesian space motion control. There are two conventional approaches to this. Firstly, resolve the Cartesian space demand to joint space using inverse kinematics and then perform the control in joint space. The second, used here, is to compute the error in Cartesian space and resolve that to joint space via the inverse Jacobian. This eliminates the need for inverse kinematics within the control loop, and its attendent problems of multiple solutions. It also illustrates some additional Simulink blocks.

This demonstration is for a Puma 560 robot moving the tool in a circle of radius 0.05m centered at the point $(0.5, 0, 0)$. The difference between the Cartesian demand and the current Cartesian pose (in end-effector coordinates) is computed by the `tr2diff` block which produces a differential motion described by a 6-vector. The Jacobian block has as its input the current manipulator joint angles and outputs the Jacobian matrix. Since the differential motion is with respect to the end-effector we use the JacobianN block rather than Jacobian0. We use standard Simulink block to invert the Jacobian and multiply it by

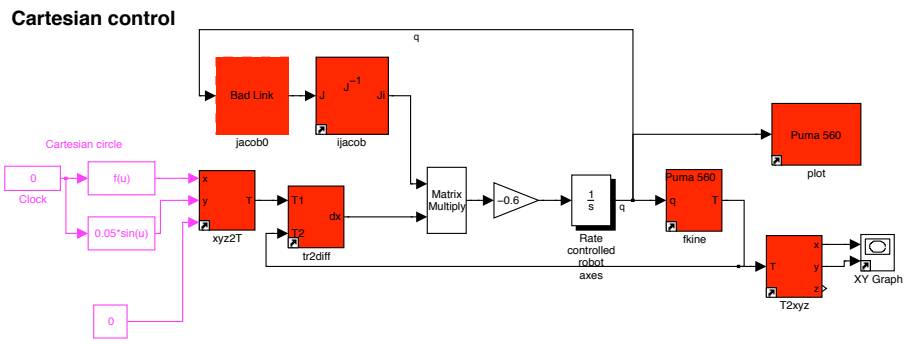


Figure 6: Robotics Toolbox example `demo5`, Cartesian space control.

the differential motion. The result, after application of a simple proportional gain, is the joint space motion required to correct the Cartesian error. The robot is modelled by an integrator as a simple rate control device, or velocity servo.

This example also demonstrates the use of the `fkine` block for forward kinematics and the `T2xyz` block which extracts the translational part of the robot's Cartesian state for plotting on the XY plane.

This demonstration is very similar to the numerical method used to solve the inverse kinematics problem in `ikine`.

To run this demo first create a robot object in the workspace,typically by using the `puma560` command, then start the simulation using Simulation/Start option from the model toolbar.

```
>> puma560
>> demo5
```

3.6 Image-based visual servoing

The Simulink model, `demo6`, shown in Figure 7, demonstrates image-based visual servoing (IBVS)[5]. This is quite a complex example that simulates not only the robot but also a camera and the IBVS algorithm. The camera is assumed to be mounted on the robot's end-effector and this coordinate is passed into the camera block so that the relative position of the target with respect to the camera can be computed. Arguments to the camera block include the 3D coordinates of the target points. The output of the camera is the 2D image plane coordinates of the target points. The target points are used to compute an image Jacobian matrix which is inverted and multiplies the desired motion of the target points on the image plane. The desired motion is simply the difference between the observed target points and the desired point positions. The result is a velocity screw which drives the robot to the desired pose with respect to a square target.

When the simulation starts a new window, the camera view, pops up. We see that initially the square target is off to one side and somewhat oblique. The image plane errors are mapped by an image Jacobian into desired Cartesian rates, and these are futher mapped by a

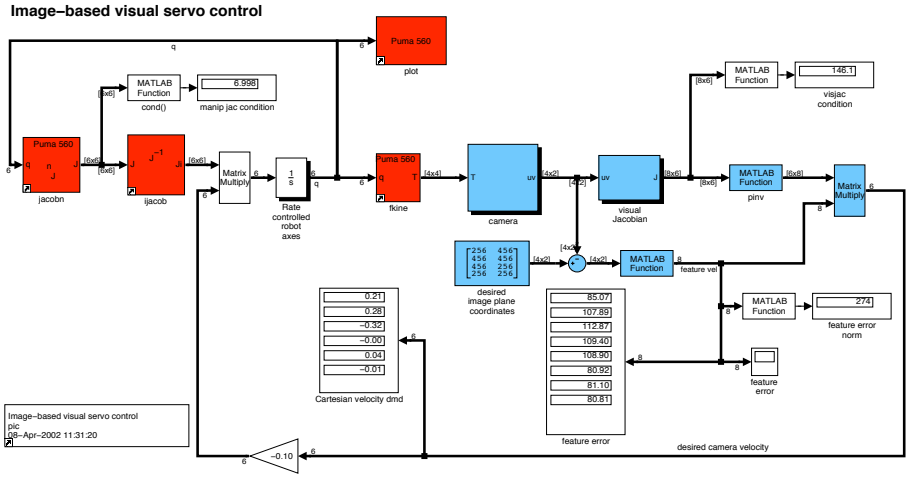


Figure 7: Robotics Toolbox example `demo6`, image-based visual servoing.

manipulator Jacobian into joint rates which are applied to the robot which is again modelled as a rate control device. This closed-loop system is performing a Cartesian positioning task using information from a camera rather than encoders and a kinematic model (the Jacobian is a weak kinematic model). Image-based visual servoing schemes have been found to be extremely robust with respect to errors in the camera model and manipulator Jacobian.

3 Tutorial

3 Manipulator kinematics

Kinematics is the study of motion without regard to the forces which cause it. Within kinematics one studies the position, velocity and acceleration, and all higher order derivatives of the position variables. The kinematics of manipulators involves the study of the geometric and time based properties of the motion, and in particular how the various links move with respect to one another and with time.

Typical robots are *serial-link manipulators* comprising a set of bodies, called *links*, in a chain, connected by *joints*[1]. Each joint has one degree of freedom, either translational or rotational. For a manipulator with n joints numbered from 1 to n, there are $n+1$ links, numbered from 0 to n. Link 0 is the base of the manipulator, generally fixed, and link n carries the end-effector. Joint i connects links i and $i-1$.

A link may be considered as a rigid body defining the relationship between two neighbouring joint axes. A link can be specified by two numbers, the *link length* and *link twist*, which define the relative location of the two axes in space. The link parameters for the first and last links are meaningless, but are arbitrarily chosen to be 0. Joints may be described by two parameters. The *link offset* is the distance from one link to the next along the axis of the joint. The *joint angle* is the rotation of one link with respect to the next about the joint axis.

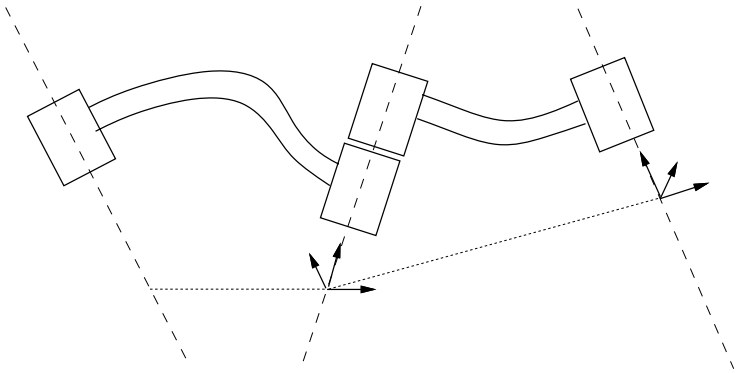
To facilitate describing the location of each link we affix a coordinate frame to it — frame i is attached to link i. Denavit and Hartenberg[6] proposed a matrix method of systematically assigning coordinate systems to each link of an articulated chain. The axis of revolute joint i is aligned with z_{i-1}. The x_{i-1} axis is directed along the normal from z_{i-1} to z_i and for intersecting axes is parallel to $z_{i-1} \times z_i$. The link and joint parameters may be summarized as:

link length	a_i	the offset distance between the z_{i-1} and z_i axes along the x_i axis;
link twist	α_i	the angle from the z_{i-1} axis to the z_i axis about the x_i axis;
link offset	d_i	the distance from the origin of frame $i-1$ to the x_i axis along the z_{i-1} axis;
joint angle	θ_i	the angle between the x_{i-1} and x_i axes about the z_{i-1} axis.

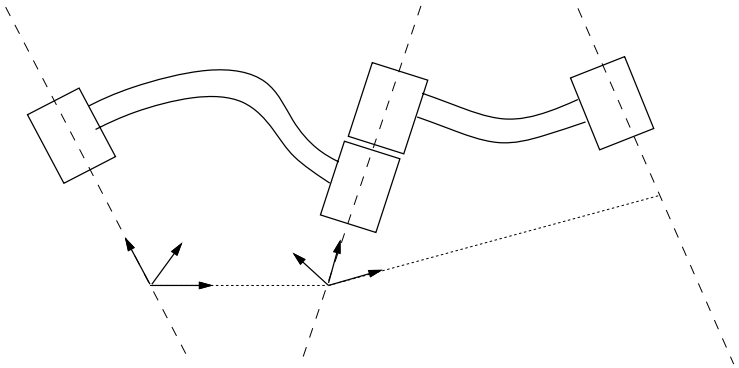
For a revolute axis θ_i is the joint variable and d_i is constant, while for a prismatic joint d_i is variable, and θ_i is constant. In many of the formulations that follow we use generalized coordinates, q_i, where

$$q_i = \begin{cases} \theta_i & \text{for a revolute joint} \\ d_i & \text{for a prismatic joint} \end{cases}$$

[1] Parallel link and serial/parallel hybrid structures are possible, though much less common in industrial manipulators.



(a) Standard form



(b) Modified form

Figure 8: Different forms of Denavit-Hartenberg notation.

and generalized forces

$$Q_i = \begin{cases} \tau_i & \text{for a revolute joint} \\ f_i & \text{for a prismatic joint} \end{cases}$$

The Denavit-Hartenberg (DH) representation results in a 4x4 homogeneous transformation matrix

$$ {}^{i-1}\mathbf{A}_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

representing each link's coordinate frame with respect to the previous link's coordinate system; that is

$$ {}^{0}\mathbf{T}_i = {}^{0}\mathbf{T}_{i-1}\ {}^{i-1}\mathbf{A}_i \tag{2}$$

where ${}^{0}\mathbf{T}_i$ is the homogeneous transformation describing the pose of coordinate frame i with respect to the world coordinate system 0.

Two differing methodologies have been established for assigning coordinate frames, each of which allows some freedom in the actual coordinate frame attachment:

1. Frame i has its origin along the axis of joint $i+1$, as described by Paul[1] and Lee[2, 7].

2. Frame i has its origin along the axis of joint i, and is frequently referred to as 'modified Denavit-Hartenberg' (MDH) form[8]. This form is commonly used in literature dealing with manipulator dynamics. The link transform matrix for this form differs from (1).

Figure 8 shows the notational differences between the two forms. Note that a_i is always the length of link i, but is the displacement between the origins of frame i and frame $i+1$ in one convention, and frame $i-1$ and frame i in the other[2]. The Toolbox provides kinematic functions for both of these conventions — those for modified DH parameters are prefixed by 'm'.

3.1 Forward and inverse kinematics

For an n-axis rigid-link manipulator, the *forward kinematic solution* gives the coordinate frame, or pose, of the last link. It is obtained by repeated application of (2)

$$\begin{aligned} {}^0\mathbf{T}_n &= {}^0\mathbf{A}_1\,{}^1\mathbf{A}_2 \cdots {}^{n-1}\mathbf{A}_n & (3)\\ &= K(\underline{q}) & (4) \end{aligned}$$

which is the product of the coordinate frame transform matrices for each link. The pose of the end-effector has 6 degrees of freedom in Cartesian space, 3 in translation and 3 in rotation, so robot manipulators commonly have 6 joints or degrees of freedom to allow arbitrary end-effector pose. The overall manipulator transform ${}^0\mathbf{T}_n$ is frequently written as $\mathbf{T}_n$, or $\mathbf{T}_6$ for a 6-axis robot. The forward kinematic solution may be computed for any manipulator, irrespective of the number of joints or kinematic structure.

Of more use in manipulator path planning is the *inverse kinematic solution*

$$\underline{q} = K^{-1}(\mathbf{T}) \tag{5}$$

which gives the joint angles required to reach the specified end-effector position. In general this solution is non-unique, and for some classes of manipulator no closed-form solution exists. If the manipulator has more than 6 joints it is said to be *redundant* and the solution for joint angles is under-determined. If no solution can be determined for a particular manipulator pose that configuration is said to be *singular*. The singularity may be due to an alignment of axes reducing the effective degrees of freedom, or the point $\mathbf{T}$ being out of reach.

The manipulator Jacobian matrix, $\mathbf{J}_\theta$, transforms velocities in joint space to velocities of the end-effector in Cartesian space. For an n-axis manipulator the end-effector Cartesian velocity is

$$\begin{aligned} {}^0\dot{\underline{x}}_n &= {}^0\mathbf{J}_\theta \dot{\underline{q}} & (6)\\ {}^{t_n}\dot{\underline{x}}_n &= {}^{t_n}\mathbf{J}_\theta \dot{\underline{q}} & (7) \end{aligned}$$

in base or end-effector coordinates respectively and where $\underline{x}$ is the Cartesian velocity represented by a 6-vector. For a 6-axis manipulator the Jacobian is square and provided it is not singular can be inverted to solve for joint rates in terms of end-effector Cartesian rates. The Jacobian will not be invertible at a kinematic singularity, and in practice will be poorly

[2]Many papers when tabulating the 'modified' kinematic parameters of manipulators list q_{-1} and α_{i-1} not a_i and α_i.

conditioned in the vicinity of the singularity, resulting in high joint rates. A control scheme based on Cartesian rate control

$$\dot{\underline{q}} = {}^0\mathbf{J}_\theta^{-1}\,{}^0\dot{\underline{x}}_n \tag{8}$$

was proposed by Whitney[9] and is known as *resolved rate motion control*. For two frames A and B related by ${}^A\mathbf{T}_B = [\underline{n}\ \underline{o}\ \underline{a}\ \underline{p}]$ the Cartesian velocity in frame A may be transformed to frame B by

$${}^B\dot{\underline{x}} = {}^B\mathbf{J}_A\,{}^A\dot{\underline{x}} \tag{9}$$

where the Jacobian is given by Paul[10] as

$${}^B\mathbf{J}_A = f({}^A\mathbf{T}_B) = \begin{bmatrix} [\underline{n}\ \underline{o}\ \underline{a}]^T & [\underline{p}\times\underline{n}\ \underline{p}\times\underline{o}\ \underline{p}\times\underline{a}]^T \\ 0 & [\underline{n}\ \underline{o}\ \underline{a}]^T \end{bmatrix} \tag{10}$$

4 Manipulator rigid-body dynamics

Manipulator dynamics is concerned with the equations of motion, the way in which the manipulator moves in response to torques applied by the actuators, or external forces. The history and mathematics of the dynamics of serial-link manipulators is well covered by Paul[1] and Hollerbach[11]. There are two problems related to manipulator dynamics that are important to solve:

- *inverse dynamics* in which the manipulator's equations of motion are solved for given motion to determine the generalized forces, discussed further in Section 4.1, and
- *direct dynamics* in which the equations of motion are integrated to determine the generalized coordinate response to applied generalized forces discussed further in Section 4.2.

