Using Matlab to Interface
Industrial Robotic & Automation Equipment

J. Norberto Pires
Mechanical Engineering Department
University of Coimbra
Polo II Coimbra, Portugal
http://www.dem.uc.pt/norberto/
norberto@robotics.dem.uc.pt

1. Introduction.
Matlab [1] is a widely used software environment for research and teaching applications on robotics and automation, mainly because it is a powerful linear algebra tool, with a very good collection of toolboxes that extend Matlab basic functionality, and because it is an interactive open environment. In this short paper we present a toolbox that enables access to real robotic and automation (R&A) equipment from the Matlab shell. If used in conjunction with a robotics toolbox [2][3][4] it will extend significantly their application, i.e., besides robotic simulation and data analysis the user can interact on-line with the equipment he or she is using. Our personal experience with this tool shows its usefulness for research applications, but also for teaching projects. With students, using Matlab means taking advantage of the reduced training required to start using it, if we compare with other programming environments and languages we also use (Microsoft Visual C++ or Visual Basic).

This paper is organized as follows: Section 2 briefly presents the software architecture adopted and how it was used to generate the Matlab MEX files presented in the paper. Section 3 and 4 exemplifies application to two types of equipment, complemented with a generic manipulation example using a robotic gripper. In section 5 the utilization of this tool with force control experiments is briefly explored. Finally conclusions are drawn in section 6.

Basically, when we want to use some kind of equipment from a computer we need to write code and define data structures to handle all its functionality. We can then pack the software into libraries, which are not very easy to distribute being language dependant, or build a software control using one of the several standard languages available
(preferably ActiveX [5] or JAVA [6]). Using a software control means implementing methods and data structures that hide from the user all the tricky parts about how to have things done with some equipment, focusing only on using its functions in a easy way. Beside that, those components are easily integrated into new projects built with programming tools that can act as containers of that type of software controls, i.e., they can be added to new projects in a "visual" way.

To interface Matlab the general approach is by using DDE [5]. Since Matlab is an interpreted language and can act as DDE client, it is fairly easy to build a DDE server that can offer to Matlab a collection of services to be accessed using the Matlab DDE implementation [7]. Using DDE is a common and largely used solution to exchange data between applications, even in industrial environments. For each equipment we need to have the specific set of services available from the DDE server, and then build the calling routines as M-files [1]. Simple. But we can do better. Services (client and server sides) can be thought as "options" of a multi-parameter driven function that can be called remotely using some calling and data representation protocols. Matlab MEX functions are parameter driven functions that can be easily used to implement service calls, just by embedding the previous mentioned libraries or software controls within the MEX function. We have done that for several types of R&A equipment building a MEX function (or module) for each one. The collection of modules constitutes the Matlab toolbox MATROBCOM. In conclusion, the basic idea is simple. For each equipment we need to design and build a server (if it is not yet available) to expose the equipment functionality as remote services. The technology to build the server is highly dependent on the equipment resources and computing facilities, but if possible some kind of RPC (Remote Procedure Calls) [8] mechanism should be used. Software controls that explore these services should then be available as basic tools to develop remote and distributed applications using the selected equipment(s). Finally, the Matlab interface to those equipments is built just by embedding the above mentioned software controls into MEX functions.
In the following sections we’ll demonstrate how to implement the above-presented ideas, using two specific equipments: an industrial robot and a force/torque sensor.

The module presented here (MATABBS4) works with any ABB robot equipped with the S4 controller [9]. It implements calls to all services available from the robot controller (RPC servers). ABB S4 controllers implement RPC server programs [10] with variable access services, file and program management services and system status services. To access those services the host computer (client) must implement RPC calling code through an ethernet or serial connection (both using TCP/IP protocols). ABB also developed an Application Specific Protocol (ASP) called RAP [10] to be used with these services. The RAP messaging protocol works in the same way as MMS specified for MAP networks [11]. To implement the PC-Robot communication code based as mentioned on RPC’s using RAP, we used the facility developed by the Sun Microsystems Open Computing (ONC) Group named SUN RPC 4.0 ported to Windows NT 4.0 or later (see Fig.1). User services may be implemented using the basic RPC services, written using the RAPID [9], a robot programming language from ABB Robotics (available at the robot controller). A simple switch-case-do loop driven by a parameter whose value can be changed by the host would do the job. The all list of functions available in the module is presented in Table I.
Table I - Functions available in the MATABBS4 module.

