Systems Analysis Modelling Simulation
Vol. 43, No. 6, June 2003, pp. 759–769

DESIGN-SIMULATION-OPTIMIZATION PACKAGE FOR A GENERIC 6-DOF MANIPULATOR WITH A SPHERICAL WRIST

MHER GRIGORIAN and TAREK SOBH*

Department of Computer Science and Engineering, University of Bridgeport, Bridgeport, CT 06601, USA

(Received 19 April 2001)

Robot manipulators are built to meet certain predetermined performance requirements. The question of whether the robot will have the desired functionality (e.g. dexterity, accuracy, reliability, speed, etc.) needs to be answered before the robot is actually built.

We have developed a software package that can greatly ease the design of a generic 6-DOF manipulator with a spherical wrist. Our package will accept as input the configuration of a generic robot in D-H parameter form and the robot dynamics parameters and produce a variety of closed form solutions that are essential to the robot designer. The package can also be used as a simulation tool that can tell the designer whether the manipulator meets the desired functionality. It will also optimize several control and structure parameters for the generic manipulator based on simulated task descriptions.

Keywords: Robotics; Prototyping; Design; Simulation; Optimization

1. INTRODUCTION

In this section we discuss various well-known modules that provide a concise means of describing a robot model. We also show how our package aids the design of a manipulator by producing symbolic solutions for each of the modules.

1.1. Position Kinematics

The forward position kinematics module is necessary to determine the position and the orientation of the robot end-effector in terms of the robot joint variables. The joint variables are the angles between two links if the joint is revolute, and the joint extension for prismatic joints. The transformation matrix T_0^6, which gives the position and orientation of the end-effector in base coordinates, can be obtained by successively multiplying the homogenous transformation matrices A_i^j between two consecutive links. In case of a 6-DOF robot, $T_0^6 = A_0^1 * A_1^2 * A_2^3 * A_3^4 * A_4^5 * A_5^6$

*Corresponding author. E-mail: sobh@bridgeport.edu

ISSN 0232-9298 print: ISSN 1029-4902 online © 2003 Taylor & Francis Ltd
DOI: 10.1080/0232929031000150337

[5,9,10]. Given the D-H parameter table, our software package will form the $A_0^1 \cdots A_5^6$ matrices, perform the necessary matrix multiplications and obtain symbolically the $T_0^1 \cdots T_0^6$ matrices. The package generates C/C++ code to compute the position and orientation of the end-effector given the suggested joint variables for a generic 6-DOF robot.

1.2. Inverse Position Kinematics

The inverse kinematics problem, as opposed to forward kinematics, is needed to compute the joint variables of a robot given the position and the orientation of the end-effector. The inverse kinematics problem is extremely time-consuming and requires heavy calculations. For a 6-DOF robot, the transformation matrix T_0^6 defines twelve highly nonlinear trigonometric equations. In a 6-DOF robot with a spherical wrist, kinematic decoupling can be used to reduce the complexity of the inverse kinematics problem. However, even after applying kinematic decoupling the inverse kinematics equations remain remarkably complex for a generic robot. For a generic robot, the latest Mathematica and Maple math engines are usually unable to obtain a closed form solution [6,8]. The Robotics Toolbox for Matlab, on the other hand, can only produce numerical solutions for the inverse kinematics equations [1].

1.3. Velocity Kinematics and Jacobian

The velocity kinematics module is needed to compute the velocity relationships between the cartesian position and orientation of the robot end-effector and the joint variables. In a 6-DOF robot, the Jacobian is a 6×6 matrix. It is extensively used in the analysis and control of robot motion, planning and creation of smooth trajectories, detection of singularities, etc. The Jacobian can be represented as $J = [J_1\ J_2\ J_3\ J_4\ J_5\ J_6]$, where if joint i is revolute,

$$J_i = \begin{bmatrix} Z_{i-1} \times (O_i - O_{i-1}) \\ Z_{i-1} \end{bmatrix}$$

