KCT : a MATLAB toolbox for motion control of KUKA robot manipulators

Francesco Chinello, Stefano Scheggi, Fabio Morbidi, Domenico Prattichizzo

***Abstract*— The *Kuka Control Toolbox* (KCT) is a collection of MATLAB functions for motion control of KUKA robot manipulators, developed to offer an intuitive and high-level programming interface to the user. The toolbox, which is compatible with all 6 DOF small and low payload KUKA robots that use the Eth.RSIXML, runs on a remote computer connected with the KUKA controller via TCP/IP. KCT includes more than 30 functions, spanning operations such as forward and inverse kinematics computation, point-to-point joint and Cartesian control, trajectory generation, graphical display, 3-D animation and diagnostics. The flexibility, ease of use and reliability of the toolbox is demonstrated through two applicative examples.**

I. INTRODUCTION

MATLAB [1] is a powerful and widely used commercial software environment for numerical computations, statistic analysis and graphical presentations available for a large number of platforms. Specific toolboxes (i.e., collection of dedicated MATLAB functions) have been developed in the past few years for support in research and education, in almost every branch of engineering, such as, e.g., telecommunications, electronics, aerospace, mechanics and control. As far as robotics is concerned, several toolboxes have been presented in the last decade for the modeling of robot systems [2]–[7]. These *simulation tools* have been inspired by various applicative scenarios, such as robotic vision [5], [6] and space robotics [3], and have addressed different targets ranging from industrial [4] to academic-educational [2], [5]–[7]. A more challenging problem is to design MATLAB toolkits, offering versatile and high-level programming environments, for motion control of *real robots*. Some work has been done in this field for one of the first industrial robots, the Puma 560 manipulator [8], [9]: however this robot is known to have some intrinsic software limitations, especially in real-time applications, which have been overcome by more recent manipulators.

In this paper we will focus on the manipulators produced by KUKA [10], one of the world's leading manufacturers of industrial robots. KUKA manipulators are designed to cover a large variety of applications in industrial settings, such as, e.g., assembly, material handling, dispensing, palletizing and welding tasks. A specific C-like programming language, called KRL (KUKA Robot Language), has been developed by KUKA for robot motion control. This language is simple and allows comfortable programming. However, it is not suited for critical real-time remote control applications, it does not support graphical interfaces and advanced calculus (such as, matrix operations, optimization and filtering tasks), and it does not allow an easy integration of external modules and hardware (such as, e.g., cameras or embedded devices using common protocols: USB, Firewire, PCI, etc.). A possible way to overcome these drawbacks is to build a MATLAB abstraction layer upon the KRL. A first step torwards this direction has been recently taken by a MATLAB toolbox called *Kuka-KRL-tbx* [11]. The authors use a *serial interface* to connect the KUKA Robot Controller (KRC) with a remote computer including MATLAB. A KRL interpreter, running on the KRC, realizes a bi-directional communication between the robot and the remote computer and it is responsible for the identification and execution of all intructions that are transmitted via the serial interface. *Kuka-KRL-tbx* offers a homogeneous environment from the early design to the operation phase and an easy integration of external hardware components. In addition, it preserves the security standards guaranteed by the KRL (workspace supervision, check of the final position switches of every robot axis, etc.).

However, *Kuka-KRL-tbx* suffers from some limitations:

- The MATLAB commands of the toolbox are one-to-one with the KRL functions: this lack of abstraction may hinder the user from designing advanced control applications.
- The *serial interface* may represent a limit in real-time control applications.
- The toolbox does not include specific routines for *graphical display*.

This paper presents a new MATLAB toolbox, called *KUKA Control Toolbox* (KCT), for motion control of KUKA robot manipulators. The toolbox, designed both for academic/educational and industrial purposes, includes a broad set of functions divided into 6 categories, spanning operations such as, forward and inverse kinematics computation, point-to-point joint and Cartesian control, trajectory generation, graphical display, 3-D animation and diagnostics.