<table>
<thead>
<tr>
<th>Function</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>Opens a communication line with a robot (RPC client)</td>
</tr>
<tr>
<td>close</td>
<td>Closes a communication line.</td>
</tr>
<tr>
<td>motor_on</td>
<td>Go to Run State</td>
</tr>
<tr>
<td>motor_off</td>
<td>Go to Standby State</td>
</tr>
<tr>
<td>prog_stop</td>
<td>Stop running program</td>
</tr>
<tr>
<td>prog_run</td>
<td>Start loaded program</td>
</tr>
<tr>
<td>prog_load</td>
<td>Load named program</td>
</tr>
<tr>
<td>prog_del</td>
<td>Delete loaded program</td>
</tr>
<tr>
<td>prog_set_mode</td>
<td>Set program mode</td>
</tr>
<tr>
<td>prog_get_mode</td>
<td>Read actual program mode</td>
</tr>
<tr>
<td>prog_prep</td>
<td>Prepare Program to Run (Program Counter to begin)</td>
</tr>
<tr>
<td>pgmstate</td>
<td>Get Program Controller State</td>
</tr>
<tr>
<td>ctlstate</td>
<td>Get Controller State</td>
</tr>
<tr>
<td>oprstate</td>
<td>Get Operational State</td>
</tr>
<tr>
<td>sysstate</td>
<td>Get System State</td>
</tr>
<tr>
<td>ctrlvers</td>
<td>Get Controller Version</td>
</tr>
<tr>
<td>ctlid</td>
<td>Get Controller ID</td>
</tr>
<tr>
<td>robpos</td>
<td>Get current robot position</td>
</tr>
<tr>
<td>read_xxxx</td>
<td>Read variable of type xxxx (there are calls for each type of variable defined in RAPID [9])</td>
</tr>
<tr>
<td>read_xdata</td>
<td>Read user defined variables</td>
</tr>
<tr>
<td>write_xxx</td>
<td>Write variable of type xxxx (there are calls for each type of variable defined in RAPID [9])</td>
</tr>
<tr>
<td>write_xdata</td>
<td>Write user defined variables</td>
</tr>
<tr>
<td>digin</td>
<td>Read digital input</td>
</tr>
<tr>
<td>digout</td>
<td>Set digital output</td>
</tr>
<tr>
<td>anain</td>
<td>Read analog input</td>
</tr>
<tr>
<td>anaout</td>
<td>Set analog output</td>
</tr>
</tbody>
</table>

The robot may be connected to the computer using a serial port or preferably an ethernet port, both using TCP/IP protocols. If a local area network is available, several users/computers may be connected to the robot at the same time (with a line or channel open), and then MATABBS4 keeps track of actual open lines/channels. Opening a line means starting a client connection to the RPC servers running on the robot. In the following we will demonstrate a few capabilities of the module MATABBS4.
To open a connection to the robot the command is,

```
>> line = matabbs4 ('open', 'babylon', 'reserved')
line =
    1
```

Name of the robot as declared in the DNS table or system "hosts" file.