and if joint i is prismatic,

$$J_i = \begin{bmatrix} Z_{i-1} \\ 0 \end{bmatrix}$$

Z_i is defined by the first three elements in third column of T_0^i, and O_i is defined by the first three elements in fourth column of T_0^i [9,10]. Given the D-H parameter table, our software package performs the necessary computations to symbolically derive the Jacobian matrix. Having derived the Jacobian matrix, it is easy to obtain the velocity of the robot end-effector in terms of joint velocities from:

$$X' = J * Q'$$

where X' is the cartesian velocity vector, J is the Jacobian matrix, and Q' is the joint velocity vector [5,9,10]. Our package generates C/C++ code to compute the end-effector velocity given join velocities.

1.4. Inverse Velocity Kinematics

The inverse velocity module is needed to express the robot joint velocity vector in terms of the end-effector cartesian velocity vector. From the previous section we can derive the inverse velocity equations as:

$$Q' = J^{-1} * X'$$

where Q' is the joint velocity vector, J^{-1} is the inverse of the Jacobian matrix, and X' is the cartesian velocity vector [9,10]. We implemented a symbolic matrix inversion routine. Given the D-H parameter table, our software package symbolically inverts the Jacobian matrix to express the joint velocity vector in terms of the end-effector velocity vector. The matrix inversion routine was designed and tested to handle long mathematical expressions. The package generates C/C++ code to compute the joint velocity vector.

1.5. Acceleration Kinematics

The acceleration kinematics module is needed to express the acceleration of the robot end-effector given the joint accelerations. By differentiating the velocity kinematics equations, we obtain:

$$X'' = J * Q'' + J' * Q'$$

where X'' is the end-effector acceleration vector, J is the Jacobian matrix, Q'' is the joint acceleration vector, J' is the time derivative of the Jacobian, and Q' is the joint velocity vector [9,10]. In order to obtain J', we implemented a symbolic differentiation library. Given the D-H parameter table, our software package symbolically differentiates the Jacobian matrix, and performs all necessary computations to obtain the end-effector acceleration in terms of the joint accelerations. The differentiation routine was designed and tested to handle long mathematical expressions. The package also generates C/C++ code to compute the acceleration kinematics for a generic 6-DOF robot.

1.6. Inverse Acceleration Kinematics

The inverse acceleration kinematics module is needed to express the joint accelerations in terms of the end-effector acceleration vector. The equations needed in this module can be derived from the acceleration kinematics equations:

$$Q'' = J^{-1} * (X'' - J' * Q')$$

where Q'' is the joint acceleration vector, $.J^{-1}$ is the inverse of the Jacobian matrix, X'' is the end-effector acceleration vector, J' is the time derivative of the Jacobian, and Q' is the joint velocity vector [9,10]. Given the D-H parameter table, our package performs all the necessary computations to find a closed form solution for the joint acceleration vector. C/C++ code to compute the joint acceleration vector is generated. The modules mentioned above were tested on several robot models, including D-H parameters for a PUMA 560 robot.

2. TRAJECTORY GENERATION

2.1. Cubic Polynomial

In most scenarios, robots are commanded to move from one position to another in a time interval t. Let the joint variables vector be Q_0 at time t_0, and the joint velocity variables vector be Q'_0. At time t_f we would like the robot to have joint variables vector Q_1, and joint velocity variables vector Q'_1. In order to obtain smooth motion from Q_0 to Q_1, trajectory generator needs to be implemented. We implemented a polynomial trajectory generator with four independent coefficients of the form:

$$Q_d = a_0 + a_1 * t + a_2 * t^2 + a_3 * t^3$$
$$Q'_d = a_1 + 2 * a_2 * t + 3 * a_3 * t^2$$

where Q_d is the desired joint position vector, and Q'_d is the desired joint velocity vector. The four coefficients a_0, a_1, a_2, a_3 can be found by solving the equations that satisfy the initial constraints (Q_0, Q'_0, Q_1, Q'_1). As a result, we have:

$$A_0 = Q_0$$
$$A_1 = Q'_1$$
$$A_2 = \left(3 * (Q_1 - Q_0) - \left(2 * Q'_0 + Q'_1\right) * t_f\right)/t_f^2$$
$$A_3 = \left(2 * (Q_0 - Q_1) + \left(Q'_0 + Q'_1\right) * t_f\right)/t_f^3$$

The trajectory generator defines where the robot joint variables should be at any time t while moving from Q_0 to Q_1 [10]. Our software package generates C/C++ code that implements the trajectory generator described above.