The equations of motion for an n-axis manipulator are given by

$$\underline{Q} = \mathbf{M}(\underline{q})\ddot{\underline{q}} + \mathbf{C}(\underline{q},\dot{\underline{q}})\dot{\underline{q}} + \mathbf{F}(\dot{\underline{q}}) + \mathbf{G}(\underline{q}) \tag{11}$$

where

- $\underline{q}$ is the vector of generalized joint coordinates describing the pose of the manipulator
- $\dot{\underline{q}}$ is the vector of joint velocities;
- $\ddot{\underline{q}}$ is the vector of joint accelerations
- $\mathbf{M}$ is the symmetric joint-space inertia matrix, or manipulator inertia tensor
- $\mathbf{C}$ describes Coriolis and centripetal effects — Centripetal torques are proportional to $\dot{q}_i^2$, while the Coriolis torques are proportional to $\dot{q}_i\dot{q}_j$
- $\mathbf{F}$ describes viscous and Coulomb friction and is not generally considered part of the rigid-body dynamics
- $\mathbf{G}$ is the gravity loading
- $\underline{Q}$ is the vector of generalized forces associated with the generalized coordinates $\underline{q}$.

The equations may be derived via a number of techniques, including Lagrangian (energy based), Newton-Euler, d'Alembert[2, 12] or Kane's[13] method. The earliest reported work was by Uicker[14] and Kahn[15] using the Lagrangian approach. Due to the enormous computational cost, $O(n^4)$, of this approach it was not possible to compute manipulator torque for real-time control. To achieve real-time performance many approaches were suggested, including table lookup[16] and approximation[17, 18]. The most common approximation was to ignore the velocity-dependent term $\mathbf{C}$, since accurate positioning and high speed motion are exclusive in typical robot applications.

Method	Multiplications	Additions	For N=6	
			Multiply	Add
Lagrangian[22]	$32\frac{1}{2}n^4 + 86\frac{5}{12}n^3 + 171\frac{1}{4}n^2 + 53\frac{1}{3}n - 128$	$25n^4 + 66\frac{1}{3}n^3 + 129\frac{1}{2}n^2 + 42\frac{1}{3}n - 96$	66,271	51,548
Recursive NE[22]	$150n - 48$	$131n - 48$	852	738
Kane[13]			646	394
Simplified RNE[25]			224	174

Table 1: Comparison of computational costs for inverse dynamics from various sources. The last entry is achieved by symbolic simplification using the software package ARM.

Orin et al.[19] proposed an alternative approach based on the Newton-Euler (NE) equations of rigid-body motion applied to each link. Armstrong[20] then showed how recursion might be applied resulting in $O(n)$ complexity. Luh et al.[21] provided a recursive formulation of the Newton-Euler equations with linear and angular velocities referred to link coordinate frames. They suggested a time improvement from 7.9s for the Lagrangian formulation to 4.5ms, and thus it became practical to implement 'on-line'. Hollerbach[22] showed how recursion could be applied to the Lagrangian form, and reduced the computation to within a factor of 3 of the recursive NE. Silver[23] showed the equivalence of the recursive Lagrangian and Newton-Euler forms, and that the difference in efficiency is due to the representation of angular velocity.

"Kane's equations" [13] provide another methodology for deriving the equations of motion for a specific manipulator. A number of 'Z' variables are introduced, which while not necessarily of physical significance, lead to a dynamics formulation with low computational burden. Wampler[24] discusses the computational costs of Kane's method in some detail.

The NE and Lagrange forms can be written generally in terms of the Denavit-Hartenberg parameters — however the specific formulations, such as Kane's, can have lower computational cost for the specific manipulator. Whilst the recursive forms are computationally more efficient, the non-recursive forms compute the individual dynamic terms (**M**, **C** and **G**) directly. A comparison of computation costs is given in Table 1.

4.1 Recursive Newton-Euler formulation

The recursive Newton-Euler (RNE) formulation[21] computes the inverse manipulator dynamics, that is, the joint torques required for a given set of joint angles, velocities and accelerations. The forward recursion propagates kinematic information — such as angular velocities, angular accelerations, linear accelerations — from the base reference frame (inertial frame) to the end-effector. The backward recursion propagates the forces and moments exerted on each link from the end-effector of the manipulator to the base reference frame[3]. Figure 9 shows the variables involved in the computation for one link.

The notation of Hollerbach[22] and Walker and Orin [26] will be used in which the left superscript indicates the reference coordinate frame for the variable. The notation of Luh et al.[21] and later Lee[7, 2] is considerably less clear.

[3]It should be noted that using MDH notation with its different axis assignment conventions the Newton Euler formulation is expressed differently[8].

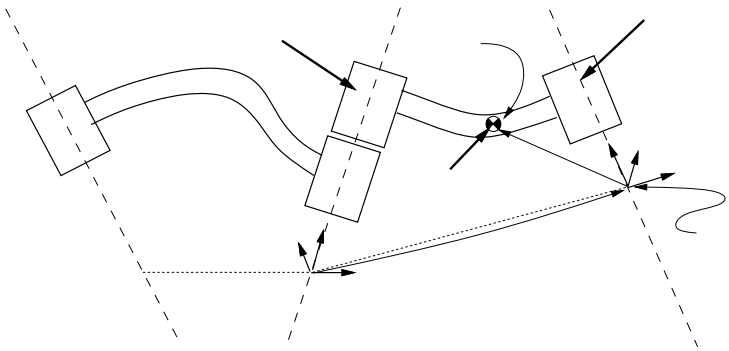


Figure 9: Notation used for inverse dynamics, based on standard Denavit-Hartenberg notation.

Outward recursion, $1 \leq i \leq n$**.**

If axis $i+1$ is rotational

$$
\begin{aligned}
{}^{i+1}\underline{\omega}_{i+1} &= {}^{i+1}\mathbf{R}_i\left({}^{i}\underline{\omega}_i + \underline{z}_0\dot{\underline{q}}_{i+1}\right) && (12)\\
{}^{i+1}\dot{\underline{\omega}}_{i+1} &= {}^{i+1}\mathbf{R}_i\left\{{}^{i}\dot{\underline{\omega}}_i + \underline{z}_0\ddot{\underline{q}}_{i+1} + {}^{i}\underline{\omega}_i \times \left(\underline{z}_0\dot{\underline{q}}_{i+1}\right)\right\} && (13)\\
{}^{i+1}\underline{v}_{i+1} &= {}^{i+1}\underline{\omega}_{i+1} \times {}^{i+1}\underline{p}^*_{i+1} + {}^{i+1}\mathbf{R}_i\,{}^{i}\underline{v}_i && (14)\\
{}^{i+1}\dot{\underline{v}}_{i+1} &= {}^{i+1}\dot{\underline{\omega}}_{i+1} \times {}^{i+1}\underline{p}^*_{i+1} + {}^{i+1}\underline{\omega}_{i+1} \times \left\{{}^{i+1}\underline{\omega}_{i+1} \times {}^{i+1}\underline{p}^*_{i+1}\right\} + {}^{i+1}\mathbf{R}_i\,{}^{i}\dot{\underline{v}}_i && (15)
\end{aligned}
$$

If axis $i+1$ is translational

$$
\begin{aligned}
{}^{i+1}\underline{\omega}_{i+1} &= {}^{i+1}\mathbf{R}_i\,{}^{i}\underline{\omega}_i && (16)\\
{}^{i+1}\dot{\underline{\omega}}_{i+1} &= {}^{i+1}\mathbf{R}_i\,{}^{i}\dot{\underline{\omega}}_i && (17)\\
{}^{i+1}\underline{v}_{i+1} &= {}^{i+1}\mathbf{R}_i\left(\underline{z}_0\dot{\underline{q}}_{i+1} + {}^{i}\underline{v}_i\right) + {}^{i+1}\underline{\omega}_{i+1} \times {}^{i+1}\underline{p}^*_{i+1} && (18)\\
{}^{i+1}\dot{\underline{v}}_{i+1} &= {}^{i+1}\mathbf{R}_i\left(\underline{z}_0\ddot{\underline{q}}_{i+1} + {}^{i}\dot{\underline{v}}_i\right) + {}^{i+1}\dot{\underline{\omega}}_{i+1} \times {}^{i+1}\underline{p}^*_{i+1} + 2\,{}^{i+1}\underline{\omega}_{i+1} \times \left({}^{i+1}\mathbf{R}_i\underline{z}_0\dot{\underline{q}}_{i+1}\right)\\
&\quad + {}^{i+1}\underline{\omega}_{i+1} \times \left({}^{i+1}\underline{\omega}_{i+1} \times {}^{i+1}\underline{p}^*_{i+1}\right) && (19)
\end{aligned}
$$

$$
\begin{aligned}
{}^{i}\dot{\underline{v}}_i &= {}^{i}\dot{\underline{\omega}}_i \times \underline{s}_i + {}^{i}\underline{\omega}_i \times \left\{{}^{i}\underline{\omega}_i \times \underline{s}_i\right\} + {}^{i}\dot{\underline{v}}_i && (20)\\
{}^{i}\underline{F}_i &= m_i\,{}^{i}\dot{\underline{v}}_i && (21)\\
{}^{i}\underline{N}_i &= \mathbf{J}_i\,{}^{i}\dot{\underline{\omega}}_i + {}^{i}\underline{\omega}_i \times \left(\mathbf{J}_i\,{}^{i}\underline{\omega}_i\right) && (22)
\end{aligned}
$$

Inward recursion, $n \geq i \geq 1$**.**

$$
\begin{aligned}
{}^{i}\underline{f}_i &= {}^{i}\mathbf{R}_{i+1}\,{}^{i+1}\underline{f}_{i+1} + {}^{i}\underline{F}_i && (23)\\
{}^{i}\underline{n}_i &= {}^{i}\mathbf{R}_{i+1}\left\{{}^{i+1}\underline{n}_{i+1} + \left({}^{i+1}\mathbf{R}_i\,{}^{i}\underline{p}^*_i\right) \times {}^{ii+1}\underline{f}_{i+1}\right\} + \left({}^{i}\underline{p}^*_i + \underline{s}_i\right) \times {}^{i}\underline{F}_i + {}^{i}\underline{N}_i && (24)\\
\underline{Q}_i &= \begin{cases} \left({}^{i}\underline{n}_i\right)^T\left({}^{i}\mathbf{R}_{i+1}\underline{z}_0\right) & \text{if link } i+1 \text{ is rotational} \\ \left({}^{i}\underline{f}_i\right)^T\left({}^{i}\mathbf{R}_{i+1}\underline{z}_0\right) & \text{if link } i+1 \text{ is translational} \end{cases} && (25)
\end{aligned}
$$

where

i is the link index, in the range 1 to n

$\mathbf{J}_i$ is the moment of inertia of link i about its COM

$\underline{s}_i$ is the position vector of the COM of link i with respect to frame i

$\underline{\omega}_i$ is the angular velocity of link i

$\underline{\dot{\omega}}_i$ is the angular acceleration of link i

$\underline{v}_i$ is the linear velocity of frame i

$\underline{\dot{v}}_i$ is the linear acceleration of frame i

$\underline{\overline{v}}_i$ is the linear velocity of the COM of link i

$\underline{\dot{\overline{v}}}_i$ is the linear acceleration of the COM of link i

$\underline{n}_i$ is the moment exerted on link i by link $i-1$

$\underline{f}_i$ is the force exerted on link i by link $i-1$

$\underline{N}_i$ is the total moment at the COM of link i

$\underline{F}_i$ is the total force at the COM of link i

$\underline{Q}_i$ is the force or torque exerted by the actuator at joint i

${}^{i-1}\mathbf{R}_i$ is the orthonormal rotation matrix defining frame i orientation with respect to frame $i-1$. It is the upper 3×3 portion of the link transform matrix given in (1).

$$ {}^{i-1}\mathbf{R}_i = \begin{bmatrix} \cos\theta_i & -\cos\alpha_i \sin\theta_i & \sin\alpha_i \sin\theta_i \\ \sin\theta_i & \cos\alpha_i \cos\theta_i & -\sin\alpha_i \cos\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i \end{bmatrix} \tag{26} $$

$$ {}^{i}\mathbf{R}_{i-1} = ({}^{i-1}\mathbf{R}_i)^{-1} = ({}^{i-1}\mathbf{R}_i)^T \tag{27} $$

${}^{i}\underline{p}_i^*$ is the displacement from the origin of frame $i-1$ to frame i with respect to frame i.

$$ {}^{i}\underline{p}_i^* = \begin{bmatrix} a_i \\ d_i \sin\alpha_i \\ d_i \cos\alpha_i \end{bmatrix} \tag{28} $$

It is the negative translational part of $({}^{i-1}\mathbf{A}_i)^{-1}$.