Load and start a program,

```
>> matabbs4 ('program_load', 'flp1:\example.prg', line)
ans = 0
```

```
>> matabbs4 ('motor_on', line)
ans = 0
```

If the robot is not in "AUTO MODE" then,

```
>> matabbs4 ('motor_on', line)
Error in 'motor_on' RPC call: Error Code: -1114
    ans = -1114. The returned error code means "Invalid System State", and is printed on ABB manuals.
```

```
>> matabbs4 ('program_prep', line)
ans = 0
```

```
>> matabbs4 ('program_run', line)
ans = 0
```

To check system state the command is,

```
>> matabbs4 ('sysstate', line)
ans =
     0  → Error code (0 = OK).
     2  → Controller State (Stand By).
     4  → Operational State (Auto Mode).
     1  → Number of Programs Loaded (1).
     4  → Program Controller State (Stopped State).
     4  → Program State (Initiated).
    93  → Space Available on System Memory (%).
```

The user should check system state after any system change state command to confirm system change and to determine when to start commanding the system. For example, if
a 'program_run' command is issued the user should wait until the controller state is 'Run State = 4', before starting using the features of the loaded robot program. Checking the system state can be performed in a cycle until a condition is met, or by monitoring the RPC messages from the robot.

To read current robot position the command is,

```
>> matabbs4 ('robpos', line)
ans =
     0   → Error code.
   4.6833e+002 → Cartesian X position (related to base frame).
   2.0689e+002 → Cartesian Y position.
  1.2061e+003 → Cartesian Z position.
   5.2125e-002 → Quaternion (q₁).
  -2.3500e-001 → Quaternion (q₂).
   4.3565e-001 → Quaternion (q₃).
  -8.6733e-001 → Quaternion (q₄).
     0   → Configuration data (ABB CF₁).
     0   → Configuration data (ABB CF₄).
  1.0000e+000 → Configuration data (ABB CF₆).
     0   → Configuration data (ABB CFₓ).
  9.0000e+009 → Position of External Axis 1 (current value = not used).
  9.0000e+009 → Position of External Axis 2 (current value = not used).
  9.0000e+009 → Position of External Axis 3 (current value = not used).
  9.0000e+009 → Position of External Axis 4 (current value = not used).
  9.0000e+009 → Position of External Axis 5 (current value = not used).
  9.0000e+009 → Position of External Axis 6 (current value = not used).
```

Reading and acting on IO,

```
>> matabbs4 ('digin', 5, line) to read the value of digital input 5.
ans = 1    Signal is ON

>> matabbs4 ('digout', 7, 1, line) to activate the digital output 7.
ans = 0

>> matabbs4 ('anain', 2, line) to read the value of analog input 2.
ans = 3453

>> matabbs4 ('anaout', 1, 236, line) to write the value 236 on analog output 1.
ans = 0
```
Acting on program variables defined in RAPID (see Table I) or user defined variables (XDATA type),

```
>> matabbs4 ('write_num', 'name', value, line) writing on a type 'num' variable.
ans = 0
>> matabbs4 ('read_bool', 'name', line)      reading the state of a 'bool' variable.
ans = 1           Variable is TRUE.
```

Suppose that we have a robot program (written in RAPID [9]), which is switched by a variable named, let say, 'decision'. The basic structure of the RAPID program would be something like (using a C-type definition),

```
while never_end;
    switch decision
        case 1: call routine_1; break;
        case 2: call routine_2; break;
        ...
        case n: call routine_n; break;
    end_switch;
end_while;
```

Suppose again that routine_1 moves the robot from actual position to another one defined by a position variable named 'new_pos' of the type robtarget. Routine_1 should then be something like,

```
PROC routine_1
    MoveJ to new_pos;
    decision = -1;
ENDPROC
```

From within Matlab we can then use the MATABBS4 module to command new positions to the robot. Those positions could be defined as a set of joint vectors specifying the joint angles to be achieved. We should then run a forward kinematics function to obtain the positions in Cartesian coordinates, with orientation in quaternions and using the ABB configuration definition. For that we can use any forward kinematics function from one of the available robotics toolboxes [2][3][4] and then convert the 4x4 transformation matrices into 'robtarget' structures. In [4] there is a function that does it all, outputting a matrix with one 'robtarget' per matrix line. If we name that matrix 'new_positions' we could then command it to the robot with the commands,
>> matabbs4 ('write_robtarget', 'new_pos', new_position[1,:], line)
an = 0
>> matabbs4 ('write_num', 'decision', 1, line)
an = 0

checking between each vector if the last command was already performed, i.e., when the following call outputs -1,
>> matabbs4 ('read_num', 'decision', line)