2.2. Constant Velocity with Cubic Polynomial Blends

In many cases, constant velocity is desired along a portion of the path of the robot movement. In such cases, constant velocity with cubic polynomial blends trajectory generator can be used. To achieve the desired performance, we specify the position and velocity in three parts. The first part from time t_0 to time t_b is cubic polynomial (described in Section 2.1). At time t_b the velocity of the joints reaches the desired velocity and the trajectory switches to a linear function, which corresponds to a constant velocity. At time $t_f - t_b$ the trajectory switches back to cubic polynomial. C/C++ code implementing this trajectory generator is generated by our package.

3. SIMULATION AND CONTROL

3.1. Dynamics

The dynamics model of a robot identifies what torques need to be applied on joints in order to cause the manipulator move in a certain manner. The rigid body dynamics of a robot have form:

$$\tau = M(Q) * Q'' + V(Q, Q') + G(Q) + F(Q, Q')$$

where τ is the torque vector applied to the links, $M(Q)$ is the inertia matrix of the manipulator, $V(Q, Q')$ is the vector of centrifugal and Coriolis terms, $G(Q)$ is the vector of gravity terms, and $F(Q, Q')$ is the model of friction [4,7]. The dynamics equations of a robot depend on joint locations, the mass, mass distribution, and length of each link, etc. Given the desired joint angles, speed, and acceleration, the dynamics equations tell what torques should be applied to each joint. Given the M, V, G and F matrices our package will generate C/C++ code for the dynamics model. The dynamics module is the most time consuming component among the manipulator's modules due to the heavy calculations involved in dynamics equations. For this reason most of the times in current industrial practice a feedback control system, which is an approximation of the nonlinear nature of the dynamics equations, is being used instead of the dynamics model [2,3]. In our software package we use a local PD controller, as described in the next sections.

3.2. Inverse Dynamics

The inverse dynamics module can be used to calculate the new position, velocity and acceleration of a manipulator given a set of torques applied to each link. The inverse dynamics equations can be derived from the dynamics equations [4,7]:

$$Q'' = M^{-1}(Q) * (\tau - V(Q, Q') - G(Q) - F(Q, Q'))$$

The inverse dynamics module can be very useful in simulating the motion of a robot. This is particularly beneficial since the designers can have a reasonable estimate on how the robot will respond to a set of torques before the robot is actually built. If the robot's performance is not satisfactory, changes can be made to the design of the model. Our software package implements an inverse dynamics model. Given the M, V, and G matrices the package will generate C/C++ code needed to calculate the resulting joint angles, speed and acceleration.

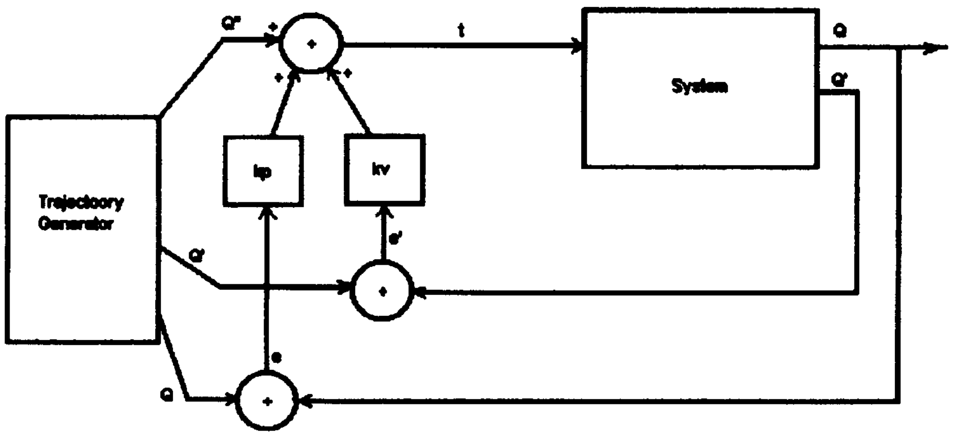


FIGURE 1 Block diagram of the PD controller.

