THE MATLAB TOOLBOX FOR MODELING COMPLEX MECHANISMS

I. PAJĄK
University of Zielona Góra
Institute of Computer Science and Production Management
Licealna 9, 65-417 Zielona Góra, POLAND
E-mails: i.pajak@iizp.uz.zgora.pl

In the paper, a new version of Matlab toolbox for modeling and simulation any mechanisms consisting of open kinematic chains is presented. This tool allows to define any manipulator described by Denavit-Hartenberg parameters and then connect the mechanisms created in this way into one complex mechanism. The package can be used to realistic visualization of robot motion necessary in research, didactic process or during design of a production cell. Available functions render it possible to show a realistic model of a mechanism based on both default and user defined appearance. The toolbox also provides a set of tools for performing simultaneous simulations of many robots and other objects in their workspace. The example illustrating a new features of the toolbox, presenting modeling and motion simulating of human upper limb is also attached.

Key words: robot toolbox, robot modeling, simulation, Matlab.

1. Introduction

Preparation of computer simulation of the robots' motions is a difficult and time consuming process. Such a task is even more difficult in the case of simulation of complicated mechanisms such as human limb. However, visualization of robots and tasks realized by them plays an important role in many applications. The validation of research results is not always possible by performing real experiments using a real robot in its environment. Even if a real manipulator is available simulation tools make it possible to verify and compare obtained solution using many different mechanisms in any designed workspace. Moreover, while teaching the basis of robotics work with real robots it is not always possible, so simulation tools are needed. In a virtual environment a student can learn issues related to the description of position and orientation in 3D space, convention used to describe the manipulator kinematics and, finally, build any mechanism. Additionally, computer modeling is useful during programming industrial robot tasks. The simulation tools allow to design and verify a production cell in a virtual environment without using mechanical robots and other equipment. In the applications presented above performing a simulation using simplified models may by insufficient, often it is important that the robot and objects in its environment look like a real workspace.

There are a lot of tools which can be used in the applications presented above. The robot manufactures provide their own software allowing simulation of the robots supplied by them (Fanuc (2011), ABB (2011)). There are also more versatile solutions which allow simulation of a broader class of mechanisms. Some of these tools are implemented as standalone applications or libraries for certain programming languages (López-Nicolás *et al*. (2009), Cakir and Butun (2007), Gourdeau (1997), Bruyninckx (2001), Bingul *et al*. (2002)). However, in robotics, both research and education are often carried out using complete environments for high-level programming and it is convenient that the simulation tool is integrated with such an environment. One of the most popular applications is MathWorks product Matlab (MathWorks (2010)) but, unfortunately, it does not have specialized toolbox for robotics. There are some independent libraries covering that gap, such as Robotic Toolbox for Matlab (Corke (1996)), SpaceLib (Legnani (2006)) and ROBOLAB (Kucuk and Bingul (2010)). These libraries provide many useful tools necessary for robotic modeling and simulations, but they show poor graphics capabilities or are limited to a narrow class of mechanisms (Kucuk and Bingul (2010)). Additionally, the tools mentioned above do not allow to define and simulate complex mechanisms at all.

In this paper, the Matlab toolbox for modeling and simulation of any mechanisms consisting of open kinematic chains is presented. This tool allows to define any manipulator described by Denavit-Hartenberg parameters and then connect the mechanisms created this way into one complex mechanism. The main advantage of that toolbox, in comparison to toolboxes mentioned above, is the possibility of making simulation of any open kinematic chain. Furthermore, a user can easily show mechanisms using default 3D appearance in a realistic manner or can define his/her own model corresponding to the real construction. The toolbox also provides a set of basic blocks and functions for manipulating position and orientation of those objects in 3D space. Using previously created elements it is possible to define the appearance of the complex production cell including many robots and objects in their workspace. Additionally, it is possible to perform the simulation of this cell showing the motions of all robots and other objects simultaneously. All motions can be calculated using one of the classical methods of trajectory planning in the configuration space.

2. Overview of the toolbox

2.1. Description of the manipulator and its workspace

In the work, we assumed each solid is described in its own local coordinate system. Hence, the position and orientation of that coordinate system sets the location of the solid precisely. To determine the position of the local coordinate system in any reference system 3×1 translation vector describing the position of the origin of the local system is required. To describe the orientation of the solid coordinate system in the reference system a 3×3 orientation matrix may be used. In a compact notation the translation vector and orientation matrix may be combined together in 4×4 homogeneous transformation matrix (Craig (1993), Pająk and Pająk (2011)).

In robotics applications, a Denavit-Hartenberg convention is commonly used to describe the manipulator kinematics. In the toolbox discussed herein, the convention named “modified Denavit-Hartenberg”, described by Craig (1993), is used to determine DH parameters. In this case a coordinate system is assigned to each joint of the manipulator in such away as the z-axis of the coordinate system $\{i\}$ is placed along the axis of joint *i*, the *x*-axis is parallel to the normal to axes of joint *i* and *i+1*, the *y*-axis completes the right-handed coordinate system. The DH parameters in this convention are defined as follows:

link length (a_i) – distance between Z_i and Z_{i+1} along X_i,

link twist (α_i) - angle between Z_i and Z_{i+1} around X_i,

joint offset (d_i) - distance between X_{i-1} and X_i along Z_i,

joint angle (θ_i) - angle between X_{i-1} and X_i around Z_i.

2.2. Basic functions

The set of basic functions in the new version of the toolbox did not change comparing to the previous one. The new way of using objects is presented in section 3.1. As in the previous version, the toolbox provides three classes of basic objects: coordinate systems, blocks and manipulators. The coordinate system makes it possible to understand the description of the position and orientation in 3D space by homogenous transformation. The objects of this class play a supporting role and they are mainly used in the teaching process. The blocks are fundamental objects in the toolbox, they are used to define the appearance of manipulators and may be used to create a robot’s workspace. Each block is approximated by polyhedron, the user can use one of predefined blocks such as cone, cuboid, cylinder, prism, sphere, etc. or he/she can define any block by specifying its vertices and faces. The objects of class manipulator have a crucial importance for the toolbox. The set of available functions allows to define any mechanism described by DH parameters and it allows to show manipulator using default 3D appearance easily.