Note: With the new versions of RAPID there is also the possibility to command joint vectors directly using the new 'jointtarget' structures. With the new versions, with a slight modification in routine_1 we can start commanding joint vectors directly (the write jointtarget calls were also implemented in MATABBS4).

Fig. 2 – Two finger pneumatic gripper.

Figure 2 shows a two finger gripper attached to the robot through a tool changer (from ATI Inc., USA). A force/torque sensor is also included (from JR3 Inc., USA), although it is not used in this example. The gripper is based on a pneumatic module (from Zaytran Inc., USA), having an air pressure regulator to control the grasp force. Currently there is no force feedback, so we use an open-loop control approach based on the mapping between the analog input to the air pressure regulator and the force between the fingers. That mapping was obtained by taking a few measures of commanded pressure (digital value to the ADC) and the obtained grasp force, using a force sensor inserted between the two fingers. The mapping function was then obtained using the Matlab function 'polyfit'. This means that before commanding the gripper to close, we need to select the
required analog value (actually the digital value to the ADC of the analog board) 
appropriate to the desired grasp force,

```matlab
>> ana_value = pressure_force_map(required_force)
ana_value = digital value to command to the ADC
```

A simple pick-and-place operation would be:

Open Gripper and approach object,
```matlab
>> matlabs4('digout', 1, 0, line)  
OPEN Gripper
>> matlabs4('write_robtarget', 'new_pos', pos_1, line)  
MOVE pos_1
>> matlabs4('write_num','decision', 1, line)
```

Move to grasp position and pick object,
```matlab
>> matlabs4('write_robtarget', 'new_pos', pos_2, line)  
MOVE pos_2
>> matlabs4('write_num','decision', 1, line)
>> ana_value = pressure_force_map(5)  
PICK object
>> matlabs4('anaout', 1, ana_value, line)
>> matlabs4('digout', 1, 1, line)
```

Approach final position (suppose that from pos_2 to pos_3 there are no obstacles),
```matlab
>> matlabs4('write_robtarget', 'new_pos', pos_3, line)  
MOVE pos_3
>> matlabs4('write_num','decision', 1, line)
```

Move to final position and place object,
```matlab
>> matlabs4('write_robtarget', 'new_pos', pos_4, line)  
MOVE pos_4
>> matlabs4('write_num','decision', 1, line)
>> matlabs4('digout', 1, 0, line)  
PLACE object
```

Move away,
```matlab
>> matlabs4('write_robtarget', 'new_pos', pos_3, line)  
MOVE pos_3
>> matlabs4('write_num','decision', 1, line)
```

This is a very simple example where we didn't care about selecting the velocity, the
position accuracy, the acceleration, etc. All that can be also parameterized remotely
before commanding any motion. For example, if we write routine_1 in the form,
PROC routine_1

SetAcc new_acc;
MoveJ to new_pos, with new_velocity and new_precision;
SetAcc default_acc;
decision = -1;
ENDPROC

where, new_acc, new_velocity and new_precision are parameters that we can change remotely and have predefined default values. In this case, the motion to point 3 above, for example, should be commanded as follows,

```matlab
>> matabbs4 ('write_num', 'new_acc', new_acc, line)   SET Values
>> matabbs4 ('write_speeddata', 'new_velocity', new_speed, line)
>> matabbs4 ('write_zonedata', 'new_precision', new_precision, line)
>> matabbs4 ('write_robtarget', 'new_pos', pos_3, line)   Move pos_3
>> matabbs4 ('write_num', 'decision', 1, line)
```


The module presented here works with force/torque sensors from JR3 Inc [19] (equipped with PC receiver boards). The complete set of functions, listed on Table II, enable the user to explore the sensor functionality from within the Matlab environment. Before trying to use the sensor the user must call the following function,

```matlab
>> matjr3 ('init_jr3')
an = 0     (0 = Operating System detected is WinNT)
```

that detects the running operating system (OS) and initializes IO access if the OS used is Windows NT (a kernel driver should then be present to access the sensor DSP receiver board). In the following some commands are exemplified.