3.3. PD Controller

In our software package we have implemented a local proportional plus derivative (PD) controller (Fig. 1). Many of the feedback algorithms used in industrial practice are based on variations of the PD controller.

The control system we used can be represented in the form:

$$\tau = f(Q,Q',Q'') * Q'' + f(Q,Q',Q'') * K_p * e_p + f(Q,Q',Q'') * K_v * e_v$$

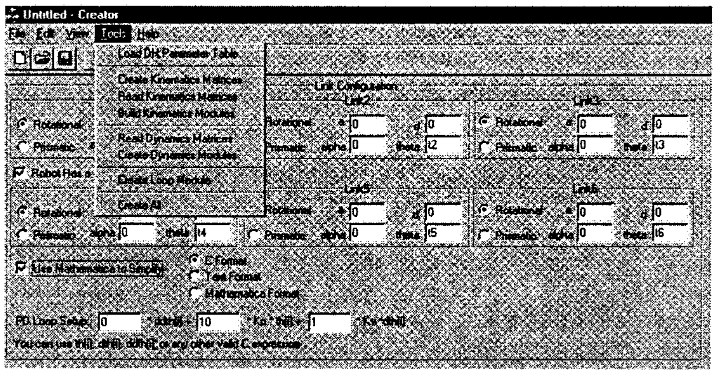
where f is a function of robot joint position, velocity, and acceleration vectors, Q'' is the desired link acceleration vector, K_p is the proportional gain, K_v is the derivative gain, e_p is the error in joint variables vector, and e_v is the error in joint velocities vector. Our package accepts any mathematical function f as a coefficient to the loop variables. This allows the user to highly customize the loop controller. When the PD feedback algorithm is being used to control a robot, the readings from sensors constitute the feedback to the control system. When it is used for simulation, the inverse dynamics model can be used to approximate the robot behavior. Since the PD controller requires very little computation, high update rates, required to obtain stability, are possible in real time [2,3]. We also have the option of including a PID and/or other feedback control functions. Nonlinear compensators can be applied to reject time varying disturbances.

4. THE PACKAGE AND IMPLEMENTATION

Our package consists of two modules: the creation module and the execution module.

4.1. Creation Module

The creation module is responsible for creating the kinematics, dynamics, and the PD controller modules. It will take as input the D-H parameter table and symbolically derive the equations in text, C/C++ or Mathematica formats. Figure 2 shows the



Link2: Rotational; a 0; d 0; Prismatic; alpha 0; theta t2

Link3: Rotational; a 0; d 0; Prismatic; alpha 0; theta t3

Robot Has a

Rotational; Prismatic; alpha 0; theta t4

Link5: Rotational; a 0; d 0; Prismatic; alpha 0; theta t5

Link6: Rotational; a 0; d 0; Prismatic; alpha 0; theta t6

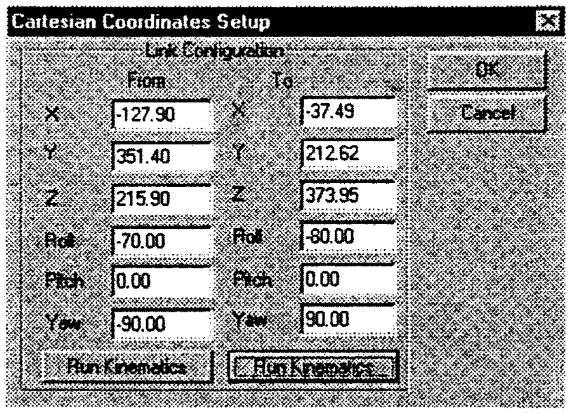
Use Mathematica to Simplify [✓]

[✓ C Format] Tex Format Mathematica Format

PD Loop Setup: 0 * ddth[i] + 10 * Kp * th[i] + 1 * Kv * dth[i]

You can use th[i], dth[i], ddth[i], or any other valid C expression

FIGURE 2 Main screen for the creation module.