$\underline{z}_0$ is a unit vector in Z direction, $\underline{z}_0 = [0\ 0\ 1]$

Note that the COM linear velocity given by equation (14) or (18) does not need to be computed since no other expression depends upon it. Boundary conditions are used to introduce the effect of gravity by setting the acceleration of the base link

$$ \dot{v}_0 = -\underline{g} \tag{29} $$

where $\underline{g}$ is the gravity vector in the reference coordinate frame, generally acting in the negative Z direction, downward. Base velocity is generally zero

$$ v_0 = 0 \tag{30} $$

$$ \omega_0 = 0 \tag{31} $$

$$ \dot{\omega}_0 = 0 \tag{32} $$

At this stage the Toolbox only provides an implementation of this algorithm using the standard Denavit-Hartenberg conventions.

4.2 Direct dynamics

Equation (11) may be used to compute the so-called inverse dynamics, that is, actuator torque as a function of manipulator state and is useful for on-line control. For simulation

the direct, integral or *forward dynamic* formulation is required giving joint motion in terms of input torques.

Walker and Orin[26] describe several methods for computing the forward dynamics, and all make use of an existing inverse dynamics solution. Using the RNE algorithm for inverse dynamics, the computational complexity of the forward dynamics using 'Method 1' is $O(n^3)$ for an n-axis manipulator. Their other methods are increasingly more sophisticated but reduce the computational cost, though still $O(n^3)$. Featherstone[27] has described the "articulated-body method" for $O(n)$ computation of forward dynamics, however for $n < 9$ it is more expensive than the approach of Walker and Orin. Another $O(n)$ approach for forward dynamics has been described by Lathrop[28].

4.3 Rigid-body inertial parameters

Accurate model-based dynamic control of a manipulator requires knowledge of the rigid-body inertial parameters. Each link has ten independent inertial parameters:

- link mass, m_i;
- three first moments, which may be expressed as the COM location, $\underline{s}_i$, with respect to some datum on the link or as a moment $\underline{S}_i = m_i \underline{s}_i$;
- six second moments, which represent the inertia of the link about a given axis, typically through the COM. The second moments may be expressed in matrix or tensor form as

$$\mathbf{J} = \begin{bmatrix} J_{xx} & J_{xy} & J_{xz} \\ J_{xy} & J_{yy} & J_{yz} \\ J_{xz} & J_{yz} & J_{zz} \end{bmatrix} \tag{33}$$

  where the diagonal elements are the *moments of inertia*, and the off-diagonals are *products of inertia*. Only six of these nine elements are unique: three moments and three products of inertia.

  For any point in a rigid-body there is one set of axes known as the *principal axes of inertia* for which the off-diagonal terms, or products, are zero. These axes are given by the eigenvectors of the inertia matrix (33) and the eigenvalues are the principal moments of inertia. Frequently the products of inertia of the robot links are zero due to symmetry.

A 6-axis manipulator rigid-body dynamic model thus entails 60 inertial parameters. There may be additional parameters per joint due to friction and motor armature inertia. Clearly, establishing numeric values for this number of parameters is a difficult task. Many parameters cannot be measured without dismantling the robot and performing careful experiments, though this approach was used by Armstrong et al.[29]. Most parameters could be derived from CAD models of the robots, but this information is often considered proprietary and not made available to researchers.

References

[1] R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. Cambridge, Massachusetts: MIT Press, 1981.

[2] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee, *Robotics. Control, Sensing, Vision and Intelligence*. McGraw-Hill, 1987.

[3] M. Spong and M. Vidyasagar, *Robot Dynamics and Control*. John Wiley and Sons, 1989.

[4] J. J. Craig, *Introduction to Robotics*. Addison Wesley, 1986.

[5] S. Hutchinson, G. Hager, and P. Corke, "A tutorial on visual servo control," *IEEE Transactions on Robotics and Automation*, vol. 12, pp. 651–670, Oct. 1996.

[6] R. S. Hartenberg and J. Denavit, "A kinematic notation for lower pair mechanisms based on matrices," *Journal of Applied Mechanics*, vol. 77, pp. 215–221, June 1955.

[7] C. S. G. Lee, "Robot arm kinematics, dynamics and control," *IEEE Computer*, vol. 15, pp. 62–80, Dec. 1982.

[8] J. J. Craig, *Introduction to Robotics*. Addison Wesley, second ed., 1989.

[9] D. Whitney, "The mathematics of coordinated control of prosthetic arms and manipulators," *ASME Journal of Dynamic Systems, Measurement and Control*, vol. 20, no. 4, pp. 303–309, 1972.

[10] R. P. Paul, B. Shimano, and G. E. Mayer, "Kinematic control equations for simple manipulators," *IEEE Trans. Syst. Man Cybern.*, vol. 11, pp. 449–455, June 1981.

[11] J. M. Hollerbach, "Dynamics," in *Robot Motion - Planning and Control* (M. Brady, J. M. Hollerbach, T. L. Johnson, T. Lozano-Perez, and M. T. Mason, eds.), pp. 51–71, MIT, 1982.

[12] C. S. G. Lee, B. Lee, and R. Nigham, "Development of the generalized D'Alembert equations of motion for mechanical manipulators," in *Proc. 22nd CDC*, (San Antonio, Texas), pp. 1205–1210, 1983.

[13] T. Kane and D. Levinson, "The use of Kane's dynamical equations in robotics," *Int. J. Robot. Res.*, vol. 2, pp. 3–21, Fall 1983.

[14] J. Uicker, *On the Dynamic Analysis of Spatial Linkages Using 4 by 4 Matrices*. PhD thesis, Dept. Mechanical Engineering and Astronautical Sciences, NorthWestern University, 1965.

[15] M. Kahn, "The near-minimum time control of open-loop articulated kinematic linkages," Tech. Rep. AIM-106, Stanford University, 1969.

[16] M. H. Raibert and B. K. P. Horn, "Manipulator control using the configuration space method," *The Industrial Robot*, pp. 69–73, June 1978.

[17] A. Bejczy, "Robot arm dynamics and control," Tech. Rep. NASA-CR-136935, NASA JPL, Feb. 1974.

[18] R. Paul, "Modelling, trajectory calculation and servoing of a computer controlled arm," Tech. Rep. AIM-177, Stanford University, Artificial Intelligence Laboratory, 1972.

[19] D. Orin, R. McGhee, M. Vukobratovic, and G. Hartoch, "Kinematics and kinetic analysis of open-chain linkages utilizing Newton-Euler methods," *Mathematical Biosciences. An International Journal*, vol. 43, pp. 107–130, Feb. 1979.

[20] W. Armstrong, "Recursive solution to the equations of motion of an n-link manipulator," in *Proc. 5th World Congress on Theory of Machines and Mechanisms*, (Montreal), pp. 1343–1346, July 1979.

[21] J. Y. S. Luh, M. W. Walker, and R. P. C. Paul, "On-line computational scheme for mechanical manipulators," *ASME Journal of Dynamic Systems, Measurement and Control*, vol. 102, pp. 69–76, 1980.

[22] J. Hollerbach, "A recursive Lagrangian formulation of manipulator dynamics and a comparative study of dynamics formulation complexity," *IEEE Trans. Syst. Man Cybern.*, vol. SMC-10, pp. 730–736, Nov. 1980.

[23] W. M. Silver, "On the equivalance of Lagrangian and Newton-Euler dynamics for manipulators," *Int. J. Robot. Res.*, vol. 1, pp. 60–70, Summer 1982.

[24] C. Wampler, *Computer Methods in Manipulator Kinematics, Dynamics, and Control: a Comparative Study.* PhD thesis, Stanford University, 1985.

[25] J. J. Murray, *Computational Robot Dynamics.* PhD thesis, Carnegie-Mellon University, 1984.

[26] M. W. Walker and D. E. Orin, "Efficient dynamic computer simulation of robotic mechanisms," *ASME Journal of Dynamic Systems, Measurement and Control*, vol. 104, pp. 205–211, 1982.

[27] R. Featherstone, *Robot Dynamics Algorithms.* Kluwer Academic Publishers, 1987.

[28] R. Lathrop, "Constrained (closed-loop) robot simulation by local constraint propogation.," in *Proc. IEEE Int. Conf. Robotics and Automation*, pp. 689–694, 1986.

[29] B. Armstrong, O. Khatib, and J. Burdick, "The explicit dynamic model and inertial parameters of the Puma 560 arm," in *Proc. IEEE Int. Conf. Robotics and Automation*, vol. 1, (Washington, USA), pp. 510–18, 1986.

2 Reference

For an n-axis manipulator the following matrix naming and dimensional conventions apply.

Symbol	Dimensions	Description
`l`	**link**	manipulator link object
`q`	$1 \times n$	joint coordinate vector
`q`	$m \times n$	m-point joint coordinate trajectory
`qd`	$1 \times n$	joint velocity vector
`qd`	$m \times n$	m-point joint velocity trajectory
`qdd`	$1 \times n$	joint acceleration vector
`qdd`	$m \times n$	m-point joint acceleration trajectory
`robot`	**robot**	robot object
`T`	4×4	homogeneous transform
`T`	$4 \times 4 \times m$	m-point homogeneous transform trajectory
`Q`	**quaternion**	unit-quaternion object
`M`	1×6	vector with elements of 0 or 1 corresponding to Cartesian DOF along X, Y, Z and around X, Y, Z. 1 if that Cartesian DOF belongs to the task space, else 0.
`v`	3×1	Cartesian vector
`t`	$m \times 1$	time vector
`d`	6×1	differential motion vector

Object names are shown in bold typeface.

A trajectory is represented by a matrix in which each row corresponds to one of m time steps. For a joint coordinate, velocity or acceleration trajectory the columns correspond to the robot axes. For homogeneous transform trajectories we use 3-dimensional matrices where the last index corresponds to the time step.

Units

All angles are in radians. The choice of all other units is up to the user, and this choice will flow on to the units in which homogeneous transforms, Jacobians, inertias and torques are represented.

Homogeneous Transforms	
`eul2tr`	Euler angle to homogeneous transform
`oa2tr`	orientation and approach vector to homogeneous transform
`rotvec`	homogeneous transform for rotation about arbitrary vector
`rotx`	homogeneous transform for rotation about X-axis
`roty`	homogeneous transform for rotation about Y-axis
`rotz`	homogeneous transform for rotation about Z-axis
`rpy2tr`	Roll/pitch/yaw angles to homogeneous transform
`tr2eul`	homogeneous transform to Euler angles
`tr2rot`	homogeneous transform to rotation submatrix
`tr2rpy`	homogeneous transform to roll/pitch/yaw angles
`transl`	set or extract the translational component of a homogeneous transform
`trnorm`	normalize a homogeneous transform

Trajectory Generation	
`ctraj`	Cartesian trajectory
`jtraj`	joint space trajectory
`trinterp`	interpolate homogeneous transforms

Quaternions	
`/`	divide quaternion by quaternion or scalar
`*`	multiply quaternion by a quaternion or vector
`inv`	invert a quaternion
`norm`	norm of a quaternion
`plot`	display a quaternion as a 3D rotation
`q2tr`	quaternion to homogeneous transform
`quaternion`	construct a quaternion
`qinterp`	interpolate quaternions
`unit`	unitize a quaternion

Manipulator Models	
`link`	construct a robot link object
`nofriction`	remove friction from a robot object
`perturb`	randomly modify some dynamic parameters
`puma560`	Puma 560 data
`puma560akb`	Puma 560 data (modified Denavit-Hartenberg)
`robot`	construct a robot object
`showlink`	show link/robot data in detail
`stanford`	Stanford arm data
`twolink`	simple 2-link example

Kinematics	
`diff2tr`	differential motion vector to transform
`fkine`	compute forward kinematics
`ftrans`	transform force/moment
`ikine`	compute inverse kinematics
`ikine560`	compute inverse kinematics for Puma 560 like arm
`jacob0`	compute Jacobian in base coordinate frame
`jacobn`	compute Jacobian in end-effector coordinate frame
`tr2diff`	homogeneous transform to differential motion vector
`tr2jac`	homogeneous transform to Jacobian

Graphics	
`drivebot`	drive a graphical robot
`plot`	plot/animate robot

Dynamics	
`accel`	compute forward dynamics
`cinertia`	compute Cartesian manipulator inertia matrix
`coriolis`	compute centripetal/coriolis torque
`fdyn`	forward dynamics (motion given forces)
`friction`	joint friction
`gravload`	compute gravity loading
`inertia`	compute manipulator inertia matrix
`itorque`	compute inertia torque
`rne`	inverse dynamics (forces given motion)

Other	
`ishomog`	test if matrix is 4×4
`maniplty`	compute manipulability
`rtdemo`	toolbox demonstration
`unit`	unitize a vector

accel

Purpose

Compute manipulator forward dynamics

Synopsis

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

Description

Returns a vector of joint accelerations that result from applying the actuator `torque` to the manipulator `robot` with joint coordinates `q` and velocities `qd`.

Algorithm

Uses the method 1 of Walker and Orin to compute the forward dynamics. This form is useful for simulation of manipulator dynamics, in conjunction with a numerical integration function.

See Also

rne, robot, fdyn, ode45

References

M. W. Walker and D. E. Orin. Efficient dynamic computer simulation of robotic mechanisms. *ASME Journal of Dynamic Systems, Measurement and Control*, 104:205–211, 1982.

cinertia

Purpose

Compute the Cartesian (operational space) manipulator inertia matrix

Synopsis

```
M = cinertia(robot, q)
```

Description

`cinertia` computes the Cartesian, or operational space, inertia matrix. `robot` is a robot object that describes the manipulator dynamics and kinematics, and `q` is an n-element vector of joint coordinates.

Algorithm

The Cartesian inertia matrix is calculated from the joint-space inertia matrix by

$$\mathbf{M}(\underline{x}) = \mathbf{J}(\underline{q})^{-T}\mathbf{M}(\underline{q})\mathbf{J}(\underline{q})^{-1}$$

and relates Cartesian force/torque to Cartesian acceleration

$$\underline{F} = \mathbf{M}(\underline{x})\ddot{\underline{x}}$$

See Also

inertia, robot, rne

References

O. Khatib, "A unified approach for motion and force control of robot manipulators: the operational space formulation," *IEEE Trans. Robot. Autom.*, vol. 3, pp. 43–53, Feb. 1987.

coriolis

Purpose

Compute the manipulator Coriolis/centripetal torque components

Synopsis

```
tau_c = coriolis(robot, q, qd)
```

Description

`coriolis` returns the joint torques due to rigid-body Coriolis and centripetal effects for the specified joint state `q` and velocity `qd`. `robot` is a robot object that describes the manipulator dynamics and kinematics.

