A computer tool for simulation and analysis: the Robotics Toolbox for MATLAB

Peter I. Corke
CSIRO Division of Manufacturing Technology
pic@mlb.dmt.csiro.au

Abstract. This paper introduces, in tutorial form, a Robotics Toolbox for MATLAB that allows the user to easily 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, and forward and inverse dynamics. The complete Toolbox and documentation is freely available via anonymous ftp.

1 Introduction

MATLAB[1] 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 areas such as kinematics, dynamics, and trajectory generation. The Toolbox is useful for simulation as well as analyzing results from experiments with real robots, and can be a powerful tool for education.

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[2] 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 and may facilitate the sharing of robot models across the research community. This would allow simulation results to be compared in a much more meaningful way than is currently done in the literature. The Toolbox also provides functions for manipulating 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 paper is written in a tutorial form and does not require detailed knowledge of MATLAB. The examples illustrate both the principal features of the Toolbox and some elementary robotic theory. Section 2 begins by introducing the functions and datatypes used to represent 3-dimensional (3D) position and orientation. Section 3 introduces the general matrix representation of an arbitrary serial-link manipulator and covers kinematics; forward and inverse solutions and the manipulator Jacobians. Section 4 is concerned with the creation of trajectories in configuration or Cartesian space. Section 5 extends the general matrix representation to include manipulator rigid-body and motor dynamics, and describes functions for forward and inverse manipulator dynamics. Details on how to obtain the package are given in Section 6.

2 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 homogeneous transformation is a 4×4 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. Homogeneous transformations

$$\mathbf{T} = \left[\begin{array}{cc} \mathbf{R} & \underline{p} \\ 0\,0\,0 & 1 \end{array} \right] \tag{1}$$

describe the relationships between Cartesian coordinate frames in terms of a Cartesian translation, $\underline{p}$, and orientation. expressed as an orthonormal rotation matrix, $\mathbf{R}$ is a 3×3. 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)
T =
    1.0000         0         0    0.5000
         0    1.0000         0         0
         0         0    1.0000         0
         0         0         0    1.0000
```

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

```
>> T = roty(pi/2)
T =
    0.0000         0    1.0000         0
         0    1.0000         0         0
   -1.0000         0    0.0000         0
         0         0         0    1.0000
```

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 resulting transformation may be interpreted as a new coordinate frame whose X-, Y- and Z-axes are parallel to unit vectors given by the first three columns of `T`. That is, the new X-axis is anti-parallel to the world Y-axis, and so on. 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

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

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

Rotation can also be represented by a quaternion[3], which will be denoted here by

$$\mathbf{q} = [s, \underline{v}] \tag{2}$$

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 $\mathbf{q}$ 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 $0.5\,\mathrm{rad}$ about the vector $[-1\ 1\ -1]$. Quaternions can be compounded (‘multiplied’) by the function `qmul()`. Quaternions offer several advantages over homogeneous transformations such as reduced arithmetic cost when compounding rotations, simpler rotational interpolation, and less need for normalization to counter the effects of numerical roundoff. To represent translation as well as rotation a quaternion/vector pair can be employed[3] but such a composite type is not yet supported by this Toolbox.

3 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[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 the following table:

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

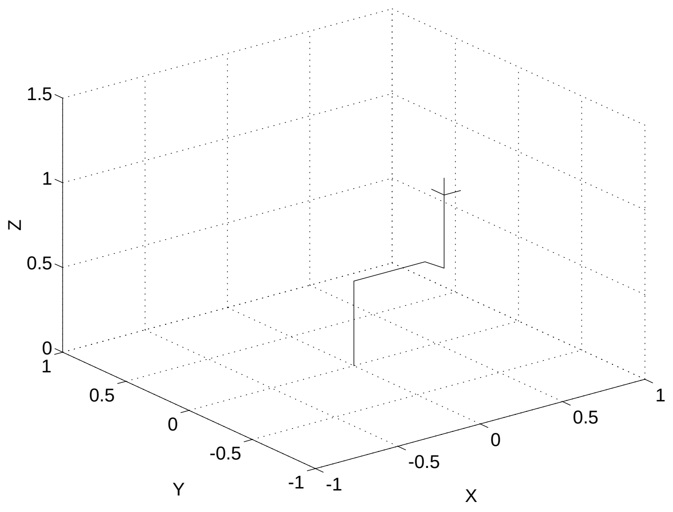

Figure 1: Visualization of Puma robot at zero joint angle pose — created by `plotbot(p560, qz)`.

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. Joint angles are represented 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[4]. 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 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 visualized 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. This simple approach eliminates the need for 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 colors 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 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 then find the corresponding joint angles using `ikine()`