In order to define the objects mentioned above the function in general form has to be used:

```
hnd = name_of_object(<required_parameters>, <properties>)
```

where: `hnd` – handle to the new object; `<required_parameters>` – the list of the function's arguments which are necessary to define the object; `<properties>` – the list of optional parameters in the form of `property_name`, `property_value` specifying the appearance of the object.

Consequently, a coordinate system object may be created by calling the following function

```
cs = csys(T)
```

where: `cs` – handle to the coordinate system object; `T` - 4×4 transformation matrix determining position and orientation.

The primary function used to define a block object is

```
b = block(p, f)
```

where: `b` – handle to the block object, `p` - a column matrix containing the *x*, *y*, *z* coordinates for each vertex of the block; `f` - a cell array containing connection matrices specifying which vertices in the `p` are connected with faces.

Using the function `block` requires determination of each point and each face of the block, so it is laborious. Therefore, an additional set of functions defining elementary blocks has been implemented: `cone`, `cube`, `cuboid`, `cylinder`, `polygon`, `polyline`, `prism`, `pyramid`, `sphere`, `srevolution`, `tetrahedron`. Additionally, the complex blocks can be created by merging previously defined blocks using the function `block_merge`. A detailed description of those function is included in the documentation.

The basic function which allows defining the manipulator is

```
r = robot(dh, joints)
```

where: `r` – handle to the manipulator object; `dh` - $4\times n$ matrix containing DH parameters set according to convention presented in section 2.1, n - number of joints; `joints` - n-element character array containing joints' types, a single joint can be defined as rotate (`'r'`) or prismatic (`'p'` or `'d'`).

Additionally, each of the object has a set of optional properties specifying its appearance, which mainly determine the way to plot the object in the Matlab figure window. The characteristic properties for each of the object are described in detail in the toolbox documentation, some of them have been presented in Pająk and Pająk (2011).

For each of the objects, discussed earlier, a set of operations has been defined. The individual operations are available by calling the functions in general form:

```
object_operation(hnd, <parameters>)
```

where: `object` – name of the object (`csys`, `block` or `robot`); `operation` – name of the operation; `hnd` – handle to the object; `<parameters>` – the list of the function's arguments which are necessary to perform the `operation`.

For example, each object can by plotted in a Matlab figure window using functions: `csys_plot(cs)`, `block_plot(b)`, `robot_plot(r)` and then it may be removed using: `csys_delete(cs)`, `block_delete(b)`, `robot_delete(r)`. The local coordinate systems connected to the objects can be plotted by calling `block_plot_csys(b)`, `robot_plot_csys(r)`, the position and/or orientation of any coordinate system and block may be changed using functions `csys_move(cs, T)`, `block_move(b, T)`, and the configuration of the manipulator may be set by calling `robot_config(r, config)`. The description of the mentioned above and other functions is available in Pająk and Pająk (2011) and the documentation provided with the toolbox.

3. New features

3.1. Variables versus handles

In the first version of the toolbox all of the objects (coordinate systems, blocks and robots) were represented as the variables, which resulted in a lot of problems. Each modification of the object (i.e. changing the position, new configuration of the manipulator, changing the appearance or even plotting the object in figure window) required to create a new variable with the modified parameters of the object. The functions realizing such operations had to return parameter corresponding to the modified object which next had to be assigned to the variable representing the new object. Additionally, the change of the object parameters did not cause the modification of its graphical representation, so the user had to delete the object from the figure window and he/she had to plot it again. The code placed below shows the example in which the cube is created, next it is plotted in figure window and after that it is moved to the new position:

```
1   c = cube(1);
2   c = block_plot(c);
3   c = block_move(c, [1;1;1]);
4   c = block_delete(c);
5   c = block_plot(c);
```

In the first line function `cube` returns the structure representing the block object, in the lines 2-5 this block is modified, so the modified structure is created in each step and it has to be assigned to the variable representing the object. In the third line the cube is moved but its graphical representation does not change automatically, to refresh the content of the figure window deleting the object and plotting it again it is required to (lines 4-5).

In the new version of the toolbox all of the objects are represented as handles to the global structures managed automatically by the program. Each function creating the object and realizing any operation uses the handle to the object only. In such a way any operation does not create the new variable, because it modifies the same structure and the handle created during creation of the object is connected to the same structure all the time. Additionally, the change of the object parameters causes the instantaneous modification of its graphical representation. So the example mentioned above in the new version of the toolbox is realized as follows:

```
1   c = cube(1);
2   block_plot(c);
3   block_move(c, [1;1;1]);
```

In the first line function `cube` returns the handle to the block object, which next is plotted in the figure window. In the third line the cube is moved and its graphical representation changes automatically, so refreshing is not needed.

3.2. Complex objects

The main functional changes in the toolbox are related to possibility of creating complex objects: blocks and manipulators. The complex object is an object composed of the several basic objects of the same type. Each component object retains individual properties associated with its appearance (such as colors, line width, local coordinate systems parameters), but is managed by parent object and each operation on parent object applies to all of its component objects.

The first of the new complex object is a complex block. To create the object of this type the following function should be used:

```
[B] = block_complex(B1, B2, B3, ...)
```

where: `B` – a handle to the new complex block; `B1`, `B2`, `B3` – the handles to the component blocks.

This function is similar to the previously existing function `block_merge`, which also allows to connect several blocks, however, there are significant differences between these functions. The function `block_merge` copies all vertices and faces of the merged blocks into the single basic object, so the new block does not have component objects and it is not possible to change properties of individual blocks. In contrast to this function, `block_complex` creates new object composed of several blocks which remain separate objects. In this case, each of the component object has a set of individual properties independent of the parent block properties.