If `q` and `qd` are row vectors, `tau_c` is a row vector of joint torques. If `q` and `qd` are matrices, each row is interpreted as a joint state vector, and `tau_c` is a matrix each row being the corresponding joint torques.

Algorithm

Evaluated from the equations of motion, using `rne`, with joint acceleration and gravitational acceleration set to zero,

$$\underline{\tau} = \mathbf{C}(\underline{q}, \underline{\dot{q}})\underline{\dot{q}}$$

Joint friction is ignored in this calculation.

See Also

robot, rne, itorque, gravload

References

M. W. Walker and D. E. Orin. Efficient dynamic computer simulation of robotic mechanisms. *ASME Journal of Dynamic Systems, Measurement and Control*, 104:205–211, 1982.

ctraj

Purpose

Compute a Cartesian trajectory between two points

Synopsis

```
TC = ctraj(T0, T1, m)
TC = ctraj(T0, T1, r)
```

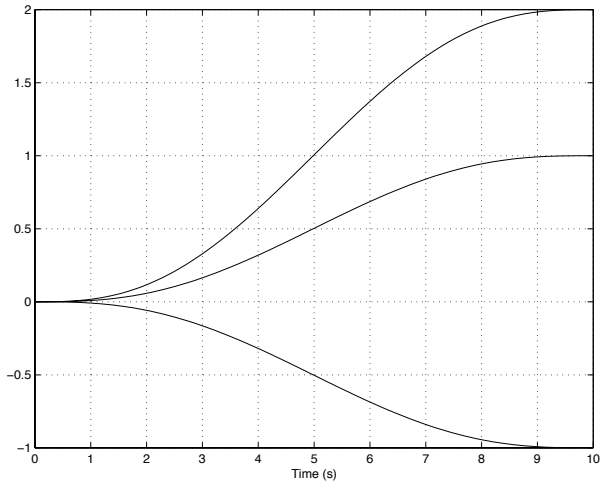
Description

`ctraj` returns a Cartesian trajectory (straight line motion) `TC` from the point represented by homogeneous transform `T0` to `T1`. The number of points along the path is `m` or the length of the given vector `r`. For the second case `r` is a vector of distances along the path (in the range 0 to 1) for each point. The first case has the points equally spaced, but different spacing may be specified to achieve acceptable acceleration profile. `TC` is a $4 \times 4 \times m$ matrix.

Examples

To create a Cartesian path with smooth acceleration we can use the `jtraj` function to create the path vector `r` with continuous derivitives.

```
>> T0 = transl([0 0 0]);  T1 = transl([-1 2 1]);
>> t= [0:0.056:10];
>> r = jtraj(0, 1, t);
>> TC = ctraj(T0, T1, r);
>> plot(t, transl(TC));
```



See Also

trinterp, qinterp, transl

References

R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control.* Cambridge, Massachusetts: MIT Press, 1981.

diff2tr

Purpose

Convert a differential motion vector to a homogeneous transform

Synopsis

```
delta = diff2tr(x)
```

Description

Returns a homogeneous transform representing differential translation and rotation corresponding to Cartesian velocity $x = [v_x\ v_y\ v_z\ \omega_x\ \omega_y\ \omega_z]$.

Algorithm

From mechanics we know that

$$\dot{\mathbf{R}} = Sk(\Omega)\mathbf{R}$$

where $\mathbf{R}$ is an orthonormal rotation matrix and

$$\mathbf{Sk}(\Omega) = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

and is a skew-symmetric matrix. This can be generalized to

$$\dot{\mathbf{T}} = \begin{bmatrix} Sk(\Omega) & \dot{P} \\ 000 & 1 \end{bmatrix} \mathbf{T}$$

for the rotational and translational case.

See Also

tr2diff

References

R. P. Paul. *Robot Manipulators: Mathematics, Programming, and Control*. MIT Press, Cambridge, Massachusetts, 1981.

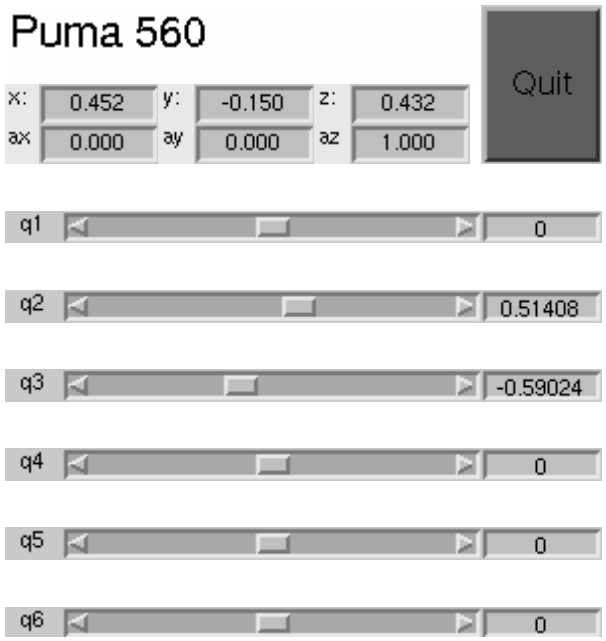
drivebot

Purpose

Drive a graphical robot

Synopsis

```
drivebot(robot)
drivebot(robot, q)
```



Description

Pops up a window with one slider for each joint. Operation of the sliders will drive the graphical robot on the screen. Very useful for gaining an understanding of joint limits and robot workspace.

The joint coordinate state is kept with the graphical robot and can be obtained using the `plot` function. If `q` is specified it is used as the initial joint angle, otherwise the initial value of joint coordinates is taken from the graphical robot.

Examples

To drive a graphical Puma 560 robot

```
>> puma560           % define the robot
>> plot(p560,qz)     % draw it
>> drivebot(p560)    % now drive it
```

See Also

robot/plot,robot

eul2tr

Purpose

Convert Euler angles to a homogeneous transform

Synopsis

```
T = eul2tr([r p y])
T = eul2tr(r,p,y)
```

Description

`eul2tr` returns a homogeneous transformation for the specified Euler angles in radians.

$$T = R_Z(a)R_Y(b)R_Z(c)$$

Cautionary

Note that 12 different Euler angle sets or conventions exist. The convention used here is the common one for robotics, but is not the one used for example in the aerospace community.

See Also

tr2eul, rpy2tr

References

R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. Cambridge, Massachusetts: MIT Press, 1981.

fdyn

Purpose

Integrate forward dynamics

Synopsis

```
[t q qd] = fdyn(robot, t0, t1)
[t q qd] = fdyn(robot, t0, t1, torqfun)
[t q qd] = fdyn(robot, t0, t1, torqfun, q0, qd0) [t
q qd] = fdyn(robot, t0, t1, torqfun, q0, qd0, arg1,
arg2, ...)
```

Description

`fdyn` integrates the manipulator equations of motion over the time interval `t0` to `t1` using MATLAB's `ode45` numerical integration function. Manipulator kinematic and dynamic chacteristics are given by the robot object `robot`. It returns a time vector `t`, and matrices of manipulator joint state `q` and joint velocities `qd`. These matrices have one row per time step and one column per joint.

Actuator torque may be specified by a user function

```
tau = torqfun(t, q, qd, arg1, arg2, ...)
```

where `t` is the current time, and `q` and `qd` are the manipulator joint coordinate and velocity state respectively. Optional arguments passed to `fdyn` will be passed through to the user function. Typically this function would be used to implement some axis control scheme as a function of manipulator state and passed in setpoint information. If `torqfun` is not specified then zero torque is applied to the manipulator.

Initial joint coordinates and velocities may be specified by the optional arguments `q0` and `qd0` respectively.

Algorithm

The joint acceleration is a function of joint coordinate and velocity given by

$$\ddot{\underline{q}} = \mathbf{M}(\underline{q})^{-1}\left\{\underline{\tau} - \mathbf{C}(\underline{q},\dot{\underline{q}})\dot{\underline{q}} - \mathbf{G}(\underline{q}) - \mathbf{F}(\dot{\underline{q}})\right\}$$

Example

The following example shows how `fdyn()` can be used to simulate a robot and its controller. The manipulator is a Puma 560 with simple proportional and derivative controller. The simulation results are shown in the figure, and further gain tuning is clearly required. Note that high gains are required on joints 2 and 3 in order to counter the significant disturbance torque due to gravity.

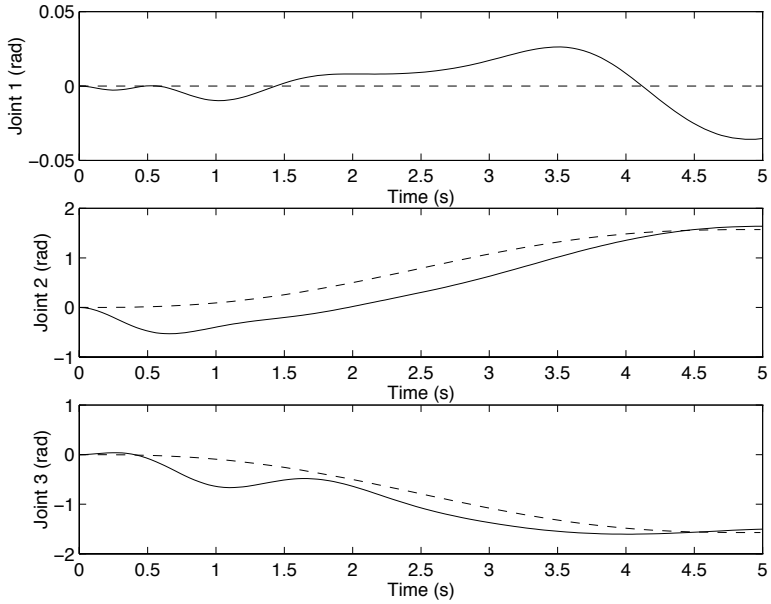
```
>> puma560                          % load Puma parameters
>> t = [0:.056:5]';                 % time vector
>> q_dmd = jtraj(qz, qr,t);         % create a path
>> qt = [t q_dmd];
>> Pgain = [20 100 20 5 5 5];   % set gains
>> Dgain = [-5 -10 -2 0 0 0];
>> [tsim,q,qd] = fdyn(nofriction(p560), 0, 5, 'taufunc', qz, qz, Pgain, Dgain, qt);
```

Note the use of `qz` a zero vector of length 6 defined by `puma560` pads out the two initial condition arguments, and we place the control gains and the path as optional arguments. Note also the use of the `nofriction()` function, see Cautionary note below. The invoked function is

```
    %
    %       taufunc.m
    %
    % user written function to compute joint torque as a function
    % of joint error.  The desired path is passed in via the global
    % matrix qt.  The controller implemented is PD with the proportional
    % and derivative gains given by the global variables Pgain and Dgain
    % respectively.
    %
function tau = taufunc(t, q, qd, Pgain, Dgain, qt)

       % interpolate demanded angles for this time
if t > qt(end,1),        % keep time in range
t = qt(end,1);
end
q_dmd = interp1(qt(:,1), qt(:,2:7), t)';

% compute error and joint torque
e = q_dmd - q;
tau = diag(Pgain)*e + diag(Dgain)*qd;
```



Results of `fdyn()` example. Simulated path shown as solid, and reference path as dashed.

Cautionary

The presence of non-linear friction in the dynamic model can prevent the integration from converging. The function `nofriction()` can be used to return a Coulomb friction free robot object.

See Also

accel, nofriction, rne, robot, ode45

References

M. W. Walker and D. E. Orin. Efficient dynamic computer simulation of robotic mechanisms. *ASME Journal of Dynamic Systems, Measurement and Control*, 104:205–211, 1982.

fkine

Purpose

Forward robot kinematics for serial link manipulator

Synopsis

```
T = fkine(robot, q)
```

Description

`fkine` computes forward kinematics for the joint coordinate `q` giving a homogeneous transform for the location of the end-effector. `robot` is a robot object which contains a kinematic model in either standard or modified Denavit-Hartenberg notation. Note that the robot object can specify an arbitrary homogeneous transform for the base of the robot and a tool offset.

If `q` is a vector it is interpreted as the generalized joint coordinates, and `fkine` returns a homogeneous transformation for the final link of the manipulator. If `q` is a matrix each row is interpreted as a joint state vector, and `T` is a $4 \times 4 \times m$ matrix where `m` is the number of rows in `q`.

Cautionary

Note that the dimensional units for the last column of the `T` matrix will be the same as the dimensional units used in the `robot` object. The units can be whatever you choose (metres, inches, cubits or furlongs) but the choice will affect the numerical value of the elements in the last column of `T`. The Toolbox definitions `puma560` and `stanford` all use SI units with dimensions in metres.

See Also

link, robot

References

R. P. Paul. *Robot Manipulators: Mathematics, Programming, and Control*. MIT Press, Cambridge, Massachusetts, 1981.
J. J. Craig, *Introduction to Robotics*. Addison Wesley, second ed., 1989.

link/friction

Purpose

Compute joint friction torque

Synopsis

```
tau_f = friction(link, qd)
```

Description

`friction` computes the joint friction torque based on friction parameter data, if any, in the link object `link`. Friction is a function only of joint velocity `qd`

If `qd` is a vector then `tau_f` is a vector in which each element is the friction torque for the the corresponding element in `qd`.

Algorithm

The friction model is a fairly standard one comprising viscous friction and direction dependent Coulomb friction

$$F_i(t) = \begin{cases} B_i \dot{q} + \tau_i^- , & \dot{\theta} < 0 \\ B_i \dot{q} + \tau_i^+ , & \dot{\theta} > 0 \end{cases}$$

See Also

link,robot/friction,nofriction

robot/friction

Purpose

Compute robot friction torque vector

Synopsis

```
tau_f = friction(robot, qd)
```

Description

`friction` computes the joint friction torque vector for the robot object `robot` with a joint velocity vector `qd`.

See Also

link, link/friction, nofriction

ftrans

Purpose

Force transformation

Synopsis

```
F2 = ftrans(F, T)
```

Description

Transform the force vector `F` in the current coordinate frame to force vector `F2` in the second coordinate frame. The second frame is related to the first by the homogeneous transform `T`. `F2` and `F` are each 6-element vectors comprising force and moment components $[F_x\ F_y\ F_z\ M_x\ M_y\ M_z]$.