KCT shares with *Kuka-KRL-tbx* the same advantages and improves it in several directions:

- The functions of KCT are not a MATLAB counterpart of the corresponding KRL commands. This makes the toolbox extremely flexible and versatile.
- KCT runs on a remote computer connected with the KRC via TCP/IP. A *multi-thread server* runs on the KRC and communicates via Eth.RSIXML (Ethernet

The authors are with the Department of Information Engineering, University of Siena, 53100 Siena, Italy. List of e-mails: {chinello,scheggi,morbidi,prattichizzo}@dii.unisi.it

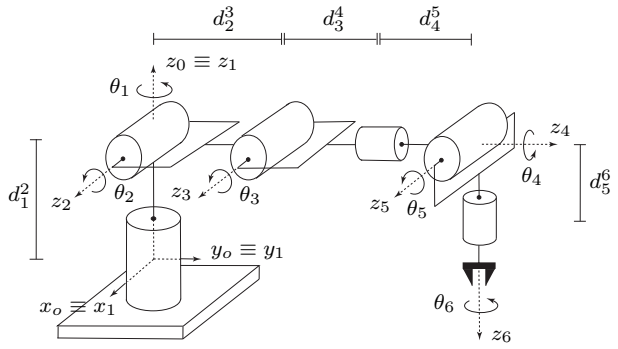


Fig. 1. Reference robot: 6 DOF elbow manipulator with spherical wrist.

Robot Sensor Interface XML) with a client managing the information exchange with the manipulator. This communication scheme guarantees high transmission rates, thus enabling real-time control applications.

- KCT has several dedicated functions for *graphics* and *animation*, and includes a graphical user interface.
- The toolbox and the relative documentation can be freely downloaded from the web page: `http://sirslab.dii.unisi.it/vision/kct`

KCT is fully compatible with all small and low payload 6 DOF KUKA robot manipulators which run the 5.4, 5.5 or 7.0 KSS (Kuka System Software): the controllers KR C2, KR C2 ed05 and KR C3 (equipped with a real-time 10/100 card) are currently supported by the toolbox. KCT can be easily integrated with other MATLAB toolboxes and it has been successfully tested on multiple platforms, including Windows, Mac and Linux.

The rest of the paper is organized as follows: Sect. II provides a comprehensive overview of the functions of KCT. Two examples are reported in Sect. III to show the flexibility of the toolbox in real scenarios. In Sect. IV, conclusions are drawn and future research directions are highlighted.

II. Overview of the Toolbox

In this section the functions of KCT will be briefly described. The 6 DOF robot manipulator shown in Fig. 1 will be considered as a reference along the paper: we will use the vector $\mathbf{q} = [\theta_1, \theta_2, \ldots, \theta_6]^T$ to denote the collection of the joint angles of the manipulator and $\mathbf{d}_{k-1}^k \in \mathbb{R}^3$, $k \in \{1, 2, \ldots, 6\}$, to indicate the displacement between the center of the $(k-1)$-th and k-th joint of the robot (note that $\mathbf{d}_0^1 \equiv \mathbf{0}$). The homogeneous matrix $\mathbf{H}_0^6 \in \mathrm{SE}(3)$ relates the coordinates of a 3-D point written in the base reference frame $\langle x_0, y_0, z_0 \rangle$, with the coordinates of the same point written in the end-effector frame $\langle x_6, y_6, z_6 \rangle$.

In the interest of clarity, the commands of KCT have been subdivided into 6 categories, according to the task they perform (see Table II, next page). The KUKA robot models currently supported by KCT are listed in Table I: up to now, the toolbox has been successfully tested on the KR3, KR5sixxr850 and KR16 robots. Fig. 2 illustrates the communication scheme between KCT and the robot manipulator. It consists of three parts:

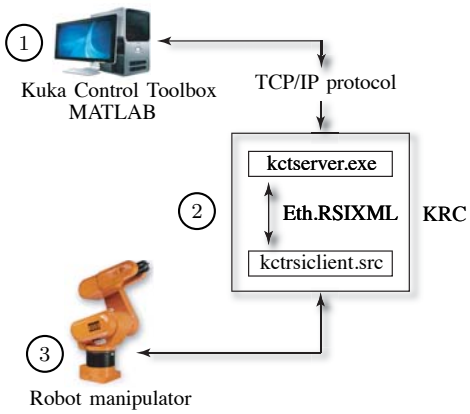


Fig. 2. Communication scheme between KCT and the manipulator.

1) A remote computer running KCT under MATLAB,
2) The KUKA Robot Controller (KRC),
3) The robot manipulator.

