Peter I. Corke*

A Robotics Toolbox for MATLAB

The Robotics Toolbox is a software package that allows a MATLAB user to readily create and manipulate datatypes fundamental to robotics such as homogeneous transformations, quaternions and trajectories. Functions provided, for arbitrary serial-link manipulators, include forward and inverse kinematics, Jacobians, and forward and inverse dynamics. This article introduces the Toolbox in tutorial form, with examples chosen to demonstrate a range of capabilities. The complete Toolbox and documentation is freely available via anonymous ftp.

MATLAB [4] is a powerful environment for linear algebra and graphical presentation that is available on a very wide range of computer platforms. The core functionality can be extended by application specific toolboxes. The Robotics Toolbox provides many functions that are required in robotics and addresses such areas as kinematics, dynamics, and trajectory generation. Combined with the interactive MATLAB environment and its powerful graphical functions it provides a very convenient approach to robotic simulation and experimental analysis.

The Toolbox is based on a very general method of representing the kinematics and dynamics of serial-link manipulators by "description matrices." These comprise, in the simplest case, the Denavit and Hartenberg parameters of the robot and can be created by the user for any serial-link manipulator. A number of examples are provided for well known robots such as the Puma 560 and the Stanford arm. The manipulator description can be elaborated, by augmenting the matrix, to include link inertial, and motor inertial and frictional parameters. Such matrices provide a concise means of describing a robot model. 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 generally written in a straightforward, or textbook, manner for pedagogical reasons rather than for maximum computational efficiency.

This tutorial assumes a moderate familiarity with MATLAB and begins by introducing the functions and datatypes used to represent 3-dimensional (3D) position and orientation. After a discussion of interpolation of 3D position and orientation, we introduce the general matrix representation of an arbitrary serial-link manipulator and cover kinematics; forward and inverse solutions and the manipulator Jacobians. We then consider the creation of trajectories in configuration or Cartesian space and extend the general matrix representation to include manipulator rigid-body and motor dynamics, and describe functions for forward and inverse manipulator dynamics. Finally we give a longer example to illustrate the power of the Toolbox/MATLAB combination to tackle sophisticated tasks in robotic simulation. Details on how to obtain the package are given in Section 8, and a complete list of all Toolbox functions are given in Table 3.

*Brisbane Laboratory, Division of Manufacturing Technology, Queensland Centre for Advanced Technologies, PO Box 883, Kenmore Qld 4069, Australia.
Email: pic@brh.dmt.csiro.au

1070-9932/96/$5.00©1996 IEEE

REPRESENTING 3D TRANSLATION AND ORIENTATION

In Cartesian coordinates translation may be represented by a position vector, ${}^{A}\underline{p}$, with respect to the origin of coordinate frame A and where $\underline{p} \in \Re^3$. If A is not given the world coordinate frame is assumed. Many representations of 3D orientation have been proposed [3] but the most commonly used in robotics are orthonormal rotation matrices and unit-quaternions. The homogenous transformation is a 4 x 4 matrix which represents translation and orientation and can be compounded simply by matrix multiplication. Such a matrix representation is well matched to MATLAB's powerful capability for matrix manipulation. Homogenous transformations

$$T = \begin{bmatrix} \mathrm{R} & \underline{p} \\ 000 & 1 \end{bmatrix}$$

describe the relationships between Cartesian coordinate frames in terms of a Cartesian translation, $\underline{p}$,and orientation expressed as a 3 x 3 orthonormal rotation matrix, `R` . A homogeneous transformation representing a pure translation of 0.5 m in the X-direction is created by

```
>> T = transl(0.5, 0.0, 0.0);
```

and a rotation of 90 deg about the Y-axis by

```
>> T = roty(pi/2);
```

Such transforms may be concatenated by multiplication, for instance,

```
>> T = transl(0.5, 0.0, 0.0) * roty(pi/2) * rotz(-pi/2)
 T =
    0.0000    0.0000    1.0000    0.5000
    1.0000    0.0000         0         0
    0.0000   -1.0000    0.0000         0
         0         0         0    1.0000
```

The orientation of the new coordinate frame may be expressed in terms of Euler angles

```
>> tr2eul(T)
ans = 0    1.5708   -1.5708
```

in units of radians, or roll/pitch/yaw angles

```
>> tr2rpy(T)
ans = -1.5708    0.0000   -1.5708
```

Homogeneous transforms can be generated from Euler or roll/pitch/yaw angles, or rotation about an arbitrary vector using the functions `eul2tr()`, `rpy2tr()`, and `rotvec()` respectively.