See Also

diff2tr

gravload

Purpose

Compute the manipulator gravity torque components

Synopsis

```
tau_g = gravload(robot, q)
tau_g = gravload(robot, q, grav)
```

Description

`gravload` computes the joint torque due to gravity for the manipulator in pose `q`.

If `q` is a row vector, `tau_g` returns a row vector of joint torques. If `q` is a matrix each row is interpreted as as a joint state vector, and `tau_g` is a matrix in which each row is the gravity torque for the the corresponding row in `q`.

The default gravity direction comes from the robot object but may be overridden by the optional `grav` argument.

See Also

robot, link, rne, itorque, coriolis

References

M. W. Walker and D. E. Orin. Efficient dynamic computer simulation of robotic mechanisms. *ASME Journal of Dynamic Systems, Measurement and Control*, 104:205–211, 1982.

ikine

Purpose

Inverse manipulator kinematics

Synopsis

```
q = ikine(robot, T)
q = ikine(robot, T, q0)
q = ikine(robot, T, q0, M)
```

Description

`ikine` returns the joint coordinates for the manipulator described by the object `robot` whose end-effector homogeneous transform is given by `T`. Note that the robot's base can be arbitrarily specified within the robot object.

If `T` is a homogeneous transform then a row vector of joint coordinates is returned. The estimate for the first step is `q0` if this is given else 0.

If `T` is a homogeneous transform trajectory of size $4 \times 4 \times m$ then `q` will be an $n \times m$ matrix where each row is a vector of joint coordinates corresponding to the last subscript of `T`. The estimate for the first step in the sequence is `q0` if this is given else 0. The initial estimate of `q` for each time step is taken as the solution from the previous time step.

Note that the inverse kinematic solution is generally not unique, and depends on the initial value `q0` (which defaults to 0).

For the case of a manipulator with fewer than 6 DOF it is not possible for the end-effector to satisfy the end-effector pose specified by an arbitrary homogeneous transform. This typically leads to non-convergence in `ikine`. A solution is to specify a 6-element weighting vector, `M`, whose elements are 0 for those Cartesian DOF that are unconstrained and 1 otherwise. The elements correspond to translation along the X-, Y- and Z-axes and rotation about the X-, Y- and Z-axes respectively. For example, a 5-axis manipulator may be incapable of independantly controlling rotation about the end-effector's Z-axis. In this case `M = [1 1 1 1 1 0]` would enable a solution in which the end-effector adopted the pose `T` *except* for the end-effector rotation. The number of non-zero elements should equal the number of robot DOF.

Algorithm

The solution is computed iteratively using the pseudo-inverse of the manipulator Jacobian.

$$\dot{\underline{q}} = \mathbf{J}^{+}(\underline{q})\Delta\left(F(\underline{q}) - T\right)$$

where Δ returns the 'difference' between two transforms as a 6-element vector of displacements and rotations (see `tr2diff`).

Cautionary

Such a solution is completely general, though much less efficient than specific inverse kinematic solutions derived symbolically.

The returned joint angles may be in non-minimum form, ie. $q + 2n\pi$.

This approach allows a solution to obtained at a singularity, but the joint coordinates within the null space are arbitrarily assigned.

Note that the dimensional units used for the last column of the `T` matrix must agree with the dimensional units used in the robot definition. These units can be whatever you choose (metres, inches, cubits or furlongs) but they must be consistent. The Toolbox definitions `puma560` and `stanford` all use SI units with dimensions in metres.

See Also

fkine, tr2diff, jacob0, ikine560, robot

References

S. Chieaverini, L. Sciavicco, and B. Siciliano, "Control of robotic systems through singularities," in *Proc. Int. Workshop on Nonlinear and Adaptive Control: Issues in Robotics* (C. C. de Wit, ed.), Springer-Verlag, 1991.

ikine560

Purpose

Inverse manipulator kinematics for Puma 560 like arm

Synopsis

```
q = ikine560(robot, config)
```

Description

`ikine560` returns the joint coordinates corresponding to the end-effector homogeneous transform `T`. It is computed using a symbolic solution appropriate for Puma 560 like robots, that is, all revolute 6DOF arms, with a spherical wrist. The use of a symbolic solution means that it executes over 50 times faster than `ikine` for a Puma 560 solution.

A further advantage is that `ikine560()` allows control over the specific solution returned. `config` is a string which contains one or more of the configuration control letter codes

`'l'`	left-handed (lefty) solution (default)
`'r'`	†right-handed (righty) solution
`'u'`	†elbow up solution (default)
`'d'`	elbow down solution
`'f'`	†wrist flipped solution
`'n'`	wrist not flipped solution (default)

Cautionary

Note that the dimensional units used for the last column of the `T` matrix must agree with the dimensional units used in the `robot` object. These units can be whatever you choose (metres, inches, cubits or furlongs) but they must be consistent. The Toolbox definitions `puma560` and `stanford` all use SI units with dimensions in metres.

See Also

fkine, ikine, robot

References

R. P. Paul and H. Zhang, "Computationally efficient kinematics for manipulators with spherical wrists," *Int. J. Robot. Res.*, vol. 5, no. 2, pp. 32–44, 1986.

Author

Robert Biro and Gary McMurray, Georgia Institute of Technology, gt2231a@acmex.gatech.edu

inertia

Purpose

Compute the manipulator joint-space inertia matrix

Synopsis

```
M = inertia(robot, q)
```

Description

`inertia` computes the joint-space inertia matrix which relates joint torque to joint acceleration

$$\underline{\tau} = \mathbf{M}(\underline{q})\ddot{\underline{q}}$$

`robot` is a robot object that describes the manipulator dynamics and kinematics, and `q` is an n-element vector of joint state. For an n-axis manipulator `M` is an $n \times n$ symmetric matrix.

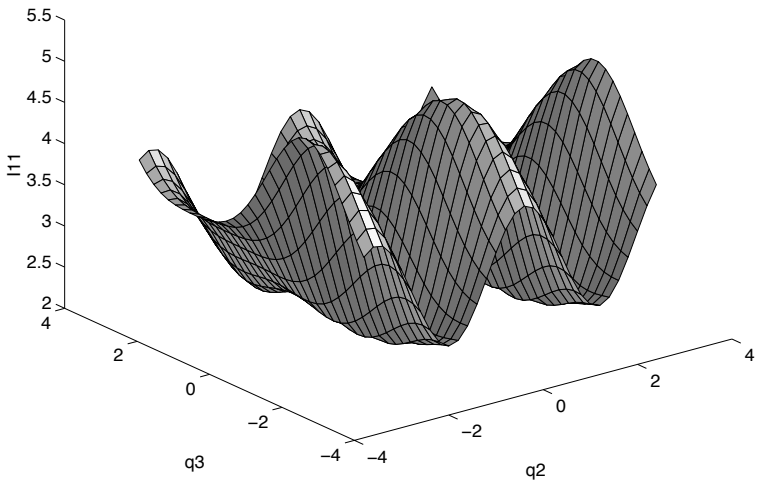
If `q` is a matrix each row is interpreted as a joint state vector, and `I` is an $n \times n \times m$ matrix where `m` is the number of rows in `q`.

Note that if the `robot` contains motor inertia parameters then motor inertia, referred to the link reference frame, will be added to the diagonal of `M`.

Example

To show how the inertia 'seen' by the waist joint varies as a function of joint angles 2 and 3 the following code could be used.

```
>> [q2,q3] = meshgrid(-pi:0.2:pi, -pi:0.2:pi);
>> q = [zeros(length(q2(:)),1) q2(:) q3(:) zeros(length(q2(:)),3)];
>> I = inertia(p560, q);
>> surfl(q2, q3, squeeze(I(1,1,:)));
```



See Also

robot, rne, itorque, coriolis, gravload

References

M. W. Walker and D. E. Orin. Efficient dynamic computer simulation of robotic mechanisms. *ASME Journal of Dynamic Systems, Measurement and Control*, 104:205–211, 1982.

ishomog

Purpose

Test if argument is a homogeneous transformation

Synopsis

```
ishomog(x)
```

Description

Returns true if `x` is a 4×4 matrix.

itorque

Purpose

Compute the manipulator inertia torque component

Synopsis

```
tau_i = itorque(robot, q, qdd)
```

Description

`itorque` returns the joint torque due to inertia at the specified pose `q` and acceleration `qdd` which is given by

$$\underline{\tau}_i = \mathbf{M}(\underline{q})\underline{\ddot{q}}$$

If `q` and `qdd` are row vectors, `itorque` is a row vector of joint torques. If `q` and `qdd` are matrices, each row is interpreted as a joint state vector, and `itorque` is a matrix in which each row is the inertia torque for the corresponding rows of `q` and `qdd`.

`robot` is a robot object that describes the kinematics and dynamics of the manipulator and drive. If `robot` contains motor inertia parameters then motor inertia, referred to the link reference frame, will be included in the diagonal of `M` and influence the inertia torque result.

See Also

robot, rne, coriolis, inertia, gravload

jacob0

Purpose

Compute manipulator Jacobian in base coordinates

Synopsis

```
jacob0(robot, q)
```

Description

`jacob0` returns a Jacobian matrix for the robot object `robot` in the pose `q` and as expressed in the base coordinate frame.

The manipulator Jacobian matrix, ${}^0\mathbf{J}_q$, maps differential velocities in joint space, $\dot{q}$, to Cartesian velocity of the end-effector expressed in the base coordinate frame.

$$ {}^0\underline{\dot{x}} = {}^0\mathbf{J}_q(\underline{q})\underline{\dot{q}} $$

For an n-axis manipulator the Jacobian is a $6 \times n$ matrix.

See Also

jacobn, diff2tr, tr2diff, robot

References

R. P. Paul, B. Shimano and G. E. Mayer. *Kinematic Control Equations for Simple Manipulators*. IEEE Systems, Man and Cybernetics 11(6), pp 449-455, June 1981.

jacobn

Purpose

Compute manipulator Jacobian in end-effector coordinates

Synopsis

```
jacobn(robot, q)
```

Description

`jacobn` returns a Jacobian matrix for the robot object `robot` in the pose `q` and as expressed in the end-effector coordinate frame.

The manipulator Jacobian matrix, ${}^0\mathbf{J}_q$, maps differential velocities in joint space, $\dot{q}$, to Cartesian velocity of the end-effector expressed in the end-effector coordinate frame.

$$ {}^n\underline{\dot{x}} = {}^n\mathbf{J}_q(\underline{q})\underline{\dot{q}} $$

The relationship between tool-tip forces and joint torques is given by

$$ \underline{\tau} = {}^n\mathbf{J}_q(\underline{q})'{}^n\underline{F} $$

For an n-axis manipulator the Jacobian is a $6 \times n$ matrix.

See Also

jacob0, diff2tr, tr2diff, robot

References

R. P. Paul, B. Shimano and G. E. Mayer. *Kinematic Control Equations for Simple Manipulators*. IEEE Systems, Man and Cybernetics 11(6), pp 449-455, June 1981.

jtraj

Purpose

Compute a joint space trajectory between two joint coordinate poses

Synopsis

```
[q qd qdd] = jtraj(q0, q1, n)
[q qd qdd] = jtraj(q0, q1, n, qd0, qd1)
[q qd qdd] = jtraj(q0, q1, t)
[q qd qdd] = jtraj(q0, q1, t, qd0, qd1)
```

Description

`jtraj` returns a joint space trajectory `q` from joint coordinates `q0` to `q1`. The number of points is `n` or the length of the given time vector `t`. A 7th order polynomial is used with default zero boundary conditions for velocity and acceleration.

Non-zero boundary velocities can be optionally specified as `qd0` and `qd1`.

The trajectory is a matrix, with one row per time step, and one column per joint. The function can optionally return a velocity and acceleration trajectories as `qd` and `qdd` respectively.

See Also

ctraj

link

Purpose

Link object

Synopsis

```
L = link
L = link([alpha, a, theta, d], convention)
L = link([alpha, a, theta, d, sigma], convention)
L = link(dyn_row, convention)
A = link(q)
show(L)
```

Description

The `link` function constructs a **link** object. The object contains kinematic and dynamic parameters as well as actuator and transmission parameters. The first form returns a default object, while the second and third forms initialize the kinematic model based on Denavit and Hartenberg parameters. The dynamic model can be initialized using the fourth form of the constructor where `dyn_row` is a 1×20 matrix which is one row of the legacy `dyn` matrix.

By default the standard Denavit and Hartenberg conventions are assumed but this can be overridden by the optional `convention` argument which can be set to either `'modified'` or `'standard'` (default). Note that any abbreviation of the string can be used, ie. `'mod'` or even `'m'`.

The second last form given above is not a constructor but a link method that returns the link transformation matrix for the given joint coordinate. The argument is given to the link object using parenthesis. The single argument is taken as the link variable q and substituted for θ or D for a revolute or prismatic link respectively.

The Denavit and Hartenberg parameters describe the spatial relationship between this link and the previous one. The meaning of the fields for each kinematic convention are summarized in the following table.

variable	DH	MDH	description
`alpha`	α_i	α_{i-1}	link twist angle
`A`	A_i	A_{i-1}	link length
`theta`	θ_i	θ_i	link rotation angle
`D`	D_i	D_i	link offset distance
`sigma`	σ_i	σ_i	joint type; 0 for revolute, non-zero for prismatic

Since Matlab does not support the concept of public class variables methods have been written to allow link object parameters to be referenced (r) or assigned (a) as given by the following table

Method	Operations	Returns
link.alpha	r+a	link twist angle
link.A	r+a	link length
link.theta	r+a	link rotation angle
link.D	r+a	link offset distance
link.sigma	r+a	joint type; 0 for revolute, non-zero for prismatic
link.RP	r	joint type; 'R' or 'P'
link.mdh	r+a	DH convention: 0 if standard, 1 if modified
link.I	r	3×3 symmetric inertia matrix
link.I	a	assigned from a 3×3 matrix or a 6-element vector interpretted as $[I_{xx}\, I_{yy}\, I_{zz}\, I_{xy}\, I_{yz}\, I_{xz}]$
link.m	r+a	link mass
link.r	r+a	3×1 link COG vector
link.G	r+a	gear ratio
link. Jm	r+a	motor inertia
link.B	r+a	viscous friction
link. Tc	r	Coulomb friction, 1×2 vector where $[\tau^+\ \tau^-]$
link. Tc	a	Coulomb friction; for symmetric friction this is a scalar, for asymmetric friction it is a 2-element vector for positive and negative velocity
link.dh	r+a	row of legacy DH matrix
link.dyn	r+a	row of legacy DYN matrix
link.qlim	r+a	joint coordinate limits, 2-vector
link.islimit(q)	r	return true if value of q is outside the joint limit bounds
link.offset	r+a	joint coordinate offset (see discussion for robot object).