To establish a connection between the remote computer and the robot controller, KCT provides `kctserver`, a C++ multi-thread server running on the KRC. `kctserver` communicates via Eth.RSIXML (a KUKA software package for TCP/IP-robot interface) with `kctrsiclient.src`, a KRL client running on the KRC and managing the information exchange with the robot manipulator. The MATLAB functions of KCT communicate with `kctserver` using specific MEX-files (default option) or via the *Instrument Control Toolbox*. The server sends the robot's current state to the computer and the velocity commands to the manipulator via `kctrsiclient.src`, in a soft real-time loop of 15 ± 1 ms (the communication has been temporized to 15 ms in order to have a suitable margin over the 12 ms physical time limit: the ± 1 ms uncertainty is currently due to the lack of a hard real-time support platform). `kctrsiclient.src` is also used to define a `HOME` position (starting position) for the manipulator. Hardware and software contraints on robot's motion are established by Eth.RSIXML. The hardware constraints depend on manipulator's physics and cannot be modified by the user. On the contrary, the software constraints can be configured at the beginning of each working session via the functions `kctrsiclient.src` or `kctsetbound` (see Sect. II-A for more details).

In the following, all the angles will be in degrees and distances in millimeters.

Small Robots (from 3 to 5 kg)				
KR3	KR5sixxr650	KR5sixxr850		
Low Payloads (from 6 to 16 kg)				
KR5arc	KR5arcHW	KR6-2	KR6-2KS	
KR16-2	KR16-2S	KR16-2KS	KR16L6-2	KR16L6-2KS

TABLE I
6 DOF KUKA robots currently supported by KCT.

Initialization	
`kctrobot`	Show the list of supported KUKA robots
`kctinit`	Load the parameters of the selected robot
`kctsetbound`	Set the workspace bounds
`kctgetbound`	Visualize the workspace bounds
Networking	
`kctclient`	Initialize the client
`kctcloseclient`	Terminate the client
Kinematics	
`kctreadstate`	Return the current configuration of the robot
`kctfkine`	Compute the forward kinematics
`kctikine`	Compute the inverse kinematics
`kctfkinerpy`	Compute the forward kinematics (give the pose)
`kctikinerpy`	Compute the inverse kinematics (from the pose)
Motion control	
`kctsetjoint`	Set the joint angles to a desired value
`kctsetxyz`	Move the end-effector in a desired position
`kctmovejoint`	Set the joint velocities to a desired value
`kctmovexyz`	Move the end-effector with a desired linear and angular velocity
`kctdrivegui`	GUI for robot motion control
`kctpathxyz`	Generate a trajectory (operational space)
`kctpathjoint`	Generate a trajectory (joint space)
`kcthome`	Drive the robot back to the initial position
`kctstop`	Stop the robot in the current position
`kctdemo`	Demonstration of the toolbox
Graphics	
`kctdisptraj`	Plot the 3-D trajectory of the end-effector
`kctdispdyn`	Plot the time history of the joint angles
`kctanimtraj`	Create a 3-D animation of the robot
Homogeneous transforms	
`kctrotox`	Hom. transform for rotation about x-axis
`kctrotoy`	Hom. transform for rotation about y-axis
`kctrotoz`	Hom. transform for rotation about z-axis
`kcttran`	Hom. transform for translation
`kctchframe`	Change the reference frame

TABLE II

LIST OF KCT FUNCTIONS DIVIDED BY CATEGORY.

'*name*'	*KR3*	*KR5sixxr650*	*KR5sixxr850*	···
'link1' [mm]	350	335	335	
'link2' [mm]	100	75	75	
'link3' [mm]	265	270	365	···
'link4' [mm]	0	90	90	
'link5' [mm]	270	295	405	
'link6' [mm]	75	80	80	

TABLE III

DATA STORED IN THE FILE `KCTROBOTDATA.MAT`.

A. Initialization

The information relative to the KUKA robots supported by KCT is stored in the MATLAB file `kctrobotdata.mat` (see Table III) and can be accessed by typing,

```
>> kctrobot();
```

To initialize a particular robot model, it is sufficient to write `kctinit('KR3')`, where the argument is a string containing the name of the robot selected (e.g., `KR3`, `KR5sixxr650`, `KR5sixxr850`, etc.) as specified in `kctrobotdata.mat`. The function `kctsetbound(B)` can be used to set the software bounds of the robot. The matrix,

$$\mathbf{B} = \begin{bmatrix} X^m & X^M & Y^m & Y^M & Z^m & Z^M \\ \theta_4^m & \theta_4^M & \theta_5^m & \theta_5^M & \theta_6^m & \theta_6^M \end{bmatrix},$$

contains the bounds on the position and orientation (limited to the joint angles θ_4, θ_5 and θ_6) of the end-effector. Note that differently from `kctrsiclient.src`, `kctsetbound` enables a MATLAB warning message in the motion control functions (see Sect. II-D), when the workspace's bounds are violated. The bounds can be graphically visualized using the function `kctgetbound`.