Reset offsets and read full scales,
```matlab
>> matjr3 ('reset_offsets')
>> fs= matjr3 ('get_full_scales')
```
```
fs = 191  196  520  134  133  182  520  182
     Fx    Fy    Fz    Mx    My    Mz    V1    V2
```
Get recommended maximum full scales,
>> rfs = matjr3('get_rec_full_scales', 1)
rfs = 191 196 520 134 133 182 520 182
   Fx  Fy  Fz  Mx  My  Mz  V1  V2

Set full scales,
>> fs = matjr3('set_full_scales', rfs)
>> off = matjr3('read_offsets')
off = 1764 584 264 34 1368 418
   Fx  Fy  Fz  Mx  My  Mz

Read force/torque data from filter 2,
>> fs = matjr3('read_ftdata', 2)
rfs = 1 0 0 1 -1 1 0 1
   Fx  Fy  Fz  Mx  My  Mz  V1  V2

5. Force Control Experiments
With this setup we can easily program explicit indirect force control experiments, taking advantage of the mathematical and graphical capabilities of Matlab+Simulink. Fig. 3 represents an explicit force control scheme, where the force control law generates position accommodation commands to the internal position controller (indirect force control approach [13][18]).

The controller input variables are the force/torque error e(k) and error variation de(k):

\[ e(k) = f_a(k) - f_d(k) \]
\[ de(k) = e(k) - e(k-1) \]  \hspace{1cm} (1)

where \( f_a(k) \) is the measured wrench and \( f_d(k) \) is the desired wrench. The controller output is the position/orientation accommodation, assumed to be small:
\[
\begin{align*}
{^6}_{\text{di}}\Delta x_{\text{di}}(k) &= {^6}_{\text{F}}R_{\text{F}}{^6}_{\text{di}}\Delta x_{\text{di}}(k) \\
{^6}_{\text{di}}\Delta \phi_{\text{di}}(k) &= {^6}_{\text{F}}R_{\text{F}}{^6}_{\text{di}}\Delta \phi_{\text{di}}(k)
\end{align*}
\] (2)

where \( {^6}_{\text{di}}\Delta x_{\text{di}}(k) \) is the position accommodation, \( {^6}_{\text{di}}\Delta \phi_{\text{di}}(k) \) is the orientation accommodation and \( \{F\} \) and \( \{6\} \) are the frames associated with the wrist force/torque sensor and robot tip, respectively.