Rotation can also be represented by a quaternion, which will be denoted here by

$$q = [s, \underline{v}]$$

where s is a scalar and $\underline{v} \in \Re^3$. A unit-quaternion has unit magnitude, that is, $s^2 + |\underline{v}|^2 = 1$ in which case $s = \sin\theta/2$, and q can be considered as a rotation of θ about the vector $\underline{v}$. The rotational component of a homogeneous transform can be converted to a unit-quaternion

```
>> q = tr2q(T)
q =  0.5000   -0.5000    0.5000   -0.5000
```

which indicates that the compounded transformation is equivalent to a rotation of 0.5 rad about the vector [-1 1 -1]. Quaternions can be compounded ("multiplied") by the function `qmul()`.

INTERPOLATION

Frequently in robotics it is necessary to interpolate between two positions or orientations. In the case where these are represented by homogeneous transforms the function

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

is used where r is a scalar such that $0 \le r \le 1$. When `r = 0` the interpolation returns `T0` and for `r = 1` returns `T1`. Values of `r` between these extremes return a "blend" of the rotational and translational components of the two transformations. Blending of rotations is considerably simpler to implement for the case of quaternions

```
>> q = qinterp(q0, q1, r)
```

KINEMATICS

Forward kinematics is the problem of solving the Cartesian position and orientation of the end-effector given knowledge of the kinematic structure and the joint coordinates. The kinematic structure of a serial-link manipulator can be succinctly described in terms of Denavit-Hartenberg parameters [6] or modified Denavit-Hartenberg parameters [2]. Within the Toolbox the manipulator's kinematics are represented in a general way by a dh matrix which is given as the first argument to Toolbox kinematic functions. The `dh` matrix describes the kinematics of a manipulator using the standard Denavit-Hartenberg conventions, where each row represents one link of the manipulator and the columns are assigned according to Table 1.

If the last column is not given, Toolbox functions assume that the manipulator is all-revolute. For an n-axis manipulator `dh` is thus an n x 4 or n x 5 matrix. Joint angles are repre-

Table 1. Column assignments for the Toolbox `dh` matrix.

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

sented by n-element row vectors. Consider the example of a Puma 560 manipulator, a common laboratory robot. The kinematics may be defined by the `puma560` command which creates a kinematic description matrix p560 in the workspace using standard Denavit-Hartenberg conventions, and the particular frame assignments of Paul and Zhang [7]. It also creates workspace variables that define special joint angle poses: `qz` for all zero joint angles, `qr` for the "READY" position and `qstretch` for a fully extended arm horizontal pose.

The forward kinematics may be computed for the zero angle pose

```
>> puma560   % define puma kinematic matrix p560
>> fkine(p560, qz)
ans =
    1.0000         0         0    0.4521
         0    1.0000         0   -0.1254
         0         0    1.0000    0.4318
         0         0         0    1.0000
```

which returns the homogeneous transform corresponding to the pose of the last link of the manipulator. The translation, given by the last column, is in the same dimensional units as the A_i and D_i data in the `dh` matrix, in this case metres. This pose can be visualised by

```
>> plotbot(p560, qz);
```

which produces the 3-D plot shown in Figure 1. The drawn line segments do not necessarily correspond to robot links, but join the origins of sequential link coordinate frames. Such an approach eliminates the need for additional, detailed, robot geometry data. A small right-handed coordinate frame is drawn on the end of the robot to show the wrist orientation. The X-, Y- and Z-axes are represented by the colours red, green and blue respectively.

Inverse kinematics is the problem of finding the robot joint coordinates, given a homogeneous transform representing the pose of the end-effector. It is very useful when the path is planned in Cartesian space, for instance a straight line

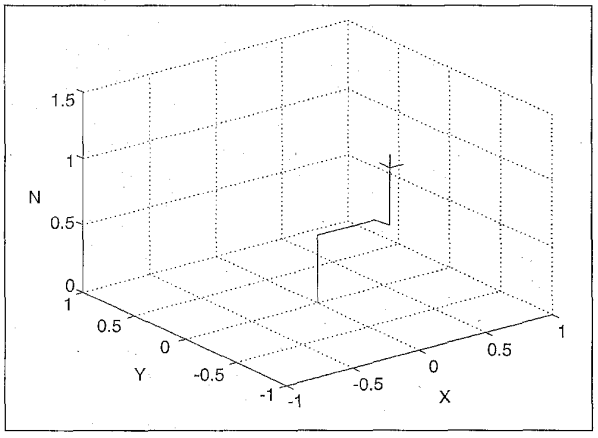


Figure 1. Visualisation of Puma robot at zero joint angle pose — created by `plotbot(p560, qz)`.

path as shown later. First generate the transform corresponding to a particular joint coordinate,

```
>> q = [0 -pi/4 -pi/4 0 pi/8 0]
q =
         0   -0.7854   -0.7854         0    0.3927         0
>> T = fkine(p560, q);
```

and now recompute the original joint angles

```
>> qi = ikine(p560, T)
qi =
    0.0000   -0.7854   -0.7854    0.0000    0.3927    0.0000
```

which compares well with the original value.