B. Networking

After the initialization step, the TCP/IP communication between KCT and `kctserver` must be established. The main steps necessary to initialize the connection are the following:

1) Put the mode selector on T1 (automatic execution) in the KUKA control panel (teach pendant).
2) Select `kctrsiclient.src` on KSS.
3) Start `kctserver`.
4) In the MATLAB workspace start the KCT/IP communication with `kctserver` by typing `kctclient('193.155.1.0')`, where `193.155.1.0` is the IP address of the KRC real-time network card.
5) Start `kctrsiclient.src` by pressing the run button on the KUKA control panel and keep pushing until the line `ST_SKIPSENS` is reached.
6) Check whether the communication is established. If it is not, return to step 2.

To close the TCP/IP communication is sufficient to type `kctcloseclient()`.

C. Kinematics

The state of the manipulator is stored in a 2×6 matrix, called `robotstate`, containing the current position and roll-pitch-yaw orientation of the end-effector (first row), and the current joint angles (second row). This matrix can be accessed using the command: `robotstate = kctreadstate()`. To compute the matrix $\mathbf{H}_0^6$ of the forward kinematics and the inverse kinematics solution expressed as a joint angles' vector $\mathbf{q}$, KCT provides the following two functions:

```
>> q = [13, 32, -43, 12, 54, 15];
>> H06 = kctfkine(q);
>> q' = kctikine(H06);
```

The command `p = kctfkinerpy(q)` is analogous to `kctfkine`, but returns the position and roll-pitch-yaw orientation of the end-effector of the robot, as a vector $\mathbf{p} = [X, Y, Z, \phi, \gamma, \psi]^T$. Finally, the function `q = kctikinerpy(p)` computes the inverse kinematics solution from the vector $\mathbf{p}$.

Fig. 3. The interface loaded by the function `kctdrivegui`.

D. Motion control

KCT provides several functions for point-to-point control and trajectory planning. The simplest task one could require, is to move the robot from an initial to a final configuration defined by robot's joint angles or by end-effector's poses. Let $\mathbf{q}_f = [\theta_1, \theta_2, \ldots, \theta_6]^T$ be the final desired joint configuration of the robot. The function,

```
>> qf = [23, 35, 12, -21, 54, 60];
>> [robotinfo, warn] = kctsetjoint(qf,'poly');
```

moves the robot from the current to the final configuration using either a polynomial approach [12, Sect. 5.5.2] or a proportional control, as specified by the second argument (`poly` or `prop`, respectively). The matrix `robotinfo` contains the time history of the joint angles and `warn` is a Boolean variable that is set to 1 when an error occurs during robot's motion. Let now $\mathbf{p}_f = [X, Y, Z, \phi, \gamma, \psi]^T$ be the final desired pose of the end-effector. The function,

```
>>  pf = [412, -2, 350, 20, 12, 15];
>>  [robotinfo, warn] = kctsetxyz(pf);
```

moves to robot from the initial to the desired pose $\mathbf{p}_f$ using a proportional controller. Note that `kctsetjoint` and `kctsetxyz` are user-level routines relying on two lower level functions: `kctmovejoint` and `kctmovexyz`. When `kctsetjoint` is called, the KUKA controller computes the joint velocities necessary to accomplish the requested task using `kctmovejoint(qdot)`. Similarly, when `kctsetxyz` is called, the linear and angular velocities of the end-effector necessary to achieve the goal are computed via `kctmovexyz(pdot)`.

A graphical user interface, inspired by Robotics Toolbox's `drivebot` GUI [2], is loaded by `kctdrivegui()`, (see Fig. 3). The joint angles of the robot can be easily regulated here by acting on six sliders, and the corresponding motion of the robot is diplayed via a 3-D animation.