```
>> 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[2] and commonly an efficient closed-form solution can be obtained. However the Toolbox is given only a generalized description of the manipulator in terms of kinematic parameters so an iterative numerical solution 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 poses which result in the same transform 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. Note that a solution is not possible if the specified transform describes a point out of reach of the manipulator — in such a case the function will return with an error.

The manipulator's Jacobian matrix, $\mathbf{J}_q$, maps differential motion or velocity between configuration and Cartesian space. For an n-axis manipulator the end-effector Cartesian velocity is

$$^{0}\dot{\underline{x}}_n = {}^{0}\mathbf{J}_q(\underline{q})\,\dot{\underline{q}} \tag{3}$$

$$^{T_n}\dot{\underline{x}}_n = {}^{T_n}\mathbf{J}_q(\underline{q})\,\dot{\underline{q}} \tag{4}$$

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 $6 \times n$ matrix.

```
>> q = [0.1 0.75 -2.25 0 .75 0];
>> J = jacob0(p560, q)
J =
    0.0501   -0.3031   -0.0102         0         0         0
    0.7569   -0.0304   -0.0010         0         0         0
    0.0000    0.7481    0.4322         0         0         0
    0.0000    0.0998    0.0998    0.9925    0.0998    0.6782
    0.0000   -0.9950   -0.9950    0.0996   -0.9950    0.0681
    1.0000    0.0000    0.0000    0.0707    0.0000    0.7317
```

or in the end-effector coordinate frame

```
>> 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 3×3 block is all zero. This 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. One such control scheme is resolved rate motion control proposed by Whitney[5]

$$\underline{\dot{q}} = {}^0\mathbf{J}_q^{-1}\, {}^0\underline{\dot{x}}_n \qquad (5)$$

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

```
>> vel = [0.1 0 0 0 0 0]'; % xlational motion in X directn
>> qvel = inv(J) * 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

```
>> n = null(J);
>> n'
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, and is referred to as ‘manipulability’. A number of scalar manipulability measures have been proposed. One by Yoshikawa

```
>> maniplty(p560(:,1:5), q)
ans =     0.0632
```

is based purely on kinematic parameters of the manipulator and would indicate, in this case, that the pose is not well conditioned.

4 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 to the upright (or READY) pose. 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 acceleration and 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

```
>> subplot(2,1,1); plot(t,q(:,2))
>> subplot(2,1,2); plot(t,q(:,3))
```

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);
```

which can then be plotted as before to give Figure 2.

A number of Toolbox functions can operate on trajectories, for instance forward kinematics. The homogeneous transform for each point of the trajectory is given by

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

Since MATLAB does not yet support 3-dimensional matrices, each row of `Ttg` is a ‘flattened’ 4x4 homogeneous transform corresponding to the equivalent row of `q`, which can be restored by means of the `reshape` function. Columns 13, 14 and 15 of `T` correspond to the X-, Y- and Z- coordinates respectively, and could be plotted against time