In the new version of the toolbox the complex blocks are used internally to define the appearance of manipulator joints (both default 3D view and user 3D view) and each joint, base and grasper are created as complex blocks.

The second of the complex object is a complex robot which allow to connect several manipulator objects. To create such a mechanism the following function should be used:

```
[R] = robot_complex(R1, R2, R3, ...)
```

where: `R` – a handle to the new complex robot; `R1`, `R2`, `R3` – the handles to the component manipulators.

This object is a set of manipulators which can have independent configurations and keep individual properties. Each operation available for a single manipulator is also allowed to the complex robot. The configuration vector of this complex mechanism consists of the configuration vectors of all component manipulators in order they were passed to the function `robot_complex`.

Additionally, in the new version of the toolbox, there is a new function which does not create a new object, but it allows to attach one mechanism to the last segment of the other one:

```
[R] = robot_connect(R, R1, R2, R3, ...)
```

where: `R` – a handle to the manipulator, other mechanism will be connected to it; `R1`, `R2`, `R3` – the handles of robots to be connected, first all the robots will be connected into a complex robot, next this mechanism will be attached to the last segment of the manipulator `R`.

The differences in the use of functions `robot_complex` and `robot_connect` can be explained by the example of the hand, fingers and upper limb. The hand can be considered as the set of five independent manipulators – fingers, so in the toolbox to construct such a mechanism the function `robot_complex` should be used. The limb can be considered as a manipulator ended by a hand, so this mechanism has to be constructed using the function `robot_connect`. This example will be discussed in the detailed manner in the Section 4.

3.3. Motion simulation

In the new version of the toolbox all functions realizing motion simulations, available in previous version, are still accessible, but they have been modified to allow simulating the complex mechanisms. Additionally, these functions can be used to simulate the motions of many objects of the same type. Moreover, in the new version it is possible to present the motions of many objects of different types.

The basic function, which allows to change the configuration of the robot is:

```
robot_config(R, conf)
```

where: `R` – a handle to the robot; `conf` – vector containing the new configuration of the robot.

In the current version, the input argument `R` can be a handle to the single manipulator or handle to the complex mechanism (defined by `robot_complex` and/or `robot_connect`). In the case of complex mechanisms, single configuration is described by configurations of all components. The configuration vector contains configuration of each component robot in order resulting from the structure of the mechanism. Additionally, it is worth noticing that, regardless of the type of the mechanism, the change of the robot configuration causes the instantaneous modification of its graphical representation.

The toolbox provides also the function which enables to show manipulator realizing the pre-planned trajectory:

```
robot_motion(R1,q1, R2,q2, ..., delay)
```

where: `R`, `q` – the handle to the robot and matrix containing its pre-planned trajectory; the function accepts any number of pairs `R`, `q`; `delay` - pause between animation steps.

The input arguments `R1`, `R2`, ... can be robots of any type, in the case of the complex mechanisms a single configuration should be defined in the same way as in the function `robot_config`. The trajectories of the robots are not parameterized by time, in the *i*-th animation step the robots are plotted in configurations included in the *i*-th columns of the matrices `q1`, `q2`, ... The number of animation steps is equal to the number of configurations of the longest trajectory.

In similar way it is possible to show the motions of the blocks:

```
block_motion(B1,T1, B2,T2, ..., delay)
```

where: `B`, `T` – the handle to the block and cell array containing transformation matrices representing position and orientation of the block in each animation step; the function accepts any number of pairs `B`, `T`; `delay` - pause between animation steps.

A complex simulation of many different types objects can be performed using function:

```
animation(t, Obj1,f1, Obj2,f2, ..., delay, trace, args)
```

where: `t` – the time vector consists of animation timestamp; `Obj`, `f` – the handle to the toolbox object (`block`, `csys`, `robot`) and the function determining configuration of `Obj` for a given time instant; `animation` accepts any number of pairs `Obj`, `f`; `delay` - pause between animation steps; `trace` – Boolean value determining if animated objects have to leave their traces; `args` – additional arguments passed to functions `f1`,`f2`,...

As opposed to the functions mentioned above, `animation` allows to perform the simulation parameterized by time. The number of animation steps is equal to the number of the timestamps in vector `t` (the length of the vector `t`). In each time instant `animation` calls the functions `f1`, `f2`, ... of the form:

```
function [conf] = fun(Obj, t, flag, args).
```

These functions determine the configuration `conf` of the object `Obj` for the time `t`. The argument `flag` is equal to `'init'` for the first call of the function and it is empty otherwise, `args` are additional arguments passed by function `animation` if user determined them. The type of the output argument `conf` depends on the type of the object `Obj`: for blocks and coordinate systems `conf` has to be a transformation matrix, for robots `conf` should contain a configuration vector of mechanism.

As in the first version of the toolbox, for all objects of the robot type (defined by `robot`, `robot_complex`, `robot_connect`), a special panel which allows to change a configuration of the mechanism is available. In the new version this panel is placed in the separate Matlab figure window and it is adapted to manipulate complex mechanism. In the case of a simple mechanism, such as a manipulator, the panel contains one group of controls and each control corresponds to a single configuration variable, if robot is a complex mechanism, the panel contains as many groups as there are components of the robot. Additionally, due to the usage of the handles, each modification of robot configuration (performed by functions described above) causes the immediate modification of controls placed on the panel.

4. Examples

In that section an example of complex mechanism such as human right upper limb is presented. At the beginning, the kinematics of the finger is discussed and then, using the presented toolbox, this mechanism is modeled. In the next step, five fingers are connected into one complex mechanism – a hand. Finally, the human upper limb is modeled as a 7 DOF manipulator and the hand is attached at its end.