	From		To
X	-127.90	X	-37.49
Y	351.40	Y	212.62
Z	215.90	Z	373.95
Roll	-70.00	Roll	-80.00
Pitch	0.00	Pitch	0.00
Yaw	-90.00	Yaw	90.00

FIGURE 3 Cartesian coordinates setup.

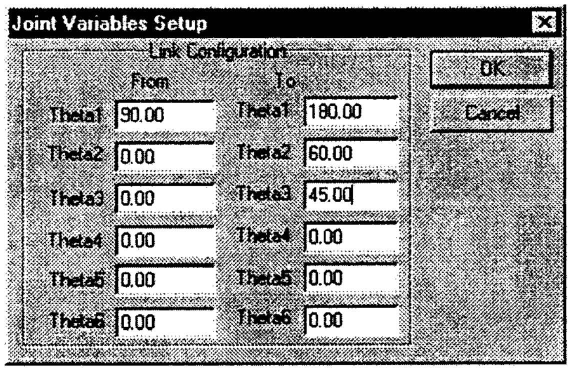
main screen for the creation module. The equations derived by this module will be used as input by the execution module to run simulation and optimization tasks.

4.2. Execution Module

The execution module can be used as a simulation and/or optimization tool. The user has to enter the initial and final positions of the manipulator either in cartesian coordinates (Fig. 3) or in terms of joint variables (Fig. 4).

Other basic settings need to be provided as well: the K_p and K_v values, the time interval in which the robot should move from the initial to the final position, the update rate in the feedback controller, the trajectory generator to be used, etc (Fig. 5).

After the settings have been successfully initialized, our software package will run the control loop on the points specified by the user and produce graphs showing the differences between the desired and actual positions of the manipulator. A simplified 3D model showing the movement of the manipulator will be displayed as well. Figure 6 show screen shots obtained during program execution.

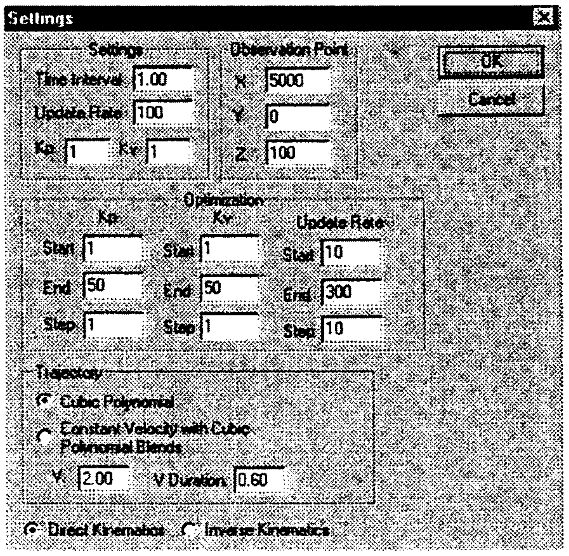


Joint Variables Setup

Link Configuration

	From		To
Theta1	90.00	Theta1	180.00
Theta2	0.00	Theta2	60.00
Theta3	0.00	Theta3	45.00
Theta4	0.00	Theta4	0.00
Theta5	0.00	Theta5	0.00
Theta6	0.00	Theta6	0.00

FIGURE 4 Joint variables setup.