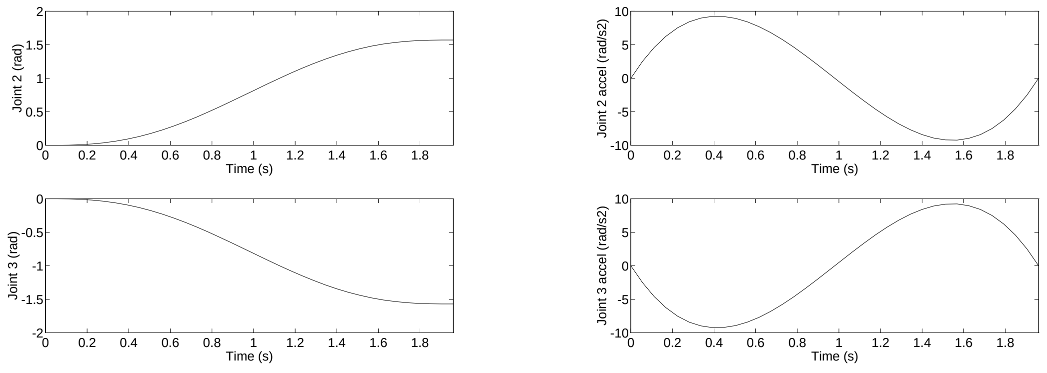

Figure 2: Joint space trajectory generated by `jtraj()`. Left: joint 2 and 3 angles; right: joint 2 and 3 acceleration.

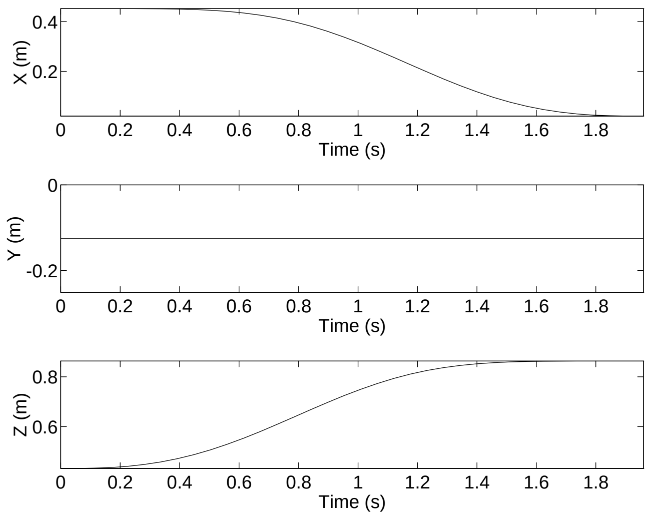

Figure 3: Cartesian coordinates of wrist for the trajectory of Figure 2.

```
>> subplot(3,1,1); plot(t, Ttg(:,13))
>> subplot(3,1,2); plot(t, Ttg(:,14))
>> subplot(3,1,3); plot(t, Ttg(:,15))
```

to give Figure 3, or we could plot X against Z to get some idea of the Cartesian path followed by the manipulator

```
>> subplot(1,1,1); plot(Ttg(:,13), Ttg(:,15))
```

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. Inverse kinematics can then be applied to determine the corresponding joint angle motion using

```
>> qc = ikine(p560, T);
```

When solving for a trajectory, the starting joint coordinates for each inverse kinematic solution is 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 in a few milliseconds.

5 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[2] and Hollerbach[6]. The equations of motion for an n-axis manipulator are given by

$$\underline{Q} = \mathbf{M}(\underline{q})\ddot{\underline{q}} + \mathbf{C}(\underline{q},\dot{\underline{q}})\dot{\underline{q}} + \mathbf{F}(\dot{\underline{q}}) + \mathbf{G}(\underline{q}) \tag{6}$$

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 $\underline{q}$.

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 Table 1. The `dyn` matrix is in fact a `dh` matrix augmented 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 $n \times 20$ matrix. This structure does not 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

Column	Symbol	Description
6	m	mass of the link
7	r_x	link COG with respect to the link coordinate frame
8	r_y	
9	r_z	
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

Table 1: Augmented column assigments for the Toolbox `dyn` matrix.