4.1. Models of the fingers

From the kinematic point of view, a finger may be considered as a 4 DOF manipulator (Li *et al.* (2011)). Fig. 1 presents a thumb and a forefinger, each cylinder it this figure represents single revolute joint, local coordinate systems attached to each joint are also presented.

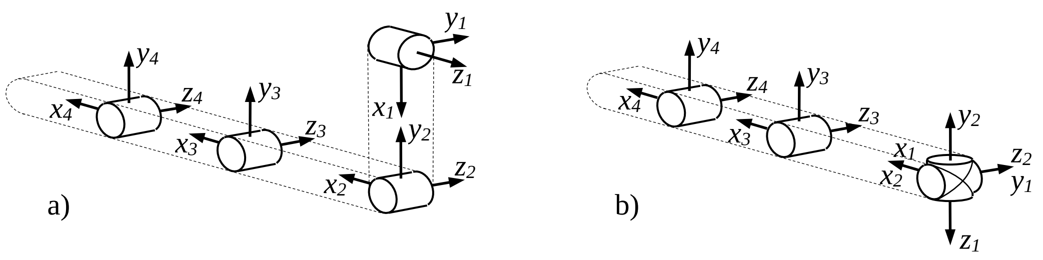


Fig. 1. Models of the fingers, a) a thumb, b) a forefinger.

The Denavit-Hartenberg parameters of the forefinger determined basing on the Fig 1b, are shown in the Table 1.

Table 1. DH parameters of the forefinger.

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	π	0	0	θ_1
2	$-\pi/2$	0	0	θ_2
3	0	l_2	0	θ_3
4	0	l_3	0	θ_4

The DH parameters of the other fingers (with the exception of the thumb) are similar, the differences are related to the link lengths l_2 and l_3. To model the thumb, according to the Fig. 1a, non-zero length of the first link is determined, so DH parameters of this finger are as follows:

Table 2. DH parameters of the thumb.

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	π	0	0	θ_1
2	$-\pi/2$	l_1	0	θ_2
3	0	l_2	0	θ_3
4	0	l_3	0	θ_4

According to DH parameters presented above, using function `robot` described in section 2.2, any finger may be modeled as a 4R manipulator in the following way:

```
1   f = robot([dh(pi,0,0,q1);dh(-pi/2,l1,0,q2);dh(0,l2,0,q3);dh(0,l3,0,q4)],...
2              ['r', 'r', 'r', 'r'],...
3              'RobotGrasper', [l4; 0; 0], 'RobotGrasperSize', 0);
```

The first argument in the above function call is an array containing DH parameters according to the convention presented in section 2.1, the second argument specifies the type of manipulator joints (in that case all the joints are revolute). The arguments given in line 3 are optional properties determining the appearance of the manipulator grasper. In this case, the grasper does not grip anything, it is used only to model a fingertip in the default views. If the own 3D view of the finger is defined this element is not needed. The definitions of the individual fingers will vary in the lengths l_1 - l_4.

The forefinger and the thumb plotted using the function `robot_plot` are shown in Fig. 2.

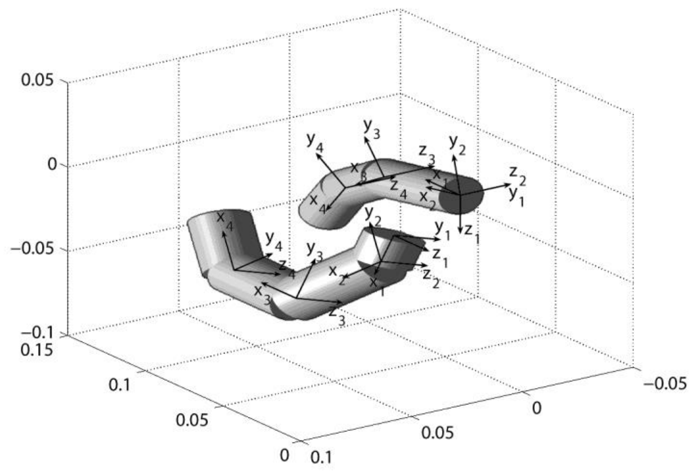


Fig. 2. Forefinger and thumb in default 3D view

4.2. Model of the hand

Using the general definition of the manipulator-finger presented above, a right hand can be constructed. To model such a complex mechanism as a hand, the toolbox function `robot_complex` should be used. Firstly, the component fingers have to be placed in the positions and orientations resulting from the project of the planned mechanism (function `robot_complex` connects the components without changing their locations). The general definition of the finger, presented in the previous section, has to be extended to the setting of the base for each finger - this parameter can be set using property `RobotBase`. Finally, the Matlab code for creating the model of the hand is given below (all dimensions are in meters):

```
 1 q1 = -pi/8;  q2 = pi/4;  q3 = pi/4;  q4 = pi/4;
 2
 3 f1 = robot([dh(pi,0,0,q1); dh(-pi/2,0.015,0,q2); dh(0,0.05,0,q3);...
 4              dh(0,0.04,0,q4)], ['r', 'r', 'r', 'r'],...
 5             'RobotBase', rp2t(roty(pi/2),[0.04;0.03;-0.015]),...
 6             'RobotGrasper', [0.03; 0; 0], 'RobotGrasperSize', 0,...
 7             'RobotCylinderRadius', 0.012);
 8
 9 q1 = 0;  q2 = -pi/16;  q3 = -pi/8;  q4 = -pi/8;
10
11 f2 = robot([dh(pi,0,0,q1); dh(-pi/2,0,0,q2); dh(0,0.05,0,q3);...
12              dh(0,0.03,0,q4)], ['r', 'r', 'r', 'r'],...
13             'RobotBase', [0.095;0.022;0],...
14             'RobotGrasper', [0.025; 0; 0], 'RobotGrasperSize', 0,...
15             'RobotCylinderRadius', 0.01);
16 f3 = robot([dh(pi,0,0,q1); dh(-pi/2,0,0,q2); dh(0,0.055,0,q3);...
17              dh(0,0.035,0,q4)], ['r', 'r', 'r', 'r'],...
18             'RobotBase', [0.095;0;0],...
19             'RobotGrasper', [0.025; 0; 0], 'RobotGrasperSize', 0,...
20             'RobotCylinderRadius', 0.01);
21 f4 = robot([dh(pi,0,0,q1); dh(-pi/2,0,0,q2); dh(0,0.05,0,q3);...
22              dh(0,0.03,0,q4)], ['r', 'r', 'r', 'r'],...
23             'RobotBase', [0.09;-0.02;0],...
24             'RobotGrasper', [0.025; 0; 0], 'RobotGrasperSize', 0,...
25             'RobotCylinderRadius', 0.009);
26 f5 = robot([dh(pi,0,0,q1); dh(-pi/2,0,0,q2); dh(0,0.04,0,q3);...
27              dh(0,0.025,0,q4)], ['r', 'r', 'r', 'r'],...
28             'RobotBase', [0.085;-0.038;0],...
29             'RobotGrasper', [0.023; 0; 0], 'RobotGrasperSize', 0,...
30             'RobotCylinderRadius', 0.008);
31
32 rhand = robot_complex(f1, f2, f3, f4, f5, ...
33                       'RobotBase', [0;0;0], ...
34                       'RobotModel', 'configex', ...
35                       'RobotCylinderRadius', 0.01);
```