It is very frequent in the applications to deal with trajectories defined by a sequence of Cartesian frames or joint angles. Consider a sequence of n points $\mathbf{p}_i = [X_i, Y_i, Z_i, \phi_i, \gamma_i, \psi_i]^T$, $i \in \{1, 2, \ldots, n\}$, stacked into the $n \times 6$ matrix $\mathbf{P} = [\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n]^T$. The following commands,

```
>>  P = [100, 200, 150, 12, -23, 0;
>>       10, 0, 50, 24, -15, 11;
>>       -50, -30, 100, -10, 40, 32]; tp = 2;
>>  [robotinfo, warn] = kctpathxyz(P,tp,1);
```

move the end-effector of the robot from point $\mathbf{p}_1$ to point $\mathbf{p}_3$ using a linear interpolation method: a time step `tp` of 2 seconds between two consecutive points is considered. The third argument of `kctpathxyz` is a Boolean variable enabling or disabling the visualization of the 3-D trajectory of the end-effector and the time history of the joint angles at the end of the task. The function `kctpathjoint` is analogous to `kctpathxyz`, the only difference being that the trajectory is defined in the joint space instead of the operational space. The first argument of `kctpathjoint` is a $n \times 6$ matrix $\mathbf{Q}$, whose rows are vectors of joint angles:

```
>>  Q = [23, 35, 12, -21, 54, 60;
>>       42, -10, 20, 14, -5, 21;
>>      -15, 31, 10, 12, 20, 80]; tp = 2;
>>  [robotinfo, warn] = kctpathjoint(Q,tp,1);
```

To immediately stop the robot in the current position, one should first terminate the execution of the motion control functions using `ctrl-c`, and then type `kctstop()`. Finally, to drive the robot back to the initial position, KCT provides the function `kcthome()`.

E. Graphics

Three functions are available in KCT for graphical display. The function `kctdisptraj(robotinfo)` plots the 3-D trajectory of the end-effector, `kctdispdyn(robotinfo)` plots the time history of the robot joint angles and `kctanimtraj(robotinfo)` creates a 3-D animation of the robot executing the requested task. A suite of options (frame's and trajectory's color, frame's dimension, viewpoint, etc.) is available for the customization of the plot. (see the `help` of the single commands for more details).

F. Homogeneous transforms

KCT provides a set of transformation functions of frequent use in robotics. Let $\mathbf{d} \in \mathbb{R}^3$ be a translation vector and α an angle. The functions,

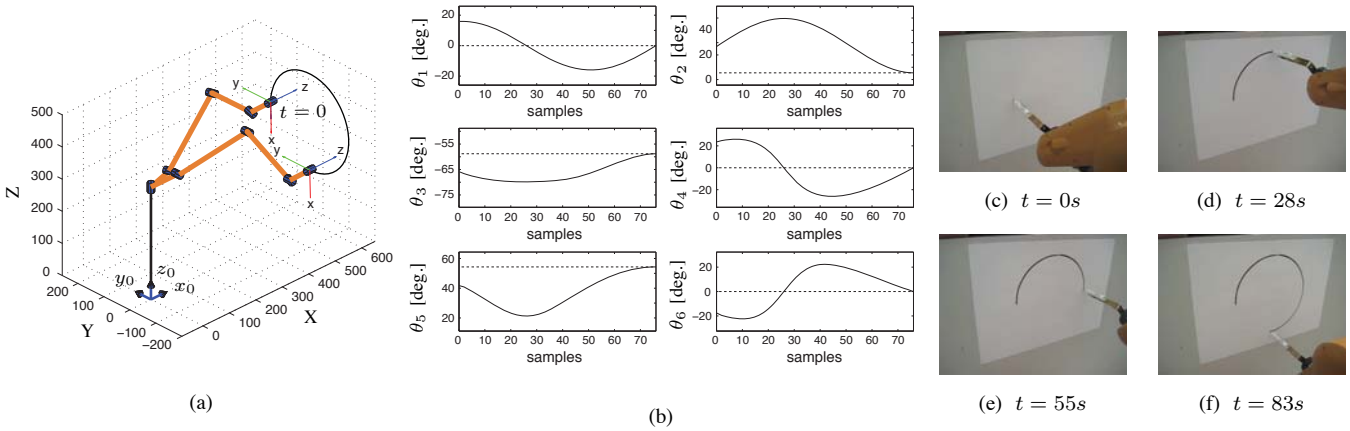
```
>>  Htr = kcttran(d); Hx = kctrotox(alpha);
>>  Hy = kctrotoy(alpha); Hz = kctrotoz(alpha);
```

provide the basic homogeneous transformations generating SE(3) for translation and rotation about the x, y, z-axes.