Table II - Functions available in the MATJR3 module.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_jr3</td>
<td>This function detects your operating system (OS) and calls the 'jr3io.sys' driver if you are using Windows NT (recommended OS). If you are using other OS the driver is not called.</td>
</tr>
<tr>
<td>read</td>
<td>Reads from a receiver board memory address.</td>
</tr>
<tr>
<td>write</td>
<td>Writes to a receiver board memory address.</td>
</tr>
<tr>
<td>system_warnings</td>
<td>Reads system saturation warnings (board memory address WARNINGS).</td>
</tr>
<tr>
<td>system_errors</td>
<td>Reads system errors (board memory address ERRORS).</td>
</tr>
<tr>
<td>command</td>
<td>Commands JR3 receiver board.</td>
</tr>
<tr>
<td>get_threshold_status</td>
<td>Gets the value of the threshold bits (board address THRESHOLD).</td>
</tr>
<tr>
<td>reset_threshold</td>
<td>Reset the threshold bits.</td>
</tr>
<tr>
<td>read_ftdata</td>
<td>Reads force/torque data from receiver board.</td>
</tr>
<tr>
<td>set_transforms</td>
<td>Sets a new transformation definition.</td>
</tr>
<tr>
<td>use_transforms</td>
<td>Selects the transformation to use.</td>
</tr>
<tr>
<td>read_offsets</td>
<td>Read offsets in use.</td>
</tr>
<tr>
<td>set_offsets</td>
<td>Set actual offsets, using the current offset index.</td>
</tr>
<tr>
<td>change_offset_num</td>
<td>Changes actual offset index (num).</td>
</tr>
<tr>
<td>reset_offsets</td>
<td>Set actual offsets to the current values read from FILTER_2.</td>
</tr>
<tr>
<td>use_offset</td>
<td>Changes actual offsets to the one defined.</td>
</tr>
<tr>
<td>peak_data</td>
<td>Set address to watch for peaks.</td>
</tr>
<tr>
<td>peak_data_reset</td>
<td>Set address to watch for peaks and resets internal values to current data.</td>
</tr>
<tr>
<td>read_peaks</td>
<td>Reads current peak values.</td>
</tr>
<tr>
<td>bit_set</td>
<td>Set bits on defined bit-map.</td>
</tr>
<tr>
<td>set_full_scales</td>
<td>Set JR3 Full_Scales.</td>
</tr>
<tr>
<td>get_full_scales</td>
<td>Reads actual full_scales.</td>
</tr>
<tr>
<td>get_recommended_full_scales</td>
<td>Reads recommended full_scales.</td>
</tr>
<tr>
<td>sensor_info</td>
<td>Reads information from the sensor and from the receiver board. Use this function to test your setup.</td>
</tr>
</tbody>
</table>
The force control function may implement several force control approaches under the indirect force control framework: classical approaches (PID controllers), fuzzy, etc. Nevertheless, the following conditions should be considered:

a) Simplicity. The force control law must be simple and easy/faster to compute to enable real time.

b) PI-type control. If a null steady state error is to be achieved a PI-type force control law should be selected. The derivative term is not desirable due to the noise associated with force readings.

c) Robustness to environment stiffness. The system should work with several environment stiffness constants (the environment is modeled as a linear spring), which generally happens under industrial environments.

d) Implementation requirements should not include significant changes to the original control system.

As a test example we use a fuzzy-PI force control law [14][15][16] on a unidirectional force control case. A Fuzzy Logic Controller (FLC) is a control law described by a knowledge base (defined with simple IF ... THEN type rules over vaguely defined - fuzzy variables) and an inference mechanism to obtain the current output control value. The designed FLC has two inputs (e(k) and de(k)) and one output (dx_d(k)). The inputs are divided in levels (Table III) in accordance with the observed sensor characteristics and fuzzyfied using triangular membership functions of unitary overlap [14]. The output is fuzzyfied in the same way. The rule base is constructed using a methodology similar to [14] and [15]. Their 2D-lookup table is used as template to generate our fuzzy-PI 2D lookup table, which was obtained using the Matlab Fuzzy Toolbox, considering a Mandani-type inference mechanism. Tuning is achieved by proper selection of the variables k_x and k_e (k_d is kept constant), by simulation of several contact situations (stiffness and desired force) [13]. The output value obtained is scaled by the robot position resolution (k<0.1 mm) before being commanded to the robot. Nevertheless, a better tuning procedure should be implemented to cope with environment stiffness variations.

To command position accommodations to the robot, a routine like the one above (presented in a simplified form) should be running in the robot controller.

PROC contact
  WHILE not_done & not_abort
    IF (delta_z <> 0 OR delta_y <> 0 OR delta_x <> 0) THEN
new_pos.x = new_pos.x + delta_x;
new_pos.y = new_pos.y + delta_y;
new_pos.z = new_pos.z + delta_z;
new_pos.x = 0;
new_pos.x = 0;
new_pos.x = 0;
ENDIF
MoveJ to new_pos;
notify_end = NOT notify_end;
ENDWHILE
ENDPROC

Fig.4 - Aspect of the experimental setup.