The inverse kinematic procedure for any specific robot can be derived symbolically [6] and commonly an efficient closed-form solution can be obtained. However the Toolbox is given only a generalised description of the manipulator in terms of kinematic parameters so an iterative numerical solution [1] must be used. Such a procedure can be slow, and the choice of starting value affects both the search time and the solution found, since in general a manipulator may have several configurations which result in the same pose for the last link. The starting point for the solution may be specified, or else it defaults to zero (which is not a particularly good choice in this case), and provides limited control over the particular solution that will be found. For a redundant manipulator a solution will be found but there is no explicit control over the null-space. For a manipulator with $n < 6$ DOF an additional argument is required to indicate which of the 6-n Cartesian DOF are to be unconstrained in the solution. Limited functions are provided for kinematic operations based on the modified Denavit and Hartenberg notation introduced by Craig [2]. Such functions are prefixed by an "m," for instance `mfkine()`.

Differential Cartesian motion, or velocity, can be represented by a 6-element vector whose first 3 elements are a differential translation, and the last 3 are a differential rotation. The corresponding homogeneous transform could be evaluated in terms of compounded transforms but a more direct approach is to use the function `diff2tr()`

```
>> D = [.1 .2 0 -.2 .1 .1]';
>> diff2tr(D)
ans =
         0   -0.1000    0.1000    0.1000
    0.1000         0    0.2000    0.2000
    0.1000   -0.2000         0         0
         0         0         0         0
```

which yields the matrix referred to by Paul [6] as Δ, which has a skew symmetric rotation submatrix. More commonly it is useful to know how a differential motion, or velocity, in one coordinate frame appears in another frame. If the second frame is represented by the homogeneous transform

```
>> T = transl(100, 200, 300) * roty(pi/8) * rotz(-pi/4);
```

with respect to the first, then the differential motion in the second frame would be given by

```
>> DT = tr2jac(T) * D;
>> DT'
ans =  -29.5109  69.7669 -42.3289  -0.2284  -0.0870   0.0159
```

`tr2jac()` has computed a 6 x 6 Jacobian matrix which transforms the differential changes from the first frame to the next.

The manipulator's Jacobian matrix, J_q, maps differential motion or velocity between configuration and Cartesian space. For an n-axis manipulator the end-effector Cartesian velocity is

$$ {}^0\dot{\underline{x}}_n = {}^0J_q(\underline{q})\dot{\underline{q}} $$

$$ {}^{Tn}\dot{\underline{x}}_n = {}^{Tn}J_q(\underline{q})\dot{\underline{q}} $$

in base or end-effector coordinates respectively and where $\dot{\underline{x}}$ is the Cartesian velocity represented by a 6-vector as above. The two Jacobians are computed by the Toolbox functions `jacob0()` and `jacobn()` respectively. For an n-axis manipulator the Jacobian is a 6 x n matrix. In the end-effector frame

```
>> q = [0.1 0.75 -2.25 0 .75 0];
>> J = jacobn(p560, q)
   J =
    0.0918   -0.7328   -0.3021         0         0         0
    0.7481    0.0000    0.0000         0         0         0
    0.0855    0.3397    0.3092         0         0         0
    0.6816         0         0    0.6816         0         0
    0.0000   -1.0000   -1.0000   -0.0000   -1.0000         0
    0.7317    0.0000    0.0000    0.7317    0.0000    1.0000
```

Note the top right 3 x 3 block is all zero and indicates, correctly, that motion of joints 4 to 6 does not cause any translational motion of the robot's end-effector — a characteristic of the spherical wrist. Many control schemes require the inverse of the Jacobian, which in this example is not singular

```
>> det(J)
ans = -0.0632
```

and may be inverted

```
>> Ji = inv(J)
```

One such control scheme is resolved rate motion control proposed by Whitney [9]

$$ \dot{\underline{q}} = {}^0J_q^{-1}\,{}^0\dot{\underline{x}}_n $$

which gives the joint velocities required to achieve the desired Cartesian velocity. In this example, in order to achieve 0.1 m/s translational motion in the end-effector X-direction the required joint velocities would be

```
>> vel = [0.1 0 0 0 0 0]';  % translation in the X direction
>> qvel = Ji * vel;
>> qvel'
ans =  0.0000   -0.2495   0.2741   0.0000   -0.0246   0.0000
```

which requires approximately equal and opposite motion of the shoulder and elbow joints. At a kinematic singularity the Jacobian becomes singular, and such simple control techniques will fail. As already discussed, at the Puma's "READY" position two of the wrist joints are aligned resulting in the loss of one degree of freedom. This is revealed by the rank of the Jacobian

```
>> rank( jacobn(p560, qr) )
ans =  5
```

which is less than that needed for independent motion along each Cartesian degree of freedom. The null space of this Jacobian is

```
>> null(J)'
ans =   0.0000   0.0000   0.0000   -0.7071   0.0000   0.7071
```

which indicates that equal and opposite motion of joints 4 and 6 will result in zero motion of the end-effector.