Let us now suppose that we wish to move the robot's end-effector with respect to an external reference frame $\langle x_w, y_w, z_w \rangle$ different from the base $\langle x_0, y_0, z_0 \rangle$. This could be useful, for instance, in an eye-in-hand framework where robot's motion should be referred to the camera frame. Let $\mathbf{H}_0^w$ be the homogeneous matrix defining the rigid motion between $\langle x_w, y_w, z_w \rangle$ and $\langle x_0, y_0, z_0 \rangle$. The function,

```
>>  H0w = kctrotoz(alpha)*kcttran(d);
>>  kctchframe(H0w);
```

fixes $\langle x_w, y_w, z_w \rangle$ as new reference frame. All the operations specified by commands executed after `kctchframe` are referred to $\langle x_w, y_w, z_w \rangle$.



(a) (b) (c) $t = 0s$ (d) $t = 28s$ (e) $t = 55s$ (f) $t = 83s$

Fig. 4. *Example 1*: (a) Trajectory of the end-effector; (b) Time history of the joint angles (solid); (c)-(f) Snapshots from the experiment.

III. Applicative Examples

This section presents two examples demonstrating the flexibility and ease of use of KCT in real scenarios. The first example shows an elementary application of the trajectory control functions (e.g., for painting, welding or assembling tasks). The second example reflects authors' personal interest in robot vision. In fact, the effectiveness of a camera rotation estimation method based on the geometry of planar catadioptric stereo (PCS) sensors [13] is tested using KCT.

The experimental results we will illustrate in the next subsections, have been obtained using the KUKA KR3 manipulator with KR C3 controller[1].

A. Drawing a circle

Suppose we wish to draw the circle (in millimeters),

$$x(k) = 600, \quad y(k) = 150\cos(k), \quad z(k) = 150\sin(k) + 310,$$

$k \in [0, 3\pi/2]$, on a paper board, with a pen mounted on the flange of the KUKA KR3 manipulator. To achieve this goal, we have first to initialize the robot using the command `kctinit('KR3')`. The TCP/IP communication between `kctserver` and KCT is then established with `kctclient('192.168.1.0')`. The software bounds are set with the commands,

```
>>  B = [450, 650, 200, -200, 0, 500;
>>       -90, 90, -90, 90, -90, 90];
>>  kctsetbound(B);
```

To draw the circle, we generated a matrix $\mathbf{P}$ of points using the following lines of code:

```
>>  k = [0:pi/50:3*pi/2];x = 600*ones(1,length(k));
>>  y = 150*cos(k); z = 150*sin(k) + 310;
>>  P = [x',y',z',repmat([0, 90, 0],length(k),1)];
```

We finally called the function `kctpathxyz(P,tp,1)`, with time step `tp = 1`. Fig. 4(a) shows the trajectory of the end-effector and Fig. 4(b) the time history of the joints angles, as returned by `kctpathxyz` setting to 1 its third argument. Figs. 4(c)-(f) show four snapshots of the real robot during the circular motion.

[1]The videos of the experiments are available at the web page: `http://sirslab.dii.unisi.it/vision/kct`

B. Planar catadioptric stereo: camera rotation estimation

Let us consider the *planar catadioptric stereo* (PCS) system [14] shown in Fig. 5(a). A pinhole camera mounted on the effector of the KUKA manipulator, observes a structured 3-D scene directly and reflected through two planar mirrors. Let us suppose we wish to estimate (using only the visual information), the rotation matrix $\mathbf{R}_w^c$ between the camera frame $\langle x_c, y_c, z_c \rangle$ and the mirrors' reference frame $\langle x_w, y_w, z_w \rangle$ while the camera moves with time (see Fig. 5(b)). By taking advantage of the epipolar geometry between the real camera and the virtual cameras associated to the two mirrors, a closed-form formula for $\mathbf{R}_w^c$ has been determined in [13, Prop. 8]. In order to test the robustness of this solution in a real-world setting, we moved the camera along a given trajectory and compared the estimated and actual roll-pitch-yaw angles of $\mathbf{R}_w^c$. Since we are interested in the camera's rotation, it is convenient to refer robot's motion with respect to the frame of the camera $\langle x_c, y_c, z_c \rangle$ in the initial point of the trajectory, i.e., for $t = 0$. We then performed the following change of frame after the initialization/connection step:

```
>>  H6c = kctrotoz(-90)*kcttran([0, 0, 50]');
>>  H06 = kctfkine(qinit); H0c = H06*H6c;
>>  kctchframe(H0c);
```