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(q, qd, qdd);
```

As with all Toolbox functions the result has one row per time step, and each 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 torques, can be computed separately by the `coriolis()` function.

Forward dynamics is the computation of joint accelerations given position and velocity state, and applied actuator torques and is particularly useful in simulation of a robot control system. The Toolbox uses Method 1 of Walker and Orin[7] which uses repeated calls to the inverse dynamics function `rne()`. Consider a Puma 560 at rest in the zero angle pose, with zero applied joint torques. The joint acceleration would be given by

```
>> a=accel(p560, qz, zeros(1,6), zeros(1,6))
>> a'
a = -0.2463 -8.7020 2.5442 0.0021 0.0672 0.0001
```

To be useful for simulation this 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

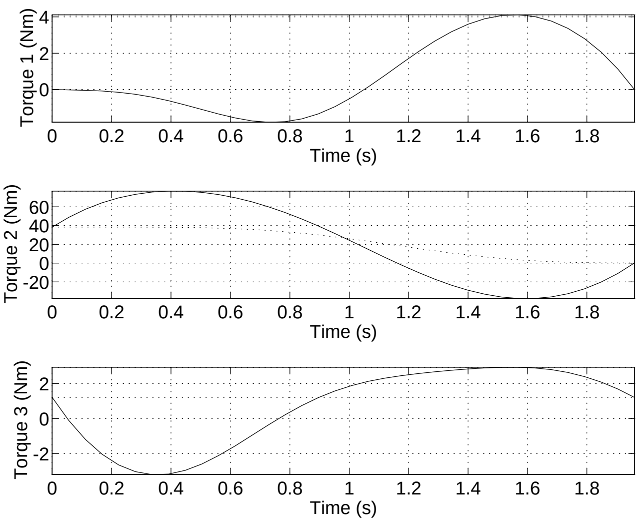

Figure 4: Joint torques for the joint space trajectory example of Figure 2.

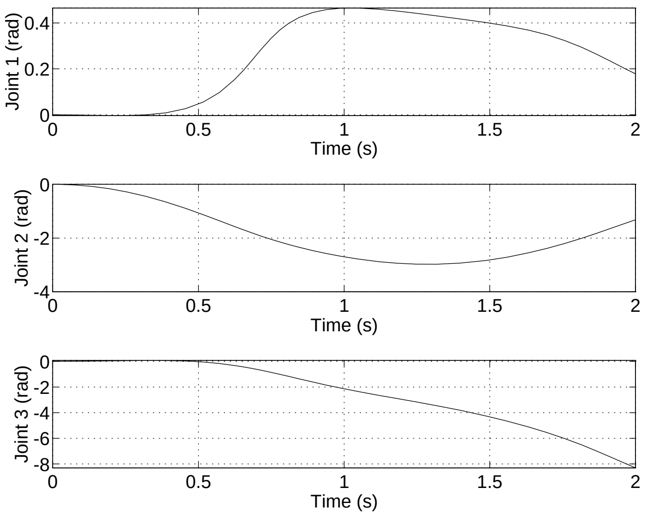

Figure 5: Simulated joint angle trajectory of Puma robot with zero applied joint torque collapsing under gravity.

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

The resulting motion is plotted versus time in Figure 5. which clearly shown 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 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).

6 Conclusion

This short paper 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 analyzing 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 `ftp.mathworks.com`, the MathWorks FTP server, in the directory `pub/contrib/misc/robot`. It is best to download all files in that directory since the Toolbox functions are quite interdependent. A comprehensive manual, in PostScript format, provides details of each Toolbox function. A menu-driven demonstration can be invoked by the function `rtdemo`.

References

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

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

[3] J. Funda, R. Taylor, and R. Paul, "On homogeneous transforms, quaternions, and computational efficiency," *IEEE Trans. Robot. Autom.*, vol. 6, pp. 382–388, June 1990.

[4] 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.

[5] D. Whitney and D. M. Gorinevskii, "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.

[6] 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.

[7] 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.