When not actually at a singularity the condition of the Jacobian can provide information about how "well-positioned" the manipulator is for making certain motions. This is referred to as "manipulability" and is computed by the `maniplty()` function.

TRAJECTORIES

As we have seen, homogeneous transforms or unit-quaternions can be used to represent the pose, position and orientation, of a coordinate frame in Cartesian space. In robotics we frequently need to deal with paths or trajectories, that is, a sequence of Cartesian frames or joint angles. Consider the example of a path which will move the Puma robot from its zero angle pose, `qz` to the upright (or READY) pose, `qr`. First create a time vector, completing the motion in 2 seconds with a sample interval of 56 ms.

```
>> t = [0:.056:2]';
```

and then compute a joint space trajectory

```
>> q = jtraj(qz, qr, t);
```

`q` is a matrix with one row per time step, and each row a joint angle vector as above. The trajectory is a fifth order polynomial which has continuous jerk. By default, the initial and final velocities are zero, but these may be specified by additional arguments. For this particular trajectory most of the motion is done by joints 2 and 3, and this can be conveniently plotted using standard MATLAB plotting commands to give Figure 2. We can also look at the velocity and acceleration profiles. We could differentiate the angle trajectory using `diff()`, but more accurate results can be obtained by requesting that `jtraj()` return angular velocity and acceleration as follows

```
>> [q,qd,qdd] = jtraj(qz, qr, t);
```

A number of Toolbox functions, such as forward kinematics, can operate on trajectories. The homogeneous transform

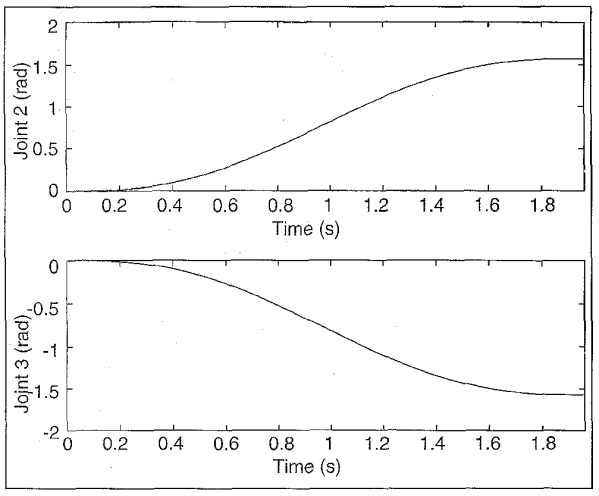


Figure 2. Joint space trajectory generated by jtraj().

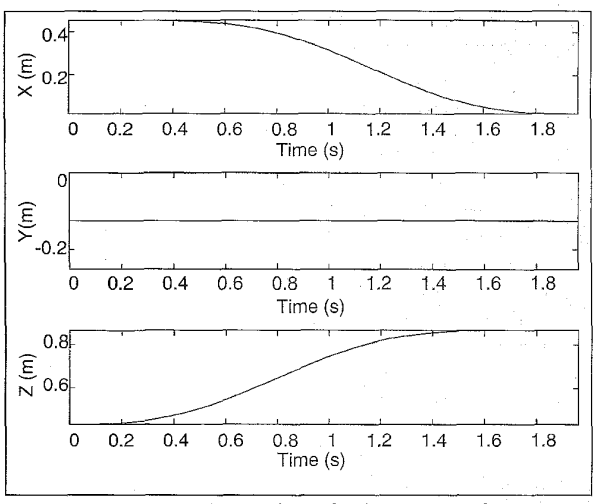


Figure 3. Cartesian coordinates of wrist for the trajectory of Figure 2.

for each point of the trajectory is given by

```
>> Tq = fkine(p560, q);
```

Since MATLAB does not (yet) support 3-dimensional matrices, each row of `Tq` is a "flattened" (in column-major order) 4 x 4 homogeneous transform corresponding to the equivalent row of `q`. The "flattened" transform can be restored by means of the `ttg()` function — for example the tenth point is

```
>> ttg(Tq, 10)
ans =
    1.0000    0.0000         0    0.4455
    0.0000    1.0000    0.0000   -0.1254
         0    0.0000    1.0000    0.5068
         0         0         0    1.0000
```

Columns 13, 14 and 15 of `Tq` correspond to the X-, Y- and Z- coordinates respectively, and can be plotted against time to give Figure 3. Alternatively we could plot X against Z to get some idea of the Cartesian path followed by the manipulator. The function `plotbot()` introduced above, will when invoked on a trajectory, display a stick figure animation of the robot moving along the path

```
>> plotbot(p560, q);
```