The default is for standard Denavit-Hartenberg conventions, zero friction, mass and inertias.

The display method gives a one-line summary of the link's kinematic parameters. The show method displays as many link parameters as have been initialized for that link.

Examples

```
>> L = link([-pi/2, 0.02, 0, 0.15])
L =
  -1.570796     0.020000        0.000000        0.150000        R       (std)
>> L.RP
ans =
  R
>> L.mdh
ans =
     0
>> L.G = 100;
>> L.Tc = 5;
>> L
```

```
L =
  -1.570796     0.020000         0.000000         0.150000         R      (std)
>> show(L)
alpha  = -1.5708
A      = 0.02
theta  = 0
D      = 0.15
sigma  = 0
mdh    = 0
G      = 100
Tc     = 5 -5
>>
```

Algorithm

For the standard Denavit-Hartenberg conventions the homogeneous transform

$$ {}^{i-1}\mathbf{A}_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

represents each link's coordinate frame with respect to the previous link's coordinate system. For a revolute joint θ_i is offset by

For the modified Denavit-Hartenberg conventions it is instead

$$ {}^{i-1}\mathbf{A}_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i\cos\alpha_{i-1} & \cos\theta_i\cos\alpha_{i-1} & -\sin\alpha_{i-1} & -d_i\sin\alpha_{i-1} \\ \sin\theta_i\sin\alpha_{i-1} & \cos\theta_i\sin\alpha_{i-1} & \cos\alpha_{i-1} & d_i\cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

See Also

showlink, robot

References

R. P. Paul. *Robot Manipulators: Mathematics, Programming, and Control*. MIT Press, Cambridge, Massachusetts, 1981.

J. J. Craig, *Introduction to Robotics*. Addison Wesley, second ed., 1989.

maniplty

Purpose

Manipulability measure

Synopsis

```
m = maniplty(robot, q)
m = maniplty(robot, q, which)
```

Description

`maniplty` computes the scalar manipulability index for the manipulator at the given pose. Manipulability varies from 0 (bad) to 1 (good). `robot` is a robot object that contains kinematic and optionally dynamic parameters for the manipulator. Two measures are supported and are selected by the optional argument `which` can be either `'yoshikawa'` (default) or `'asada'`. Yoshikawa's manipulability measure is based purely on kinematic data, and gives an indication of how 'far' the manipulator is from singularities and thus able to move and exert forces uniformly in all directions.

Asada's manipulability measure utilizes manipulator dynamic data, and indicates how close the inertia ellipsoid is to spherical.

If `q` is a vector `maniplty` returns a scalar manipulability index. If `q` is a matrix `maniplty` returns a column vector and each row is the manipulability index for the pose specified by the corresponding row of `q`.

Algorithm

Yoshikawa's measure is based on the condition number of the manipulator Jacobian

$$\eta_{yoshi} = \sqrt{|\mathbf{J}(\underline{q})\mathbf{J}(\underline{q})'|}$$

Asada's measure is computed from the Cartesian inertia matrix

$$\mathbf{M}(\underline{x}) = \mathbf{J}(\underline{q})^{-T}\mathbf{M}(\underline{q})\mathbf{J}(\underline{q})^{-1}$$

The Cartesian manipulator inertia ellipsoid is

$$\underline{x}'\mathbf{M}(\underline{x})\underline{x} = 1$$

and gives an indication of how well the manipulator can accelerate in each of the Cartesian directions. The scalar measure computed here is the ratio of the smallest/largest ellipsoid axes

$$\eta_{asada} = \frac{\min x}{\max x}$$

Ideally the ellipsoid would be spherical, giving a ratio of 1, but in practice will be less than 1.

See Also

jacob0, inertia,robot

References

T. Yoshikawa, "Analysis and control of robot manipulators with redundancy," in *Proc. 1st Int. Symp. Robotics Research*, (Bretton Woods, NH), pp. 735–747, 1983.

robot/nofriction

Purpose

Remove friction from robot object

Synopsis

```
robot2 = nofriction(robot)
robot2 = nofriction(robot, 'all')
```

Description

Return a new robot object with modified joint friction properties. The first form sets the Coulomb friction values to zero in the constituent links The second form sets viscous and Coulomb friction values in the constituent links are set to zero.

The resulting robot object has its name string prepended with 'NF/'.

This is important for forward dynamics computation (`fdyn()`) where the presence of friction can prevent the numerical integration from converging.

See Also

link/nofriction, robot, link, friction, fdyn

link/nofriction

Purpose

Remove friction from link object

Synopsis

```
link2 = nofriction(link)
link2 = nofriction(link, 'all')
```

Description

Return a new link object with modified joint friction properties. The first form sets the Coulomb friction values to zero. The second form sets both viscous and Coulomb friction values to zero.

This is important for forward dynamics computation (`fdyn()`) where the presence of friction can prevent the numerical integration from converging.

See Also

robot/nofriction, link, friction, fdyn

oa2tr

Purpose

Convert OA vectors to homogeneous transform

Synopsis

```
oa2tr(o, a)
```

Description

`oa2tr` returns a rotational homogeneous transformation specified in terms of the Cartesian orientation and approach vectors `o` and `a` respectively.

Algorithm

$$\mathbf{T} = \begin{bmatrix} \underline{\hat{o}} \times \underline{\hat{a}} & \underline{\hat{o}} & \underline{\hat{a}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $\underline{\hat{o}}$ and $\underline{\hat{a}}$ are unit vectors corresponding to `o` and `a` respectively.

See Also

rpy2tr, eul2tr

perturb

Purpose

Perturb robot dynamic parameters

Synopsis

```
robot2 = perturb(robot, p)
```

Description

Return a new robot object with randomly modified dynamic parameters: link mass and inertia. The perturbation is multiplicative so that values are multiplied by random numbers in the interval (1-p) to (1+p).

Useful for investigating the robustness of various model-based control schemes where one model forms the basis of the model-based controller and the peturbed model is used for the actual plant.

The resulting robot object has its name string prepended with 'P/'.

See Also

fdyn, rne, robot

puma560

Purpose

Create a Puma 560 robot object

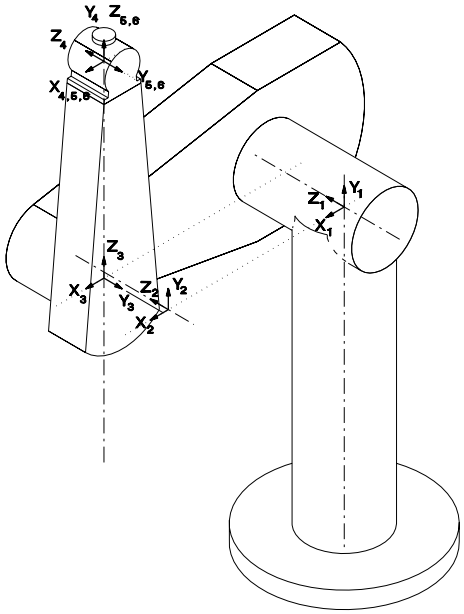
Synopsis

```
puma560
```

Description

Creates the **robot** object `p560` which describes the kinematic and dynamic characteristics of a Unimation Puma 560 manipulator. The kinematic conventions used are as per Paul and Zhang, and all quantities are in standard SI units.

Also defines the joint coordinate vectors `qz`, `qr` and `qstretch` corresponding to the zero-angle, ready and fully extended (in X-direction) poses respectively.



Details of coordinate frames used for the Puma 560 shown here in its zero angle pose.

See Also

robot, puma560akb, stanford

References

R. P. Paul and H. Zhang, "Computationally efficient kinematics for manipulators with spherical wrists," *Int. J. Robot. Res.*, vol. 5, no. 2, pp. 32–44, 1986.

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

P. Corke and B. Armstrong-Hélouvry, "A meta-study of PUMA 560 dynamics: A critical appraisal of literature data," *Robotica*, vol. 13, no. 3, pp. 253–258, 1995.

puma560akb

Purpose

Create a Puma 560 robot object

Synopsis

```
puma560akb
```

Description

Creates the **robot** object `p560m` which describes the kinematic and dynamic characteristics of a Unimation Puma 560 manipulator. It uses Craig's modified Denavit-Hartenberg notation with the particular kinematic conventions from Armstrong, Khatib and Burdick. All quantities are in standard SI units.

Also defines the joint coordinate vectors `qz`, `qr` and `qstretch` corresponding to the zero-angle, ready and fully extended (in X-direction) poses respectively.

See Also

robot, puma560, stanford

References

B. Armstrong, O. Khatib, and J. Burdick, "The explicit dynamic model and inertial parameters of the Puma 560 arm," in *Proc. IEEE Int. Conf. Robotics and Automation*, vol. 1, (Washington, USA), pp. 510–18, 1986.

qinterp

Purpose

Interpolate unit-quaternions

Synopsis

```
QI = qinterp(Q1, Q2, r)
```

Description

Return a unit-quaternion that interpolates between Q1 and Q2 as `r` varies between 0 and 1 inclusively. This is a spherical linear interpolation (slerp) that can be interpreted as interpolation along a great circle arc on a sphere.

If `r` is a vector, then a cell array of quaternions is returned corresponding to successive values of `r`.

Examples

A simple example

```
>> q1 = quaternion(rotx(0.3))

q1 =
  0.98877 <0.14944, 0, 0>

>> q2 = quaternion(roty(-0.5))

q2 =
  0.96891 <0, -0.2474, 0>

>> qinterp(q1, q2, 0)

ans =
  0.98877 <0.14944, 0, 0>

>> qinterp(q1, q2, 1)

ans =
  0.96891 <0, -0.2474, 0>

>> qinterp(q1, q2, 0.3)

ans =
  0.99159 <0.10536, -0.075182, 0>

>>
```

References

K. Shoemake, "Animating rotation with quaternion curves.," in *Proceedings of ACM SIGGRAPH*, (San Francisco), pp. 245–254, The Singer Company, Link Flight Simulator Division, 1985.

quaternion

Purpose

Quaternion object

Synopsis

```
q = quaternion(qq)
q = quaternion(v, theta)
q = quaternion(R)
q = quaternion([s vx vy vz])
```

Description

`quaternion` is the constructor for a **quaternion** object. The first form returns a new object with the same value as its argument. The second form initializes the quaternion to a rotation of `theta` about the vector `v`.

Examples

A simple example

```
>> quaternion(1, [1 0 0])

ans =
  0.87758 <0.47943, 0, 0>

>> quaternion( rotx(1) )

ans =
  0.87758 <0.47943, 0, 0>

>>
```

The third form sets the quaternion to a rotation equivalent to the given 3×3 rotation matrix, or the rotation submatrix of a 4×4 homogeneous transform.

The fourth form sets the four quaternion elements directly where `s` is the scalar component and `[vx vy vz]` the vector.

All forms, except the last, return a unit quaternion, ie. one whose magnitude is unity.

Some operators are overloaded for the quaternion class

`q1 * q2`	returns quaternion product or compounding
`q * v`	returns a quaternion vector product, that is the vector v is rotated by the quaternion. v is a 3×3 vector
`q1 / q2`	returns $q_1 * q_2^{-1}$
`q^j`	returns q^j where j is an integer exponent. For $j > 0$ the result is obtained by repeated multiplication. For $j < 0$ the final result is inverted.
`double(q)`	returns the quaternion coeffients as a 4-element row vector
`inv(q)`	returns the quaterion inverse
`norm(q)`	returns the quaterion magnitude
`plot(q)`	displays a 3D plot showing the standard coordinate frame after rotation by `q`.
`unit(q)`	returns the corresponding unit quaterion

Some public class variables methods are also available for reference only.

method	Returns
`quaternion.``d`	return 4-vector of quaternion elements
`quaternion.``s`	return scalar component
`quaternion.``v`	return vector component
`quaternion.``t`	return equivalent homogeneous transformation matrix
`quaternion.``r`	return equivalent orthonormal rotation matrix

Examples

```
>> t = rotx(0.2)
t =
    1.0000         0         0         0
         0    0.9801   -0.1987         0
         0    0.1987    0.9801         0
         0         0         0    1.0000
>> q1 = quaternion(t)
q1 =
  0.995 <0.099833, 0, 0>

>> q1.r
ans =
    1.0000         0         0
         0    0.9801   -0.1987
         0    0.1987    0.9801

>> q2 = quaternion( roty(0.3) )
q2 =
  0.98877 <0, 0.14944, 0>

>> q1 * q2
ans =
```

```
  0.98383 <0.098712, 0.14869, 0.014919>

>> q1*q1
ans =
  0.98007 <0.19867, 0, 0>

>> q1^2
ans =
  0.98007 <0.19867, 0, 0>

>> q1*inv(q1)
ans =
  1 <0, 0, 0>

>> q1/q1
ans =
  1 <0, 0, 0>

>> q1/q2
ans =
  0.98383 <0.098712, -0.14869, -0.014919>

>> q1*q2^-1
ans =
  0.98383 <0.098712, -0.14869, -0.014919>
```

Cautionary

At the moment vectors or arrays of quaternions are not supported. You can however use cell arrays to hold a number of quaternions.

See Also

quaternion/plot

References

K. Shoemake, “Animating rotation with quaternion curves.,” in *Proceedings of ACM SIGGRAPH*, (San Francisco), pp. 245–254, The Singer Company, Link Flight Simulator Division, 1985.

quaternion/plot

Purpose

Plot quaternion rotation

Synopsis

`plot(Q)`

Description

`plot` is overloaded for **quaternion** objects and displays a 3D plot which shows how the standard axes are transformed under that rotation.