The control cycle implemented in Matlab starts reading the actual force, then computes the position accommodation, commands the accommodation to the robot and waits to the end_of_command message (sent when notify_end changes state). The robot makes an RPC call (named spontaneous message) to a RPC server running in the PC [12][13]. (with information about the event occurred in the robot controller). The RPC fires a DDEPoke command to Matlab if the event is a change in the RAPID variable notify_end; if a DDE connection is established Matlab can detect the message and start the next cycle. Another approach is to check the variable notify_end until it changes state (polling), as already mentioned. In conclusion, there are two different ways of synchronization: RPC messages or polling. The first one enables an automatic synchronization keeping Matlab free for other tasks between cycles. The polling mechanism is based on RPC calls made to the robot controller, which means that they can slow down the robot if made intensively in a routine cycle.
The experimental setup, the force profile commanded to the robot and the measured force are presented in fig. 5.

Fig. 5 - Force control results using a fuzzy_PI control law [13]: desired force is a 20N step.

**Table III.** Levels for fuzzy variables: $a_1$, $b_1$, $a_2$ and $b_2$ are set by the user to define the dead zone. The distribution is similar for the negative ranges (in that case the levels are also negative).

<table>
<thead>
<tr>
<th>$e(k)/fd$ (%)</th>
<th>$de(k)/fd$ (%)</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1 \rightarrow b_1$</td>
<td>$a_2 \rightarrow b_2$</td>
<td>0</td>
</tr>
<tr>
<td>$a_1 \rightarrow 5$</td>
<td>$a_2 \rightarrow 5$</td>
<td>+1</td>
</tr>
<tr>
<td>5 $\rightarrow$ 10</td>
<td>5 $\rightarrow$ 10</td>
<td>+2</td>
</tr>
<tr>
<td>10 $\rightarrow$ 15</td>
<td>10 $\rightarrow$ 15</td>
<td>+3</td>
</tr>
<tr>
<td>15 $\rightarrow$ 20</td>
<td>15 $\rightarrow$ 20</td>
<td>+4</td>
</tr>
<tr>
<td>20 $\rightarrow$ 30</td>
<td>20 $\rightarrow$ 25</td>
<td>+5</td>
</tr>
<tr>
<td>30 $\rightarrow$ 40</td>
<td>25 $\rightarrow$ 30</td>
<td>+6</td>
</tr>
<tr>
<td>40 $\rightarrow$ 50</td>
<td>30 $\rightarrow$ 35</td>
<td>+7</td>
</tr>
<tr>
<td>50 $\rightarrow$ 75</td>
<td>35 $\rightarrow$ 50</td>
<td>+8</td>
</tr>
<tr>
<td>$&gt; 75$</td>
<td>$&gt; 50$</td>
<td>+9</td>
</tr>
</tbody>
</table>
The discussion of force control approaches and their implementation details is out of the scope of this paper. Nevertheless, the interested reader can find some application inside and results of a fuzzy-PI controller in [13].

6. Conclusion.
In this short paper we showed how to use R&A equipment from within Matlab, based on a distributed software architecture that may be used to include support to other type of equipment. In fact we built with it several other modules, not presented in the paper, to handle CCD cameras, PLCs and microcontrollers. With this type of interface even non-trained users can easily use actual R&A equipment, since Matlab is a fairly simple environment to use. This is particularly true with students, as our personal experience clearly show. The ideas presented here were then applied to an industrial robot and to an intelligent sensor (force/torque sensor), and demonstrated with several examples. Finally a force control experiment was briefly introduced with the objective of demonstrating an R&D application of the presented setup.

Note: Instructions for download.
The software presented here along with other related tools may be freely downloaded from the site http://www.dem.uc.pt/norberto/.