where $\mathbf{H}_6^c$ (known) and $\mathbf{H}_0^6$ are the homogeneous transformation matrices between the camera and the end-effector frames, and between the end-effector and the base frames, respectively. `qinit` is the vector of the joint angles of the robot in the initial point of the trajectory. `kctdrivegui` has been used to drive the eye-in-hand robot through other 14 points, where the real and mirror-reflected scenes were clearly visible. The value of the joint angles in the via points has been collected in the 15×6 matrix $\mathbf{Q}$ and the function `[robotinfo, warn] = kctpathjoint(Q,tp,0)` with `tp = 1`, has been called. The actual rotation matrices $\mathbf{R}_w^c$ along the trajectory have been obtained using the following commands (the rigid motion between $\langle x_c, y_c, z_c \rangle$ for $t = 0$ and $\langle x_w, y_w, z_w \rangle$ has been measured in the real setup: it corresponds to a rotation of an angle $\alpha = 90°$ about the x-axis and to

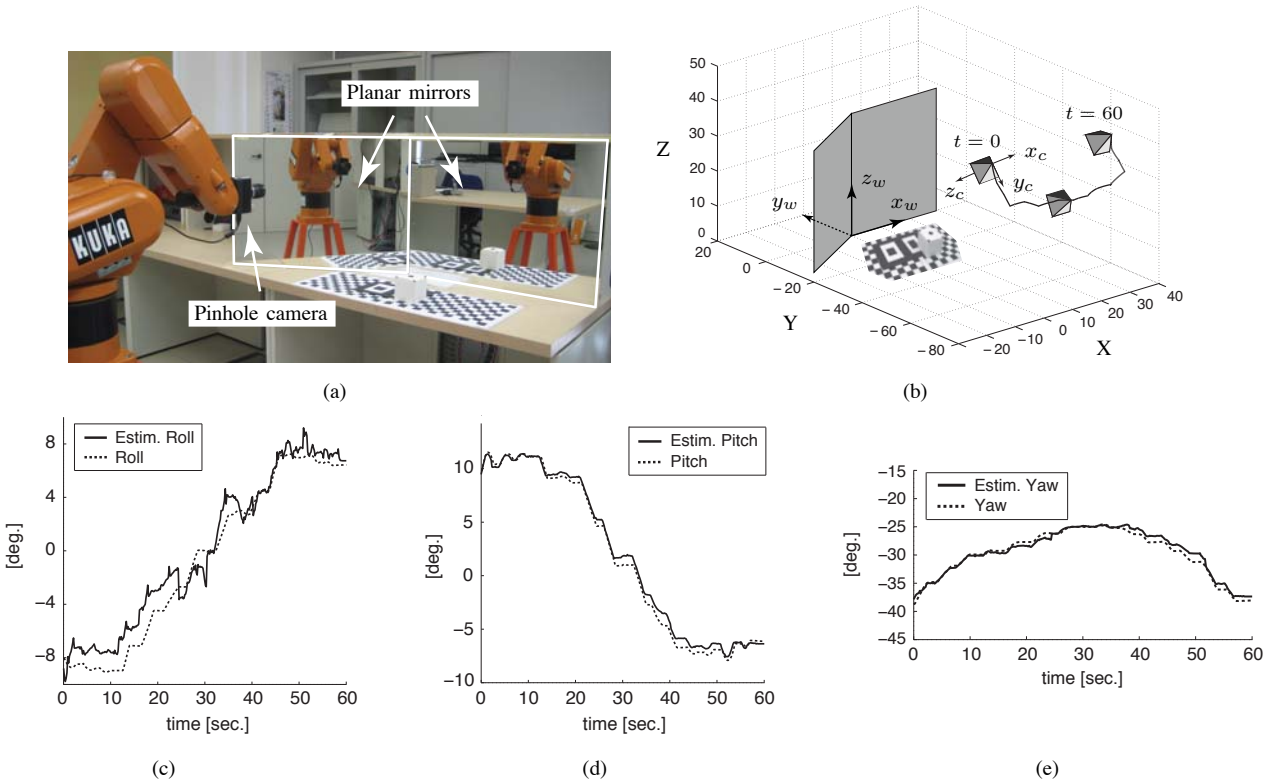


Fig. 5. *Example 2*: (a) PCS experimental setup; (b) 3-D trajectory of the camera with respect to the mirrors; (c)-(e) Time history of the estimated (solid) and actual (dash) roll-pitch-yaw angles of $\mathbf{R}_w^c$.