Examples

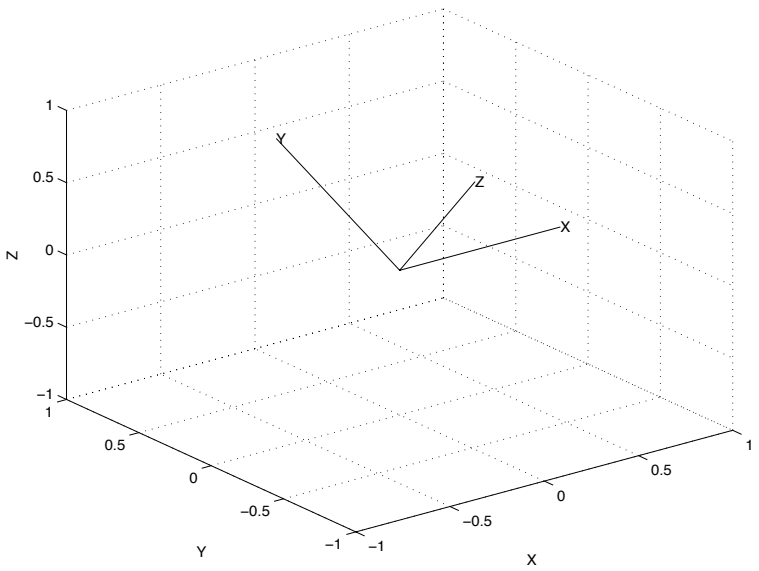
A rotation of 0.3rad about the X axis. Clearly the X axis is invariant under this rotation.

```
>> q=quaternion(rotx(0.3))

q =

  0.85303<0.52185, 0, 0>

>> plot(q)
```



See Also

quaternion

rne

Purpose

Compute inverse dynamics via recursive Newton-Euler formulation

Synopsis

```
tau = rne(robot, q, qd, qdd)
tau = rne(robot, [q qd qdd])

tau = rne(robot, q, qd, qdd, grav)
tau = rne(robot, [q qd qdd], grav)

tau = rne(robot, q, qd, qdd, grav, fext)
tau = rne(robot, [q qd qdd], grav, fext)
```

Description

`rne` computes the equations of motion in an efficient manner, giving joint torque as a function of joint position, velocity and acceleration.

If `q`, `qd` and `qdd` are row vectors then `tau` is a row vector of joint torques. If `q`, `qd` and `qdd` are matrices then `tau` is a matrix in which each row is the joint torque for the corresponding rows of `q`, `qd` and `qdd`.

Gravity direction is defined by the robot object but may be overridden by providing a gravity acceleration vector `grav = [gx gy gz]`.

An external force/moment acting on the end of the manipulator may also be specified by a 6-element vector `fext = [Fx Fy Fz Mx My Mz]` in the end-effector coordinate frame.

The torque computed may contain contributions due to armature inertia and joint friction if these are specified in the parameter matrix `dyn`.

The MEX-file version of this function is over 1000 times faster than the M-file. See Section 1 of this manual for information about how to compile and install the MEX-file.

Algorithm

Coumputes the joint torque

$$\underline{\tau} = \mathbf{M}(\underline{q})\ddot{\underline{q}} + \mathbf{C}(\underline{q},\dot{\underline{q}})\dot{\underline{q}} + \mathbf{F}(\dot{\underline{q}}) + \mathbf{G}(\underline{q})$$

where **M** is the manipulator inertia matrix, **C** is the Coriolis and centripetal torque, **F** the viscous and Coulomb friction, and **G** the gravity load.

Cautionary

The MEX file currently ignores support base and tool transforms.

See Also

robot, fdyn, accel, gravload, inertia, friction

Limitations

A MEX file is currently only available for Sparc architecture.

References

J. Y. S. Luh, M. W. Walker, and R. P. C. Paul. On-line computational scheme for mechanical manipulators. *ASME Journal of Dynamic Systems, Measurement and Control*, 102:69–76, 1980.

robot

Purpose

Robot object

Synopsis

```
r = robot
r = robot(rr)
r = robot(link ...)
r = robot(DH ...)
r = robot(DYN ...)
```

Description

`robot` is the constructor for a robot object. The first form creates a default robot, and the second form returns a new robot object with the same value as its argument. The third form creates a robot from a cell array of link objects which define the robot's kinematics and optionally dynamics. The fourth and fifth forms create a robot object from legacy DH and DYN format matrices.

The last three forms all accept optional trailing string arguments which are taken in order as being robot name, manufacturer and comment.

Since Matlab does not support the concept of public class variables methods have been written to allow robot object parameters to be referenced (r) or assigned (a) as given by the following table

method	Operation	Returns
robot.`n`	r	number of joints
robot.`link`	r+a	cell array of link objects
robot.`name`	r+a	robot name string
robot.`manuf`	r+a	robot manufacturer string
robot.`comment`	r+a	general comment string
robot.`gravity`	r+a	3-element vector defining gravity direction
robot.`mdh`	r	DH convention: 0 if standard, 1 if modified. Determined from the link objects.
robot.`base`	r+a	homogeneous transform defining base of robot
robot.`tool`	r+a	homogeneous transform defining tool of robot
robot.`dh`	r	legacy DH matrix
robot.`dyn`	r	legacy DYN matrix
robot.`q`	r+a	joint coordinates
robot.`qlim`	r+a	joint coordinate limits, $n \times 2$ matrix
robot.`islimit`	r	joint limit vector, for each joint set to -1, 0 or 1 depending if below low limit, OK, or greater than upper limit
robot.`offset`	r+a	joint coordinate offsets
robot.`plotopt`	r+a	options for `plot()`
robot.`lineopt`	r+a	line style for robot graphical links
robot.`shadowopt`	r+a	line style for robot shadow links
robot.`handle`	r+a	graphics handles

Some of these operations at the robot level are actually wrappers around similarly named link object

functions: `offset`, `qlim`, `islimit`.

The offset vector is added to the user specified joint angles before any kinematic or dynamic function is invoked (it is actually implemented within the link object). Similarly it is subtracted after an operation such as inverse kinematics. The need for a joint offset vector arises because of the constraints of the Denavit-Hartenberg (or modified Denavit-Hartenberg) notation. The pose of the robot with zero joint angles is frequently some rather unusual (or even unachievable) pose. The joint coordinate offset provides a means to make an arbitrary pose correspond to the zero joint angle case.

Default values for robot parameters are:

robot.name	'noname'
robot.manuf	''
robot.comment	''
robot.gravity	$[0\,0\,9.81]\,\mathrm{m/s^2}$
robot.offset	`ones(n,1)`
robot.base	`eye(4,4)`
robot.tool	`eye(4,4)`
robot.lineopt	'Color', 'black', 'Linewidth', 4
robot.shadowopt	'Color', 'black', 'Linewidth', 1

The multiplication operator, `*`, is overloaded and the product of two robot objects is a robot which is the series connection of the multiplicands. Tool transforms of all but the last robot are ignored, base transform of all but the first robot are ignored.

The `plot` function is also overloaded and is used to provide a robot animation.

Examples

```
>> L{1} = link([ pi/2  0  0  0])
L =
    [1x1 link]

>> L{2} = link([ 0  0 0.5 0])
L =
    [1x1 link]    [1x1 link]

>> r = robot(L)

r =
 (2 axis, RR)
               grav = [0.00 0.00 9.81]
               standard D&H parameters

  alpha           A              theta           D             R/P
1.570796        0.000000        0.000000        0.000000        R     (std)
0.000000        0.000000        0.500000        0.000000        R     (std)
```

```
>>
```

See Also

link,plot

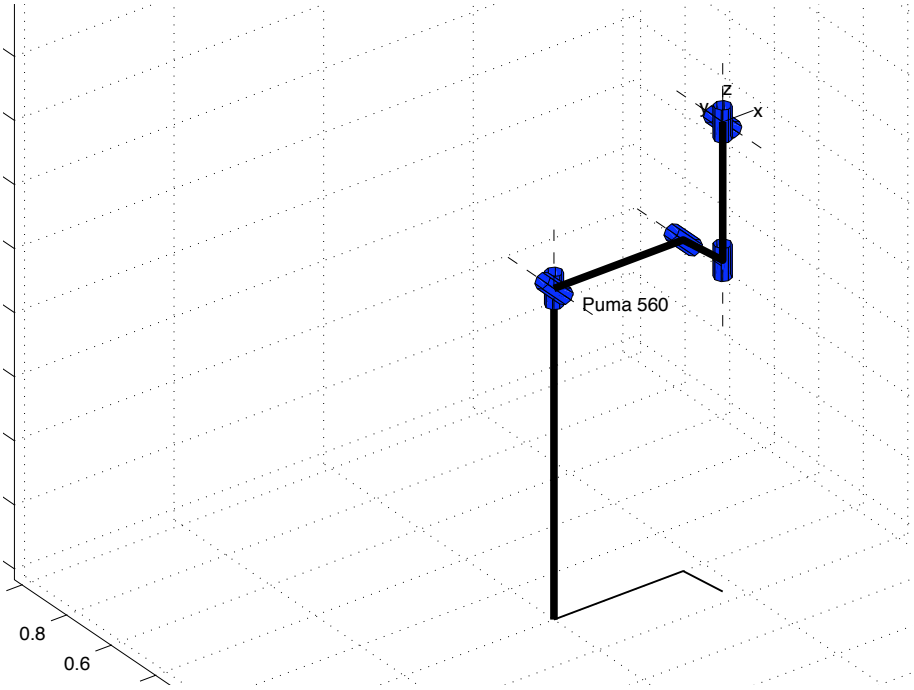
robot/plot

Purpose

Graphical robot animation

Synopsis

```
plot(robot, q)
plot(robot, q, arguments...)
```



Description

`plot` is overloaded for **robot** objects and displays a graphical representation of the robot given the kinematic information in `robot`. The robot is represented by a simple stick figure polyline where line segments join the origins of the link coordinate frames. If `q` is a matrix representing a joint-space trajectory then an animation of the robot motion is shown.

GRAPHICAL ANNOTATIONS

The basic stick figure robot can be annotated with

- shadow on the 'floor'
- XYZ wrist axes and labels, shown by 3 short orthogonal line segments which are colored: red (X or normal), green (Y or orientation) and blue (Z or approach). They can be optionally labelled XYZ or NOA.
- joints, these are 3D cylinders for revolute joints and boxes for prismatic joints
- the robot's name

All of these require some kind of dimension and this 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 below. These various annotations do slow the rate at which animations will be rendered.

OPTIONS

Options are specified by a variable length argument list comprising strings and numeric values. The allowed values are:

`workspace w`	set the 3D plot bounds or workspace to the matrix `[xmin xmax ymin ymax zmin zmax]`
`perspective`	show a perspective view
`ortho`	show an orthogonal view
`base, nobase`	control display of base, a line from the floor upto joint 0
`wrist, nowrist`	control display of wrist axes
`name, noname`	control display of robot name near joint 0
`shadow, noshadow`	control display of a 'shadow' on the floor
`joints, nojoints`	control display of joints, these are cylinders for revolute joints and boxes for prismatic joints
`xyz`	wrist axis labels are X, Y, Z
`noa`	wrist axis labels are N, O, A
`mag scale`	annotation scale factor
`erase, noerase`	control erasure of robot after each change
`loop, noloop`	control whether animation is repeated endlessly

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

1. Cell array of options returned by the function `PLOTBOTOPT` if found on the user's current path.
2. Cell array of options returned by the `.plotopt` method of the `robot` object. These are set by the `.plotopt` method.
3. List of arguments in the command line.

GETTING GRAPHICAL ROBOT STATE

Each graphical robot has a unique tag set equal to the robot's name. When `plot` is called it looks for all graphical objects with that name and moves them. The graphical robot holds a copy of the `robot` object as `UserData`. That copy contains the graphical handles of all the graphical sub-elements of the robot and also the current joint angle state.

This state is used, and adjusted, by the `drivebot` function. The current joint angle state can be obtained by `q = plot(robot)`. If multiple instances exist, that of the first one returned by `findobj()` is given.

Examples

To animate two Pumas moving in the same figure window.

```
>> clf
>> p560b = p560;                          % duplicate the robot
>> p560b.name = 'Another Puma 560';       % give it a unique name
>> p560b.base = transl([-.05 0.5 0]);     % move its base
```

```
>> plot(p560, qr);   % display it at ready position
>> hold on
>> plot(p560b, qr);  % display it at ready position
>> t = [0:0.056:10];
>> jt = jtraj(qr, qstretch, t);  % trajectory to stretch pose
>> for q = jt', % for all points on the path
>>    plot(p560, q);
>>    plot(p560b, q);
>> end
```

To show multiple views of the same robot.

```
>> clf
>> figure             % create a new figure
>> plot(p560, qz);    % add a graphical robot
>> figure             % create another figure
>> plot(p560, qz);    % add a graphical robot
>> plot(p560, qr);    % both robots should move
```

Now the two figures can be adjusted to give different viewpoints, for instance, plan and elevation.

Cautionary

`plot()` options are only processed on the first call when the graphical object is established, they are skipped on subsequent calls. Thus if you wish to change options, clear the figure before replotting.

See Also

drivebot, fkine, robot

rotvec

Purpose

Rotation about a vector

Synopsis

```
T = rotvec(v, theta)
```

Description

`rotvec` returns a homogeneous transformation representing a rotation of `theta` radians about the vector `v`.

See Also

rotx, roty, rotz

rotx,roty,rotz

Purpose

Rotation about X, Y or Z axis

Synopsis

```
T = rotx(theta)
T = roty(theta)
T = rotz(theta)
```

Description

Return a homogeneous transformation representing a rotation of `theta` radians about the X, Y or Z axes.

See Also

rotvec

rpy2tr

Purpose

Roll/pitch/yaw angles to homogeneous transform

Synopsis

```
T = rpy2tr([r p y])
T = rpy2tr(r,p,y)
```

Description

`rpy2tr` returns a homogeneous transformation for the specified roll/pitch/yaw angles in radians.

$$T = R_Z(r)R_Y(p)R_X(y)$$

See Also

tr2rpy, eul2tr

References

R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. Cambridge, Massachusetts: MIT Press, 1981.

rtdemo

Purpose

Robot Toolbox demonstration

Synopsis

```
rtdemo
```

Description

This script provides demonstrations for most functions within the Robotics Toolbox.