The lines 3-7 define the thumb and set it in the initial configuration specified in line 1. The definitions of the other fingers are given in lines 11-30, the initial configurations of these fingers are the same and determined in line 9. It is worth noticing that due to opposable thumbs, configurations of the thumb and the other fingers are different. For the same reason, the base of the thumb is rotated by $\pi/2$ radians. Additionally, for each finger property `RobotCylinderRadius` is set to match the thickness of the fingers to assumed dimensions of the hand. In lines 32-35 using the function `robot_complex`, fingers `f1`, ... `f5` are connected into one complex mechanism – a hand. In default views this function adds additional elements which connect the base of hand (in this case point `[0;0;0]`) to the base of each finger. The diameter of these elements, for default 3D view, is determined by property `RobotCylinderRadius` in line 35. Additionally, model of the mechanism is set to `configex` (property `RobotModel` in line 34). This value of the property causes that the connection points are marked by additional spheres in default 3D view. All properties, with the exception of `RobotBase` property, are important only when the mechanism is shown in one of the default views. If the user defines his/her own appearance of the mechanism these properties can be omitted.

To ensure realistic behavior of modeled hand, the limits on movements of the individual fingers can be added. In the presented toolbox for its purpose property `RobotRange` may be used. This property can be set in the definition of the mechanisms or using the function `robot_set`. The table below contains proposed limits for each finger and the code below presents an example of setting the limits for the thumb.

Table 3. Limits for fingers.

finger	θ_1	θ_2	θ_3	θ_4
1	$[-\pi/2, 0]$	$[0, \pi/2]$	$[0, \pi/2]$	$[0, \pi/2]$
2	$[-\pi/9, 0]$	$[-\pi/2, 0]$	$[-\pi/2, 0]$	$[-\pi/2, 0]$
3	$[-\pi/18, \pi/18]$	$[-\pi/2, 0]$	$[-\pi/2, 0]$	$[-\pi/2, 0]$
4	$[0, \pi/9]$	$[-\pi/2, 0]$	$[-\pi/2, 0]$	$[-\pi/2, 0]$
5	$[0, \pi/5]$	$[-\pi/2, 0]$	$[-\pi/2, 0]$	$[-\pi/2, 0]$

```
robot_set(f1, 'RobotRange', [-pi/2,0; 0,pi/2; 0,pi/2; 0,pi/2]);
```

Finally, the hand plotted using the function `robot_plot` is shown in Fig. 3.

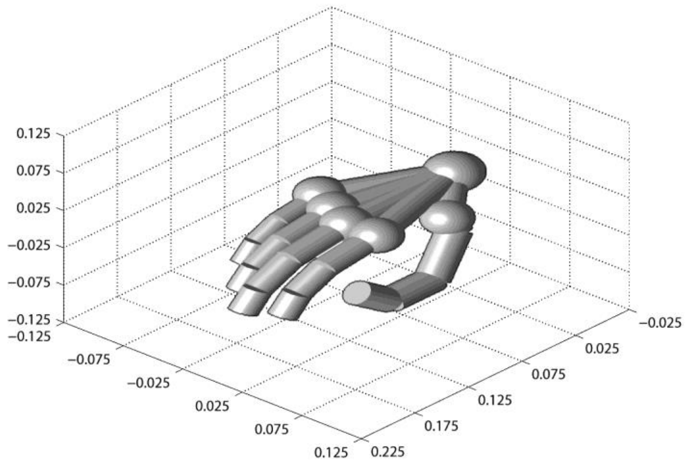


Fig. 3. Hand in default 3D view

4.3. Model of the upper limb

In order to create a model of the right upper limb, models of the arm and forearm have to be created. From the kinematic point of view, an arm and a forearm may be considered together as a 7 DOF manipulator (Li *et al*. (2011)). Fig. 4 presents the scheme of this mechanism and local coordinate systems attached to each joint.

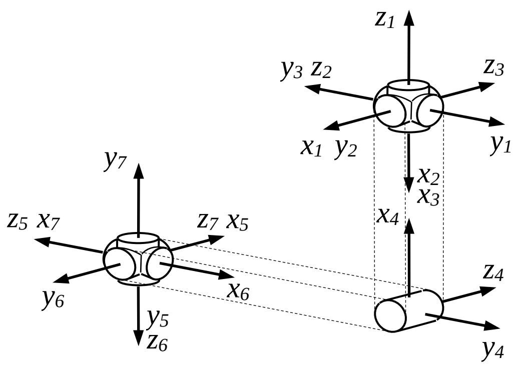


Fig. 4. Scheme of the arm and forearm.