a translation $\mathbf{d} = [-100,\, 0,\, 850]^T$):

```
>> Hwc_init = kctrotox(alpha)*kcttran(d);
>> for i=1:length(robotinfo)
>>       Hwc = Hwc_init*kctfkine(robotinfo(i,:));
>>       Rwc = Hwc(1:3,1:3);
>> end
```

The time history of the roll-pitch-yaw angles of the matrices $\mathbf{R}_w^c$ thus computed (dash), together with the estimated values (solid), is shown in Figs. 5(c)-5(e).

IV. Conclusions and Future Work

This paper describes an open-source MATLAB toolbox for motion control of KUKA robot manipulators. The *KUKA control toolbox* (KCT) runs on a remote computer connected with the KUKA controller via TCP/IP, and it includes a heterogenous set of functions for kinematics computation, trajectory planning and graphical display. The flexibility and reliability of the toolbox has been demonstrated via two real-word examples. KCT is an ongoing software project: work is in progress to extend the compatibility of the toolbox to all (not necessarily 6 DOF) small and low payload KUKA robots. We also aim to create a robot simulator for off-line validation of motion control tasks and extend the functionality of KCT to the Simulink environment.

References

[1] MATLAB and Simulink for Technical Computing. The MathWorks Inc., USA. [Online]: `http://www.mathworks.com/`.

[2] P.I. Corke. A Robotics Toolbox for MATLAB. *IEEE Rob. Autom. Mag.*, 3(1):24–32, 1996.

[3] K. Yoshida. The SpaceDyn: a MATLAB Toolbox for Space and Mobile Robots. In *Proc. IEEE/RSJ Int. Conf. Intel. Robots Syst*, pages 1633–1638, 1999.

[4] A. Breijs, B. Klaassens, and R. Babuška. Automated design environment for serial industrial manipulators. *Ind. Robot*, 32(1):32–34, 2005.

[5] G.L. Mariottini and D. Prattichizzo. EGT for Multiple View Geometry and Visual Servoing: Robotics and Vision with Pinhole and Panoramic Cameras. *IEEE Robot. Autom. Mag.*, 12(4):26–39, 2005.

[6] P.I. Corke. The Machine Vision Toolbox: a MATLAB toolbox for vision and vision-based control. *IEEE Robot. Autom. Mag.*, 12(4):16–25, 2005.

[7] R. Falconi and C. Melchiorri. RobotiCad: an Educational Tool for Robotics. In *Proc. 17th IFAC World Cong.*, pages 9111–9116, 2008.

[8] W.E. Dixon, D. Moses, I.D. Walker, and D.M. Dawson. A Simulink-Based Robotic Toolkit for Simulation and Control of the PUMA 560 Robot Manipulator. In *Proc. IEEE/RSJ Int. Conf. Intel. Robots Syst*, pages 2202–2207, 2001.

[9] M. Casini, F. Chinello, D. Prattichizzo, and A. Vicino. RACT: a Remote Lab for Robotics Experiments. In *Proc. 17th IFAC World Cong.*, pages 8153–8158, 2008.

[10] KUKA Robotics Corporation [Online]: `http://www.kuka-robotics.com/`.

[11] G. Maletzki, T. Pawletta, S. Pawletta, and B. Lampe. A Model-Based Robot Programming Approach in the MATLAB-Simulink Environment. In *Int. Conf. Manuf. Res.*, pages 377–382, 2006. [Online]. `http://www.mb.hs-wismar.de/~gunnar/software/KukaKRLTbx.html`.

[12] M.W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley, 2006.

[13] G.L. Mariottini, S. Scheggi, F. Morbidi, and D. Prattichizzo. Planar Catadioptric Stereo: Single and Multi-View Geometry for Calibration and Localization. In *Proc. IEEE Int. Conf. Robot. Automat*, pages 1510–1515, 2009.

[14] J. Gluckman and S.K. Nayar. Catadioptric Stereo using Planar Mirrors. *Int. J. Comput. Vision*, 44(1):65–79, 2001.