Straight line, or "Cartesian," paths can be generated in a similar way between two points specified by homogeneous transforms.

```
>> t = [0:.056:2];              % create a time vector
>> T0 = transl(0.6, -0.5, 0.0);% define the start point
>> T1 = transl(0.4, 0.5, 0.2); % and destination
>> Ts = ctraj(T0, T1, t);      % compute a Cartesian path
```

The resulting trajectory, `Ts`, has one row per time step and that row is again a "flattened" homogeneous transform. The function `ctraj()` makes use of the interpolation function `trinterp()` introduced earlier. Inverse kinematics can then be applied to determine the corresponding joint angle motion using

```
>> tic, qc = ikine(p560, Ts); toc
elapsed_time =
53.7800              % on a 33MHz 486DX
```

When solving for a trajectory, the starting joint coordinates for each inverse kinematic solution are taken as the result of the previous solution. Once again the joint space trajectory could be plotted against time or animated using `plotbot()`. Clearly this approach is slow, and would not be suitable be for a real robot controller where an inverse kinematic solution would be required every few milliseconds.

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 are well covered in the literature. The equations of motion for an n-axis manipulator are given by

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

where $\underline{q}$ is the vector of generalised joint coordinates describing the pose of the manipulator, $\dot{\underline{q}}$ the vector of joint velocities, $\ddot{\underline{q}}$ the vector of joint accelerations, M the symmetric joint-space inertia matrix, C the Coriolis and centripetal torques, F the viscous and Coulomb friction, G the gravity loading, and $\underline{Q}$ is the vector of generalised forces associated with the generalised coordinates $\underline{q}$. Within the Toolbox the manipulator's kinematics and dynamics are represented in a general way by a `dyn` matrix which is given as the first argument to Toolbox dynamic functions. Each row represents one link of the manipulator and the columns are assigned according to Tables 1 and 2. The `dyn` matrix is in fact a `dh` matrix augment-

Table 2: Column assignments for the Toolbox `dyn` matrix. Rows 1-5 are as per Table 1.

Column	Symbol	Description
6	m	link mass
7	r_x	link COG with respect to the link coordinate frame
8	r_y	
9	r_x	
10	I_{xx}	elements of link inertia tensor about the link COG
11	I_{yy}	
12	I_{zz}	
13	I_{xy}	
14	I_{yz}	
15	I_{xz}	
16	J_m	armature inertia
17	G	reduction gear ratio; joint speed/link speed
18	B	viscous friction, motor referred
19	τ_c^+	Coulomb friction (positive rotation), motor referred
20	τ_c^-	Coulomb friction (negative rotation), motor referred

ed with additional columns for the inertial and mass parameters of each link, as well as the motor inertia and friction parameters. Such a definition allows a `dyn` matrix to be passed to any Toolbox function in place of a `dh` matrix but not vice versa. For an n-axis manipulator `dyn` is an n x 20 matrix. This structure does not yet allow for joint cross-coupling, as found in the Puma robot's wrist, joint angle limits, or motor electrical parameters such as torque constant and driver current or voltage limits.

Inverse dynamics computes the joint torques required to achieve the specified state of joint position, velocity and acceleration. The recursive Newton-Euler formulation is an efficient matrix oriented algorithm for computing the inverse dynamics, and is implemented by the Toolbox function `rne()`. Using the joint space trajectory from the trajectory example above (Figure 2) the required joint torques can be computed for each point of the trajectory by

```
>> tau = rne(p560, qd, qdd);
```

in units of Nm. As with all Toolbox functions the result has one row per time step, and each result row is a joint torque vector. Joint torque for the base axes is plotted as a function of time in Figure 4. Much of the torque on joints 2 and 3 of a Puma 560 (mounted conventionally) is due to gravity. That component can be computed separately

```
>> tau_g = gravload(p560, q);
>> plot(t, taug(:,1:3))
```

and is plotted as the dashed lines in Figure 4. The torque component due to velocity terms, Coriolis and centripetal

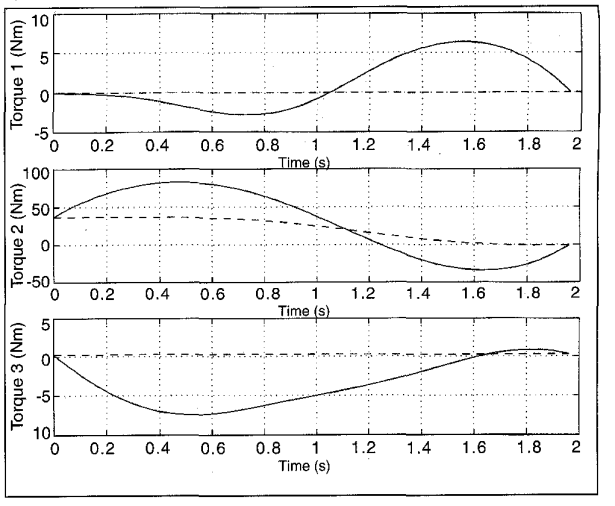


Figure 4. Joint torques for the joint space trajectory example of Figure 2.

torques, can be computed separately by the `coriolis()` function.