The Denavit-Hartenberg parameters of the discussed mechanism, determined basing on the Fig 4, are shown in the Table 4.

Table 4. DH parameters of the arm and forearm.

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	$\pi/2$	0	0	θ_2
3	$\pi/2$	0	0	θ_3
4	0	l_1	0	θ_4
5	$\pi/2$	0	l_2	θ_5
6	$-\pi/2$	0	0	θ_6
7	$\pi/2$	0	0	θ_7

According to DH parameters this mechanism may be modeled as a 7R manipulator in the following way (as in above section all dimensions are given in meters):

```
1 q1 = 0;  q2 = -pi/2;  q3 = 0;  q4 = pi;  q5 = 0;  q6 = pi/2;  q7 = pi;
2
3 rlimb = robot( [dh(0,0,0,q1); dh(pi/2,0,0,q2); dh(pi/2,0,0,q3);...
4                 dh(0,0.25,0,q4); dh(pi/2,0,0.25,q5);...
5                 dh(-pi/2,0,0,q6); dh(pi/2,0,0,q7)],...
6                ['r', 'r', 'r', 'r', 'r', 'r', 'r'],...
7                'RobotCylinderRadius', 0.02);
```

The lines 3-7 define the basic part of the limb and set it in the initial configuration specified in line 1. The property `RobotCylinderRadius` (line 7) is set to match the thickness of the limb to dimensions of the hand determined in previous section. Additionally, similarly to the fingers definitions, the model of the limb should be completed by setting the movement limits:

```
robot_set(rlimb, 'RobotRange', [-pi/2,pi/2; -pi,-pi/2;...
                                -pi/4,pi; pi/2,5*pi/4; -pi/2,pi;...
                                 pi/4,3*pi/4; 3*pi/4,3*pi/2]);
```

Finally, the definition of the limb model should be completed by attaching a hand, defined in section 4.2, at the end of the limb. In this purpose the function `robot_connect`, described above, has to be used:

```
robot_connect(rlimb, rhand);
```

It is worth noticing, that function `robot_connect` does not create new objects, it completes the existing mechanism, so after calling this function the `rlimb` is the handle to complete mechanism consisting of the arm, forearm and hand. The right upper limb plotted using the function `robot_plot` is shown in Fig. 5.

Default 3D view shows the limb schematically, if more realistic view is needed, it is possible to create own 3D model of mechanism. For this purpose shape of each joint has to be determined by block objects.

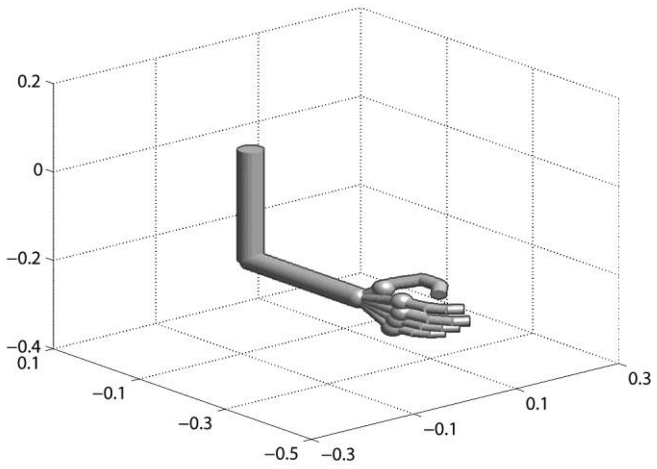


Fig. 5. Limb in default 3D view.

The fragment of the Matlab code defining the own appearance of the hand is presented below.

```
 1 b1 = block_move(cuboid(0.02,0.01,0.005),[0;0;-0.0025]);
 2 b2 = block_move(cuboid(0.03,0.02,0.02),[0.025;0;-0.01]);
 3 b3 = block_move(prism([[-0.01;-0.01] [0;-0.01] [0.01;0] [0.01;0.01]...
 4                        [-0.01;0.01]], 0.007), [0.05;0;-0.01]);
 5 b4 = block_move(prism([[-0.01;-0.01] [0;-0.01] [0.01;0] [0.01;0.01]...
 6                        [-0.01;0.01]], 0.007), [0.05;0; 0.003]);
 7 robot_part(f2,2,block_merge(b1,b2,b3,b4))
 8
 9 b1 = block_move(cuboid(0.02,0.01,0.005),[0;0;-0.0025]);
10 b2 = block_move(cuboid(0.01,0.02,0.02),[0.015;0;-0.01]);
11 b3 = block_move(prism([[-0.01;-0.01] [0;-0.01] [0.01;0] [0.01;0.01]...
12                        [-0.01;0.01]], 0.007), [0.03;0;-0.01]);
13 b4 = block_move(prism([[-0.01;-0.01] [0;-0.01] [0.01;0] [0.01;0.01]...
14                        [-0.01;0.01]], 0.007), [0.03;0; 0.003]);
15 robot_part(f2,3,block_merge(b1,b2,b3,b4));
16
17 b1=block_move(cuboid(0.02,0.01,0.005),[0;0;-0.0025]);
18 b2=block_move(cuboid(0.005,0.02,0.02),[0.0125;0;-0.01]);
19 b3=block_move(prism(arc([0.015;0.01],[0.015;-0.01],0.01),0.02),[0;0;-0.01]);
20 robot_part(f2,4,block_merge(b1,b2,b3));
21
22 pt = [[0;0.02;0.015] [0.02;0.04;0.015] etc. ];
23 fc = { [1:26; 27:52], [3 4 56 55; 7 8 60 59; etc. ], etc. };
24 robot_part(rhand, block(pt, fc));
```

The code above contains the complete definition of the forefinger and the fragment of the palm's definition. The forefinger is defined as a 4R manipulator (see section 4.1), user 3D view consists of three segments placed in the second, third and forth (grasper) local coordinate system. The lines 1-7 contain definition of the forefinger's first segment. It consists of four blocks: two cuboids and two prisms, which are connected into one block (function `block_merge`) and next attached to the second joint of the finger using the function `robot_part` (line 7). The second segment of the finger (lines 9-15) is constructed similarly, it differs only in the dimensions. The last segment (lines 17-20) contains three blocks, the first two are constructed as in the previous segments, the last one is a rounded prism and it creates the fingertip. All the segments defining the appearance of the finger can be seen in the Fig. 6a.