Cautionary

This script clears all variables in the workspace and deletes all figures.

showlink

Purpose

Show robot link details

Synopsis

```
showlink(robot)
showlink(link)
```

Description

Displays in detail all the parameters, including all defined inertial parameters, of a link. The first form provides this level of detail for all links in the specified manipulator. roll/pitch/yaw angles in radians.

Examples

To show details of Puma link 2

```
>> showlink(p560.link{2})
alpha  = 0
A      = 0.4318
theta  = 0
D      = 0
sigma  = 0
mdh    = 0
offset = 0
m      = 17.4
r      = -0.3638
           0.006
          0.2275
I      = 0.13           0           0
             0       0.524           0
             0           0       0.539
Jm     = 0.0002
G      = 107.815
B      = 0.000817
Tc     = 0.126      -0.071
qlim   =
>>
```

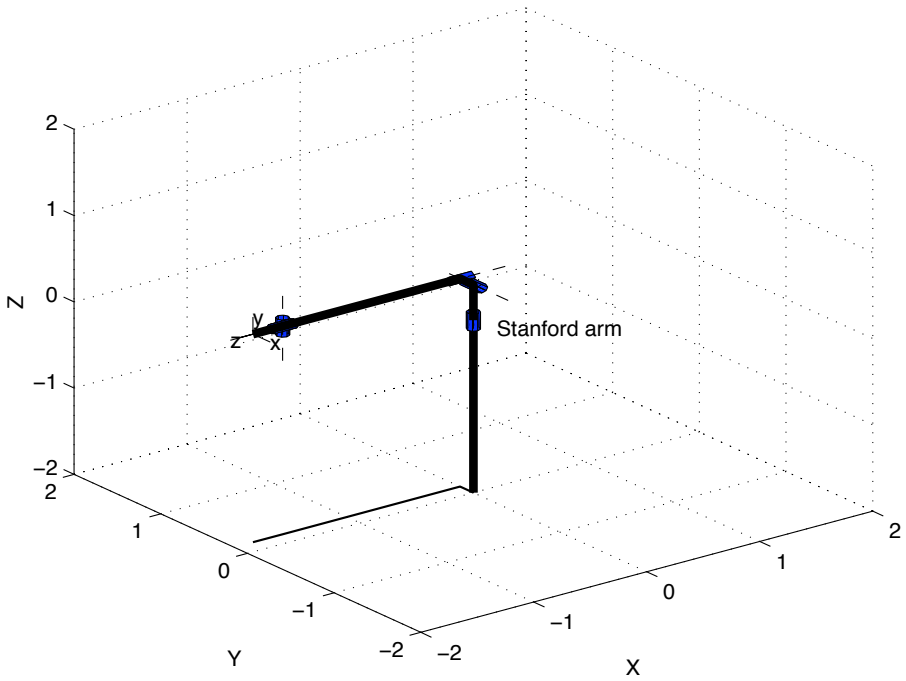
stanford

Purpose

Create a Stanford manipulator robot object

Synopsis

```
stanford
```



Description

Creates the **robot** object `stan` which describes the kinematic and dynamic characteristics of a Stanford manipulator. Specifies armature inertia and gear ratios. All quantities are in standard SI units.

See Also

robot, puma560

References

R. Paul, "Modeling, trajectory calculation and servoing of a computer controlled arm," Tech. Rep. AIM-177, Stanford University, Artificial Intelligence Laboratory, 1972.

R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. Cambridge, Massachusetts: MIT Press, 1981.

tr2diff

Purpose

Convert a homogeneous transform to a differential motion vector

Synopsis

```
d = tr2diff(T)
d = tr2diff(T1, T2)
```

Description

The first form of `tr2diff` returns a 6-element differential motion vector representing the incremental translation and rotation described by the homogeneous transform `T`. It is assumed that `T` is of the form

$$\begin{bmatrix} 0 & -\delta_z & \delta_y & d_x \\ \delta_z & 0 & -\delta_x & d_y \\ -\delta_y & \delta_x & 0 & d_z \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The translational elements of `d` are assigned directly. The rotational elements are computed from the mean of the two values that appear in the skew-symmetric matrix.

The second form of `tr2diff` returns a 6-element differential motion vector representing the displacement from `T1` to `T2`, that is, `T2` - `T1`.

$$d = \begin{bmatrix} \underline{p}_2 - \underline{p}_1 \\ 1/2\,(\underline{n}_1 \times \underline{n}_2 + \underline{o}_1 \times \underline{o}_2 + \underline{a}_1 \times \underline{a}_2) \end{bmatrix}$$

See Also

diff2tr

References

R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. Cambridge, Massachusetts: MIT Press, 1981.

tr2eul

Purpose

Convert a homogeneous transform to Euler angles

Synopsis

```
[a b c] = tr2eul(T)
```

Description

`tr2eul` returns a vector of Euler angles, in radians, corresponding to the rotational part of the homogeneous transform `T`.

$$T_{rot} = R_Z(a)R_Y(b)R_Z(c)$$

Cautionary

Note that 12 different Euler angle sets or conventions exist. The convention used here is the common one for robotics, but is not the one used for example in the aerospace community.

See Also

eul2tr, tr2rpy

References

R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. Cambridge, Massachusetts: MIT Press, 1981.

tr2jac

Purpose

Compute a Jacobian to map differential motion between frames

Synopsis

```
jac = tr2jac(T)
```

Description

`tr2jac` returns a 6×6 Jacobian matrix to map differential motions or velocities between frames related by the homogeneous transform `T`.

If `T` represents a homogeneous transformation from frame A to frame B, ${}^{A}\mathbf{T}_{B}$, then

$$ {}^{B}\underline{\dot{x}} = {}^{B}\mathbf{J}_{A}\,{}^{A}\underline{\dot{x}} $$

where ${}^{B}\mathbf{J}_{A}$ is given by `tr2jac(T)`.

See Also

tr2diff, diff2tr

References

R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. Cambridge, Massachusetts: MIT Press, 1981.

tr2rpy

Purpose

Convert a homogeneous transform to roll/pitch/yaw angles

Synopsis

```
[a b c] = tr2rpy(T)
```

Description

`tr2rpy` returns a vector of roll/pitch/yaw angles, in radians, corresponding to the rotational part of the homogeneous transform `T`

$$T_{rot} = R_Z(r)R_Y(p)R_X(y)$$

See Also

rpy2tr, tr2eul

References

R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. Cambridge, Massachusetts: MIT Press, 1981.

transl

Purpose

Translational transformation

Synopsis

```
T = transl(x, y, z)
T = transl(v)
v = transl(T)
xyz = transl(TC)
```

Description

The first two forms return a homogeneous transformation representing a translation expressed as three scalar `x`, `y` and `z`, or a Cartesian vector `v`.

The third form returns the translational part of a homogeneous transform as a 3-element column vector.

The fourth form returns a matrix whose columns are the X, Y and Z columns of the $4 \times 4 \times m$ Cartesian trajectory matrix `TC`.

See Also

ctraj

trinterp

Purpose

Interpolate homogeneous transforms

Synopsis

```
T = trinterp(T0, T1, r)
```

Description

`trinterp` interpolates between the two homogeneous transforms `T0` and `T1` as `r` varies between 0 and 1 inclusively. This is generally used for computing straight line or 'Cartesian' motion. Rotational interpolation is achieved using quaternion spherical linear interpolation.

Examples

Interpolation of homogeneous transformations.

```
>> t1=rotx(.2)

t1 =
    1.0000         0         0         0
         0    0.9801   -0.1987         0
         0    0.1987    0.9801         0
         0         0         0    1.0000

>> t2=transl(1,4,5)*roty(0.3)

t2 =
    0.9553         0    0.2955    1.0000
         0    1.0000         0    4.0000
   -0.2955         0    0.9553    5.0000
         0         0         0    1.0000

>> trinterp(t1,t2,0) % should be t1

ans =
    1.0000         0         0         0
         0    0.9801   -0.1987         0
         0    0.1987    0.9801         0
         0         0         0    1.0000

>> trinterp(t1,t2,1)   % should be t2

ans =
    0.9553         0    0.2955    1.0000
         0    1.0000         0    4.0000
   -0.2955         0    0.9553    5.0000
         0         0         0    1.0000
```

```
>> trinterp(t1,t2,0.5) % 'half way' in between

ans =
    0.9887    0.0075    0.1494    0.5000
    0.0075    0.9950   -0.0998    2.0000
   -0.1494    0.0998    0.9837    2.5000
         0         0         0    1.0000

>>
```

See Also

ctraj, qinterp

References

R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. Cambridge, Massachusetts: MIT Press, 1981.

trnorm

Purpose

Normalize a homogeneous transformation

Synopsis

```
TN = trnorm(T)
```

Description

Returns a normalized copy of the homogeneous transformation `T`. Finite word length arithmetic can lead to homogeneous transformations in which the rotational submatrix is not orthogonal, that is, $det(\mathbf{R}) \neq -1$.

Algorithm

Normalization is performed by orthogonalizing the rotation submatrix $\underline{n} = \underline{o} \times \underline{a}$.

See Also

oa2tr

References

J. Funda, "Quaternions and homogeneous transforms in robotics," Master's thesis, University of Pennsylvania, Apr. 1988.

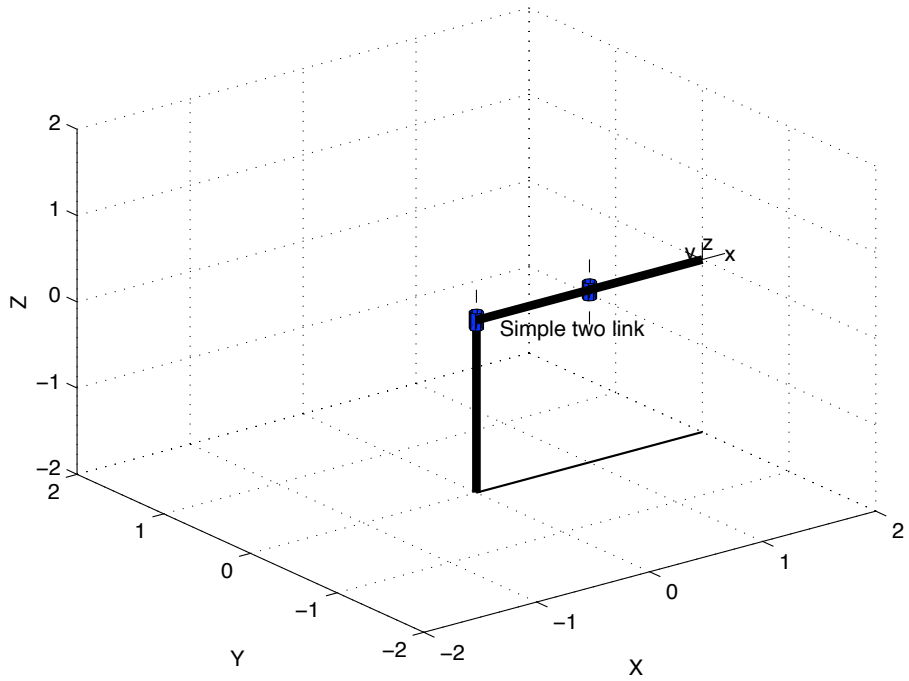
twolink

Purpose

Load kinematic and dynamic data for a simple 2-link mechanism

Synopsis

```
twolink
```



Description

Creates the **robot** object `tl` which describes the kinematic and dynamic characteristics of a simple two-link planar manipulator. The manipulator operates in the horizontal (XY) plane and is therefore not influenced by gravity.

Mass is assumed to be concentrated at the joints. All masses and lengths are unity.

See Also

puma560, stanford

References

Fig 3-6 of "Robot Dynamics and Control" by M.W. Spong and M. Vidyasagar, 1989.

unit

Purpose

Unitize a vector

Synopsis

```
vn = unit(v)
```

Description

`unit` returns a unit vector aligned with `v`.

Algorithm

$$\underline{v}_n = \frac{\underline{v}}{||\underline{v}||}$$

dh (legacy)

Purpose

Matrix representation of manipulator kinematics

Description

A `dh` matrix describes the kinematics of a manipulator in a general way using the standard Denavit-Hartenberg conventions. Each row represents one link of the manipulator and the columns are assigned according to the following table.

Column	Symbol	Description
1	α_i	link twist angle
2	A_i	link length
3	θ_i	link rotation angle
4	D_i	link offset distance
5	σ_i	joint type; 0 for revolute, non-zero for prismatic

If the last column is not given, toolbox functions assume that the manipulator is all-revolute. For an n-axis manipulator `dh` is an $n \times 4$ or $n \times 5$ matrix.

The first 5 columns of a `dyn` matrix contain the kinematic parameters and maybe used anywhere that a `dh` kinematic matrix is required — the dynamic data is ignored.

Lengths A_i and D_i may be expressed in any unit, and this choice will flow on to the units in which homogeneous transforms and Jacobians are represented. All angles are in radians.

See Also

puma560,stanford,mdh

References

R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. Cambridge, Massachusetts: MIT Press, 1981.

dyn (legacy)

Purpose

Matrix representation of manipulator kinematics and dynamics

Description

A `dyn` matrix describes the kinematics and dynamics of a manipulator in a general way using the standard Denavit-Hartenberg conventions. Each row represents one link of the manipulator and the columns are assigned according to the following table.

Column	Symbol	Description
1	α	link twist angle
2	A	link length
3	θ	link rotation angle
4	D	link offset distance
5	σ	joint type; 0 for revolute, non-zero for prismatic
6	mass	mass of the link
7	rx	link COG with respect to the link coordinate frame
8	ry	
9	rz	
10	Ixx	elements of link inertia tensor about the link COG
11	Iyy	
12	Izz	
13	Ixy	
14	Iyz	
15	Ixz	
16	Jm	armature inertia
17	G	reduction gear ratio; joint speed/link speed
18	B	viscous friction, motor referred
19	Tc+	coulomb friction (positive rotation), motor referred
20	Tc-	coulomb friction (negative rotation), motor referred

For an n-axis manipulator, `dyn` is an $n \times 20$ matrix. The first 5 columns of a `dyn` matrix contain the kinematic parameters and maybe used anywhere that a `dh` kinematic matrix is required — the dynamic data is ignored.

All angles are in radians. The choice of all other units is up to the user, and this choice will flow on to the units in which homogeneous transforms, Jacobians, inertias and torques are represented.

See Also

dh