The `inertia()` function computes the manipulator inertia matrix, $M(q)$, for a given configuration. The next example will plot the inertia "seen" by joint 1, that is M_{11}, as the manipulator moves along the trajectory

```
>> M11 = [];
>> for qi = q',% for each row of q
       M = inertia(p560, qi');
       M11 = [M11; M(1,1)];
>> end
>> plot(t, M11);
```

and the result is shown in Figure 5. Clearly the inertia seen by joint 1 varies considerably over this path. This vividly demonstrates the challenge of control design in robotics, achieving stability and high-performance in the face of significant plant

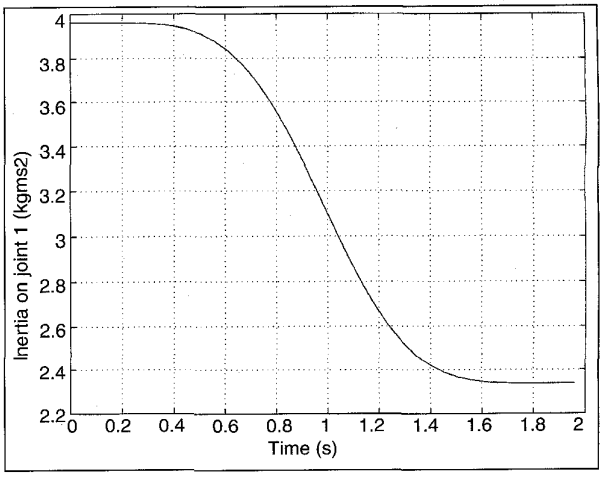


Figure 5. Inertia of joint 1, M_{11}, as the manipulator moves along the trajectory given in Figure 2.

parameter variation. In this example the inertia varies by a factor of

```
>> max(M11)/min(M11)
ans = 1.6947
```

over the path chosen.

Forward dynamics is the computation of joint accelerations given position and velocity state, and applied actuator torques and is particularly useful in simulation of robot control systems. The Toolbox function `accel()` uses Method 1 of Walker and Orin [8] which in turn uses repeated calls to the inverse dynamics function `rne()`. To be useful for simulation such a function must be integrated, and this is achieved by the Toolbox function `fdyn()` which uses the MATLAB function `ode45()`. It also allows for a user written function to return the applied joint torque as a function of manipulator state and this can be used to model arbitrary axis control strategies — if not specified zero torques are applied. To simulate the motion of the Puma 560 from rest in the zero angle pose with zero applied joint torques

```
>> tic
>> [t q qd] = fdyn(p560, 0, 2);
elapsed_time =
1.6968e+003       % on a 33MHz 486DX
```

The resulting motion is plotted versus time in Figure 6 which shows that the robot is collapsing under gravity. An animation using `plotbot()` clearly depicts this. It is interesting to note that rotational velocity of the upper and lower arm links result in centripetal and Coriolis torques on the waist joint, causing it to rotate. This simulation takes nearly half an hour to execute on a reasonable PC and is due to the very large number of calls to the `rne()` function (Ideally the `rne()` function should be implemented by a computationally more efficient MEX file.)

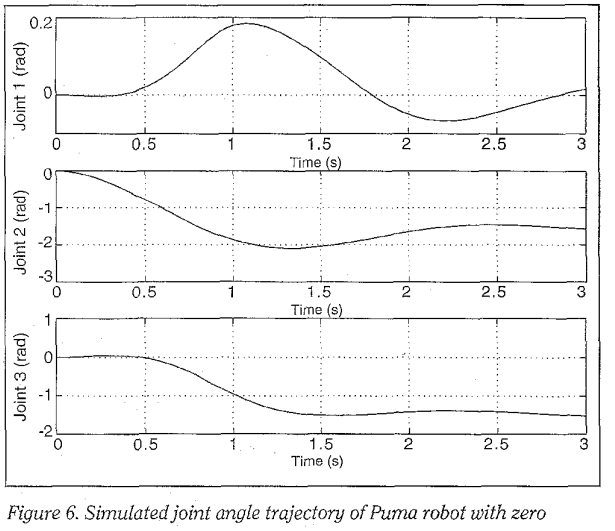


Figure 6. Simulated joint angle trajectory of Puma robot with zero applied joint torque collapsing under gravity.