The lines 22-24 contain the fragment of the palm's definition. In this example the palm is defined as a complex block including 78 vertices and 37 faces, so the full definition cannot be presented. Two exemplary vertices are defined in the line 22, the definition of the first four faces is presented in the line 23. The block described by the sets of vertices and faces is created in line 24 (function `block`), and next, using the function `robot_part`, it is attached to the complex mechanism (`rhand`) defining the hand.

The appearance of the other fingers, the arm and forearm can be defined in a similar way. The complete definition of the right upper limb is available as one of the examples provided with the toolbox. The hand defined by the code presented above is shown in Fig. 6b.

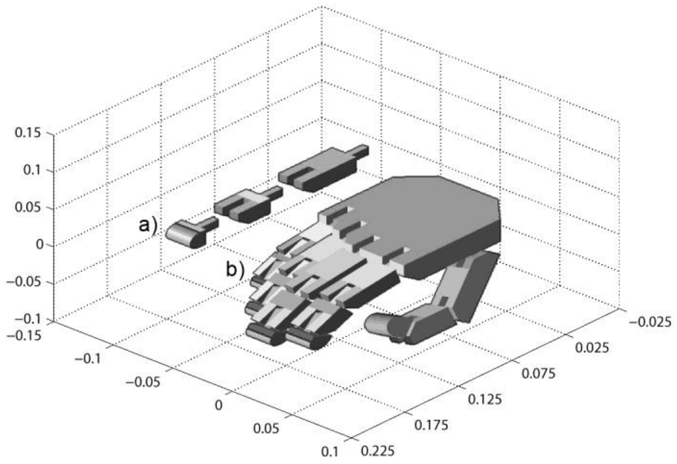
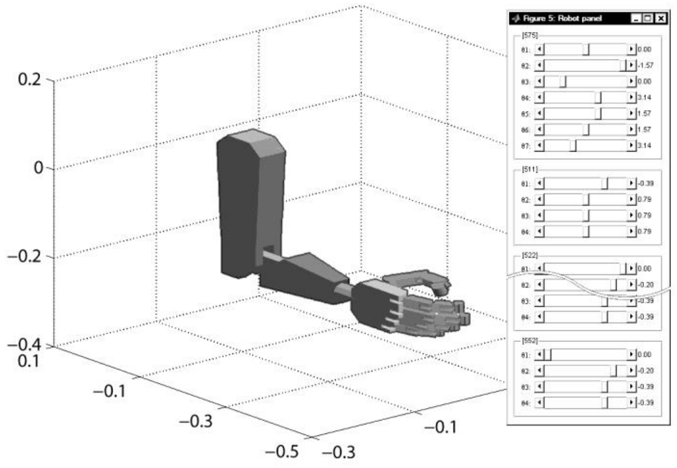


Fig. 6. a) The finger's segments, b) the hand in user 3D view.

4.4. Simulation of the limb motion

To simulate the motions of the complex mechanism as the upper limb, the functions described in section 3.3 may be used. The simplest, user-friendly method, is using a special panel (function `robot_panel`) which allows to change a configuration of the mechanism with the set of sliders. The limb is a complex mechanism consisting of six manipulators: one 7R manipulator – arm and five 4R manipulators – fingers. To describe the configuration of the whole mechanism 27 configuration variables are needed, so in this case, the panel includes 27 sliders placed in 6 groups. The first group (7 sliders) is connected to the basic part of the limb i.e. the arm and forearm, the other groups (4 sliders each) are connected to the five fingers. The upper right limb plotted in Matlab figure window in user 3D view and its panel are shown in the Fig. 7.



```
[575]
θ1: 0.00
θ2: -1.57
θ3: 0.00
θ4: 3.14
θ5: 1.57
θ6: 1.57
θ7: 3.14
[511]
θ1: -0.39
θ2: 0.79
θ3: 0.79
θ4: 0.79
[522]
θ1: 0.00
θ2: -0.20
θ3: -0.39
θ4: -0.39
[552]
θ1: 0.00
θ2: -0.20
θ3: -0.39
θ4: -0.39
```

Fig. 7. The upper right limb with panel.

In order to simulate the motion of the limb functions `robot_motion` or `animation` (section 3.3) may be used. In the example below the usage of the new function `animation` is presented.

```
1 q0 = [0;-pi;0;pi/2;pi/2;pi/2;pi; -pi/8;pi/2;0;0; repmat([0;0;0;0],4,1)];
2 qt = [0;-pi/2;0;pi;pi/2;pi/2;pi;...
3       0;pi/8;pi/8;pi/8; repmat([0;-pi/16;-pi/8;-pi/8],4,1) ];
4
5 [q, v, a, t, c] = trj_poly3([0,1.25,2,3,5], q0, qt);
6
7 f = inline('c''*[t^3; t^2; t; 1]', 'R', 't', 'flag', 'c');
8
9 animation(0:0.1:5, rlimb, f, 0.01, 0, c)
```

The limb performs the task of movement form initial configuration q0 (line 1) to final configuration qt (lines 2-3). In order to plan the trajectory the function `trj_poly3` has been used (line 5). This function implements a classical method of trajectory planning in configuration space, the resulting trajectory is described by cubic polynomials. Function `trj_poly3` has been available in the previous version of the toolbox and it has been described in the detailed manner in Pająk and Pająk (2011). The first three output arguments of this function contain configurations, velocities and accelerations for each time instant and each joint of the mechanism. The fourth argument is a vector of time, the last one is a matrix containing polynomials' coefficients describing calculated trajectory. The output argument `q` could be used directly to perform the simulation based on the function `robot_motion`, but in this example the function `animation`, which allows to perform the simulation parameterized by time, is used, so polynomials' coefficients (argument `c`) are needed. The inline function `f` (line 7) determines configuration of the limb for each time instant based on coefficients `c` calculated by `trj_poly3`. The motion of the limb in a few selected time instants is shown in Fig. 8.