Settings

Settings: Time Interval 1.00; Update Rate 100; Kp 1; Kv 1

Observation Point: X 5000; Y 0; Z 100

Optimization

	Kp		Kv		Update Rate
Start	1	Start	1	Start	10
End	50	End	50	End	300
Step	1	Step	1	Step	10

Trajectory

[✓ Cubic Polynomial] Constant Velocity with Cubic Polynomial Blends

V: 2.00 V Duration: 0.60

[✓ Direct Kinematics] Inverse Kinematics

FIGURE 5 Settings screen.

When run in the optimization mode, our package can determine the best values for K_p, K_v, and/or the update rate in addition to some potential robot dynamics parameter optimizations. Given the range of loop parameters (Fig. 5), the package runs a control loop for all combinations of the parameters under consideration. It will determine the best loop parameters by accumulating the absolute values of the difference between the desired and observed joint angles. Figure 7 shows a set of optimized parameters obtained by our package.

The user then can run the simulation with optimized parameters (Fig. 8) and compare the results with the performance obtained by using custom, nonoptimized ones (Fig. 9).

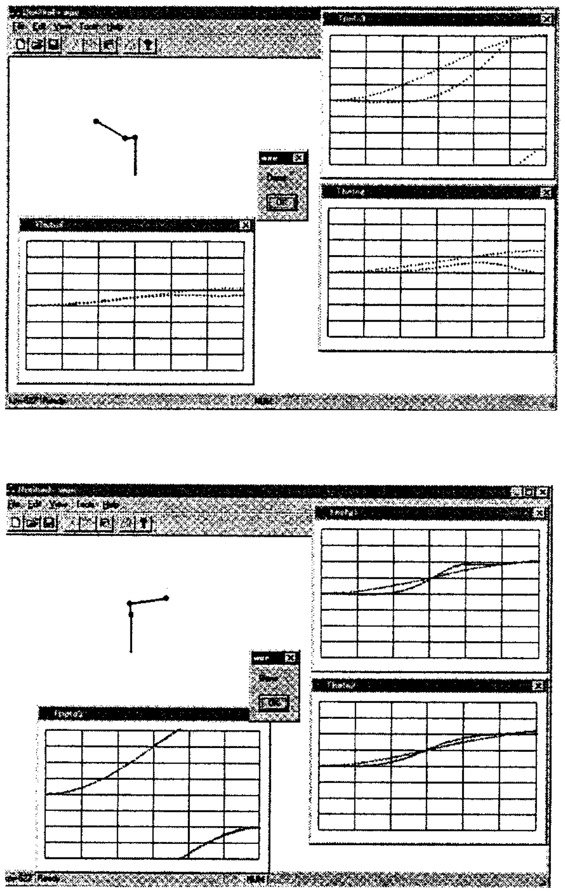


FIGURE 6 Simulation screen shot (the graphs represent desired *versus* actual robot coordinates).

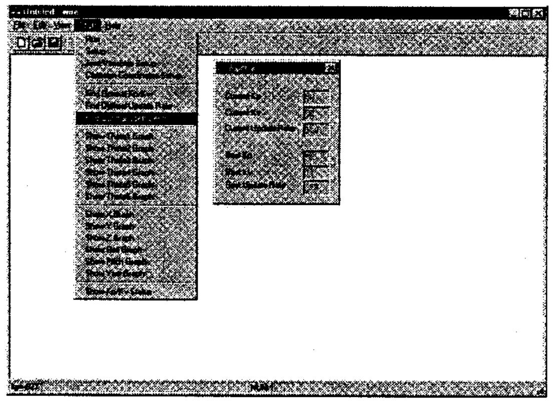


FIGURE 7 K_p, K_v, and update rate optimization.

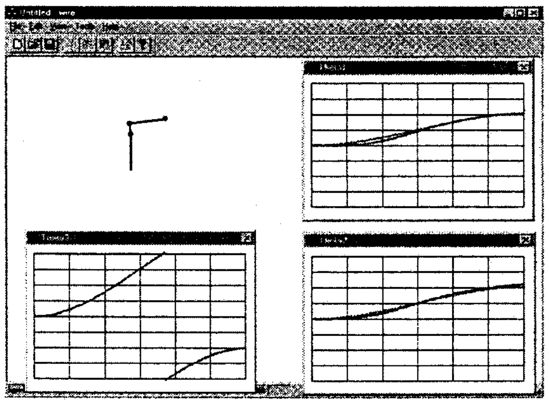


FIGURE 8 Desired *versus* observed robot coordinates with optimized parameters.