A MORE COMPLEX EXAMPLE

This section presents a more complex example, showing how the view of an object from a moving camera can easily be simulated using the combined functionality of MATLAB and the Robotics Toolbox. It is assumed that the camera is attached to the end-effector, but rotated by -90 deg about the wrist's X-axis.

```
1   figure(gcf)      % bring the figure to the top
2   axis([-256 256 -256 256])      % create 512 x 512
    pixel image
3   grid
4   xlabel('X (pixels)')
5   ylabel('Y (pixels)')
6   set(gca, 'drawmode', 'fast');
7   h = line('XData', 0, 'YData', 0, ...
8        'LineStyle', '*', 'EraseMode', 'xor');
9
10  f = 12e-3;           % camera focal length in m
11  alphax = 79.2e3      % pix/m for Pulnix TM-6 +
    Digimax
12  alphay = 120.5e3     % pix/m for Pulnix TM-6 +
    Digimax
13  % camera calibration matrix
14  C = [ alphax 0 0 0; 0 alphay 0 0; 0 0 1 0] * ...
15    [ 1 0 0 0; 0 1 0 0; 0 0 -1/f 1; 0 0 0 1];
16
17  Tobj = transl(.5, 3, .5);     % location of object
    in world
18  for tt = Ttg'
19  T6 = reshape(tt', 4, 4); % a trajectory point T6
20  Tcam = T6 * rotx(-pi/2); % camera mounting
    transform
21
22  Tobj_cam = inv(Tcam) * Tobj;  % object wrt camera
23
24  x = C * Tobj_cam(:,4);        % camera transform
25  x = x(1)/x(3); Y = x(2)/x(3); % homogeneous coords
26
27  set(h, 'xdata', X, ydata', Y);   % move the marker
28  drawnow     % flush graphics to screen
29  end
```

Lines 1-8 create a graphical plot which represents the image plane of the camera in units of pixels. The object is represented by an asterisk. Lines 10-15 create a camera transformation, or camera calibration matrix, from known fundamental camera properties such as pixel pitch and lens focal length. The parameters used are those for a Pulnix TM-6 camera with a Datacube digitiser and framestore. The robot trajectory used is that from the example in Section 5. The location of the object in this example, set in line 17, is constant. For each trajectory point, the position of the object with respect to the camera is computed, line 22, and the image plane coordinates determined at line 24. The homogeneous image plane coordinates are calculated in line 25, and used to update the position of the object on the plot. Figure 7 shows the path of the object on the image plane as the robot mounted camera moves along the trajectory.

Table 3: List of all Robotics Toolbox functions.

Homogeneous Transforms	
eul2tr	Euler angle to homogeneous transform
oa2tr	orientation and approach vector to homogeneous transform
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
tr2rpy	homogeneous transform to roll/pitch/yaw angles
trnorm	normalise a homogeneous transform
transl	set/retrieve translational component of a transform
Quaternions	
q2tr	quaternion to homogeneous transform
qinv	inverse of quaternion
qnorm	normalise a quaternion
qqmul	multiply (compound) quaternions
qvmul	multiply vector by quaternion
qinterp	interpolate quaternions
tr2q	homogeneous transform to unit-quaternion
Kinematics	
dh	Denavit-Hartenberg conventions
diff2tr	differential motion vector to transform
fkine	compute forward kinematics
ikine	compute inverse kinematics
ikine560	specialized inverse kinematics for Puma 560
jacob0	compute Jacobian in base coordinate frame
jacobn	compute Jacobian in end-effector coordinate frame
linktrans	compute a link transform homogeneous transform
mdh	modified Denavit-Hartenberg conventions
mfkine	compute forward kinematics (modified Denavit-Hartenberg)
mlinktrans	compute a link transform homogeneous transform (modified Denavit-Hartenberg)
tr2diff	homogeneous transform to differential motion vector
tr2jac	homogeneous transform to Jacobian
Dynamics	
cinertia	compute normalised manipulator inertia matrix
coriolis	compute centripetal/coriolis torque
dyn	dynamics conventions
friction	joint friction
gravload	compute gravity loading
inertia	compute manipulator inertia matrix
itorque	compute inertia torque
rne	inverse dynamics
Manipulator Models	
puma560	Puma 560 data
stanford	Stanford arm data
Trajectory Generation	
ctraj	Cartesian trajectory
drivepar	Cartesian trajectory parameters
jtraj	joint space trajectory
trinterp	interpolate homogeneous transforms
ttg	retrieve n'th transform from trajectory
Graphics	
plotbot	animate robot
Other	
plotbot	vector cross product
maniplty	compute manipulability
rtdemo	toolbox demonstration
unit	unitize a vector