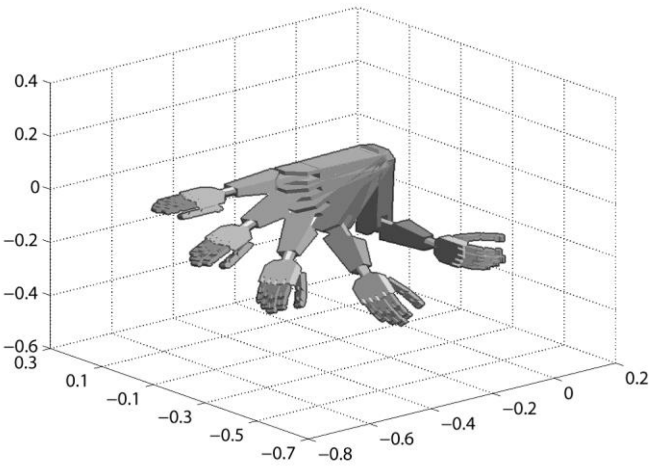


Fig. 8. The motion of the limb

5. Conclusions

In the paper, a new version of Matlab toolbox for visualization of robots and their workspaces in a realistic manner has been presented. This tool allows to model and simulate any mechanisms consisting of open kinematic chains. Any manipulator described by Denavit-Hartenberg parameters may be defined and several manipulators created in this way can be connected into one complex robot. Each mechanism may be easily shown in it default 3D view, it is also possible to create a model of a mechanism corresponding to its real appearance using a set of polyhedrons. The toolbox can be useful for educational purposes, for presentation of research results and during design and verification of production cells.

In the current version of this tool, improvement of connections between a structure representing a robot and its graphical representation has been implement. All of the objects are represented as handles to the global structures managed automatically by the program. Therefore, each modification of the object (i.e. changing the position, new configuration, changing the appearance) causes the instantaneous modification of its graphical representation. Animation capabilities of the toolbox have increased, now it is possible to perform simultaneous motions simulation of multiply robots and other objects.

The presented version of the toolbox significantly improves the modeling process of the robots in a virtual environment, but it does not include all planned functions and it is still being developed. In the next versions, collision detection between robots and other objects during their movements will be added. Additionally, a new classes of mechanisms representing mobile robots and closed kinematic chains will be introduced. These robots types, according to the authors knowledge, have not been implemented in other Matlab toolboxes. In the next stages taking into account the dynamics of modeled mechanisms is also planned.

The Robot Toolbox is freely available as the .zip archive file from its website: `http://www.uz.zgora.pl/~gpajak/rtoolbox`. After downloading the archive has to be unzipped to any folder on the computer and path to this folder must to be added to the Matlab search path. The documentation for each toolbox function is available through the Matlab function `help`, all provided examples may be run after calling function `demo_robot`.

Nomenclature

a_i	- manipulator link length of the *i*-th joint
d_i	- manipulator *i*-th joint offset
α_i	- manipulator link twist of the *i*-th joint
θ_i	- manipulator *i*-th joint angle

References

ABB (2011): *RobotStudio: of offline robot programming for ABB robots*. - http://www.abb.com.

Bingul Z., Koseeyaporn P. and Cook G.E. (2002): *Windowsbased robot simulation tools*. - 7th International Conference on Control, Automation, Robotics and Vision, Singapore.

Bruyninckx H. (2001): *Open robot control software: the OROCOS project*. - IEEE International Conference on Robotics and Automation (ICRA), pp.2523-2528.

Corke P.I. (1996): *A robotics toolbox for MATLAB*. - IEEE Robotics and Automation Magazine, vol.3, No.1, pp.24-32.

Craig J.J. (1989): *Introduction to Robotics*. - Cambridge, Massachusetts: Addison Wesley.

Fanuc (2011): *ROBOGUIDE: a family of offline robot simulation software*. - http://www.fanucrobotics.com.

Fu K.S., Gonzalez R.C. and Lee C.S.G. (1987): *Robotics. Control, Sensing, Vision and Intelligence*. – New York: McGraw-Hill.

Gourdeau R. (1997): *Object oriented programming for robotic manipulators simulation*. - IEEE Robotics and Automation Magazine, vol.4, No.3.

Li K., Chen I., Yeo S.H., Lim Ch.K. (2011): *Development of finger-motion capturing device based on optical linear encoder*. – Journal of Rehabilitation Research & Development, vol. 48, No. 1, pp. 69-82.

Kucuk S. and Bingul Z. (2010): *An off-line robot simulation toolbox*. - Computer Application in Engineering Education, 18, pp.41-52, DOI 10.1002/cae.20236.

Legnani G. (2006): *SPACELIB: a software library for the Kinematic and dynamic analysis of systems of rigid bodies*. - U. di Brescia. http://www.ing.unibs.it/~glegnani/

López-Nicolás G., Romeo A. and Guerrero J.J. (2009): *Project Based Learning of Robot Control and Programming*. - ICEE & ICEER, pp.1-8.

MathWorks, Inc. (2010): *Matlab Programming Fundamentals* – Natick, Massachusetts.

Pająk G., Pająk I. (2011): *The robot toolbox for Matlab* – Journal of Applied Mechanics and Engineering, Vol. 16, no 3, s. 809-820

Paul R.P. (1981): *Robot Manipulators: Mathematics, Programming and Control*. - Cambridge, Massachusetts: MIT Press.