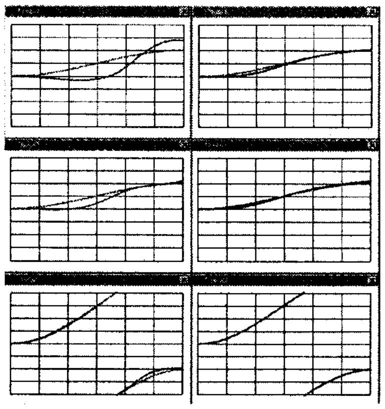


FIGURE 9 User defined *versus* optimized K_p, K_v, and update rate.

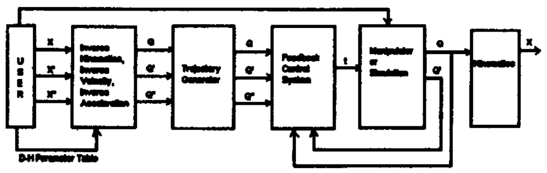

USER
X
X'
X''
Inverse Kinematics, Inverse Velocity, Inverse Acceleration
Q
Q'
Q''
Trajectory Generator
Feedback Control System
t
Manipulator or Simulation
X
D-H Parameter Table

FIGURE 10 Block diagram for the simulation package.

5. CONCLUSION AND FUTURE ENHANCEMENTS

Figure 10 summarizes our software package:

Our package can reduce the difficulties that arise during the design of a new robot by creating a number of modules that need to be created before a robot is built.

The feedback control simulation will give a good estimate of whether the robot will have the desired functionality. The package can optimize the PD controller and determine the optimal values for some dynamic parameters that minimize robot errors. Potential future enhancements to the package include: (1) generation of the inverse kinematics module and including a numerical solution package for a set of robots; (2) implementation of more advanced trajectory generation algorithms; (3) implementation of a joint PID (Proportional Integral Derivative) controller, which can achieve higher accuracy and reliability than the current PD controller, etc.

References

[1] P.I. Corke (1996). A Robotics toolbox for MATLAB. *IEEE Robotics and Automation*, **3**(1).

[2] Mohamed Dekhil, Thomas C. Henderson, Tarek M. Sobh and A. Sabbavarapu (1996). *Prototyping a Three-link Robot Manipulator*. Presented in the Second World Automation Congress, Sixth International Symposium on Robotics and Manufacturing (ISRAM 96), Montpellier, France.

[3] Mohamed Dekhil, Tarek M. Sobh, Thomas C. Henderson, Anil Sabbavarapu and Robert Mecklenburg (1994). *Robot Manipulator Prototyping* (*Complete Design Review*). University of Utah, Salt Lake City, Utah, U.S.A.

[4] Charlos Canudas De Wit, Bruno Siciliano and Georges Bastin (1996). *Theory of Robot Control*. Springer-Verlag, London.

[5] C.Y. Ho and J. Sriwattanathamma (1990). *Robot Kinematics Symbolic Automation and Numerical Synthesis*. Ablex Publishing Corporation, Westport, Connecticut, U.S.A.

[6] Maple V Release 5, Version 5.00, Waterloo Maple Inc., Waterloo, Ontario, Canada.

[7] Andrew W. Marris and Charles E. Stoneking (1967). *Advanced Dynamics*, McGraw-Hill, New York, New York, U.S.A.

[8] Mathematica 3.0, Wolfram Research Inc., Champaign, Illinois, U.S.A.

[9] Phillip John McKerrow (1991). *Introduction to Robotics*. Addison Wesley, Boston, Massachusetts, U.S.A.

[10] W. Mark Spong (1989). *Robot Dynamics and Control*. John Wiley, Hoboken, New Jersey, U.S.A.