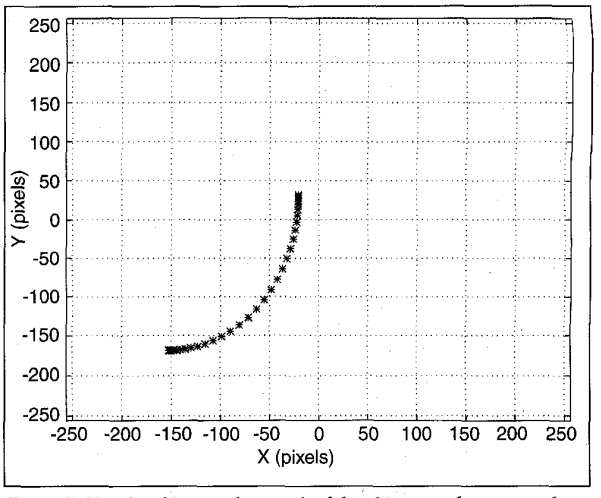


Figure 7. Simulated image plane path of the object as robot moves along path.

CONCLUSION

This short article has demonstrated, in tutorial form, the principle features of the Robotics Toolbox for MATLAB. The Toolbox provides many of the essential tools necessary for robotic modelling and simulation, as well as analysing experimental results or teaching. A key feature is the use of a single matrix to completely describe the kinematics and dynamics of any serial-link manipulator. The Robotics Toolbox is freely available from the MathWorks FTP server `ftp.mathworks.com` in the directory `pub/contrib/misc/robot`. It is best to download all files in that directory since the Toolbox functions are quite interdependent. The file `robot.ps` 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`.

The Toolbox has been developed and tested with MATLAB v4.x under Windows, SunOS and Solaris. The Toolbox does not function under MATLAB v3.x due to the significant changes introduced between MATLAB versions. Problems have also been encountered with the Student edition, particularly the trajectory examples, since these require matrices larger than the limits imposed. The public domain MATLAB clone Octave (available via anonymous ftp from `ftp.che.utexas.edu` in the directory `/pub/octave`) has also been tested. A large number of functions work without change, some need small modifications to accommodate differences. Octave compatibility will be one area of future work.

COMPARISON WITH OTHER PACKAGES

Unlike the commercial robot simulation packages, the Robotics Toolbox is targeted toward research and education needs. Another well known software package addressing this area is Robotica [5]. While both packages share the same overall goals of making robot analysis easier to perform, there are a number of significant differences in both fundamental approach and detail. Robotica makes use of the mathematical software package Mathematica, rather than MATLAB, as well

as Simnon and a C language Open-Look front end — the Toolbox needs only MATLAB for all mathematical and graphical requirements. Robotica is able to generate symbolic solutions for manipulator kinematics and dynamics, whereas the Toolbox uses numeric solutions. Symbolic solutions may provide greater insight into the problem under study, and more efficient execution of kinematic and dynamic functions. The recently released Symbolic Toolbox for MATLAB would allow similar symbolic techniques to be integrated with the Robotics Toolbox.

REFERENCES

[1] S. Chieaverini, L. Sciavicco, and B. Siciliano. Control of robotic systems through singularities. In C. Canudas de Wit, editor, Proc. Int. Workshop on Nonlinear and Adaptive Control: Issues in Robotics. Springer-Verlag, 1991.

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

[3] J. Funda, R.H. Taylor, and R.P. Paul. On homogeneous transforms, quaternions, and computational efficiency. IEEE Trans. Robot. Autom., 6(3):382{388, June 1990.

[4] The MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760. Matlab User's Guide, January 1990.

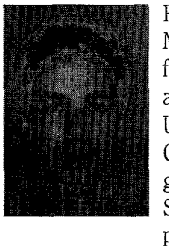
[5] J.F. Nethery and M. W. Spong. Robotica: a Mathematica package for robot analysis. IEEE Robotics and Automation magazine, 1(1):13-20, March 1994.

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

[7] R. P. Paul and Hong Zhang. Computationally efficient kinematics for manipulators with spherical wrists. Int. J. Robot. Res., 5(2):32-44, 1986.

[8] 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.

[9] D.E. Whitney and D. M. Gorinevskii. The mathematics of coordinated control of prosthetic arms and manipulators. ASME Journal of Dynamic Systems, Measurement and Control, 20(4):303-309, 1972.



Peter Corke received the B.Eng. and M.Eng.Sc degrees in Electrical Engineering from the University of Melbourne in 1981 and 1986 respectively. After teaching at that University from 1982-83, he joined the CSIRO Division of Manufacturing Technology, where he is now a Principal Research Scientist within the Integrated Automation program. In 1998-89 he spent a year as a CSIRO overseas fellow at the GRASP laboratory, University of Pennsylvania. At CSIRO he has been involved in a number of projects spanning robot control architectures, force controlled deburring, high-speed machine vision, and vision applications. He has recently completed a PhD at the University of Melbourne on the topic of high-bandwidth visual servoing.