

The Role of Simulation Tools in the Teaching of Robot Control and Programming

A. Romeo*

* Universidad de Zaragoza/Departamento de Informática e Ingeniería de Sistemas (DIIS),
Zaragoza, Spain (e-mail: romeo@unizar.es).

Abstract: Simulation tools can play an important role in the learning of complex subjects. This paper describes *RobotScene*: a simulation tool developed for its use in an Industrial Robotics project-based course in the context of a degree on Industrial Electronics Engineering. *RobotScene* not only allows students to design solids, robots, and scenarios, but also to program the created robot as well. Using this tool, students can improve their knowledge about some aspects of robot control design as kinematics. In addition, they can acquire robot programming skills, covering in this form several of the major learning objectives related to robot control and programming.

1. INTRODUCTION

It's well known that Industrial Robotics is a multidisciplinary subject whose in-depth knowledge involves many disciplines taught in higher education, as kinematics, dynamics, electronics and computer science. Two aspects are mainly treated in a course of Industrial Robotics for an engineering degree: The modelling of the robot for the design of its control system, and the practice to obtain programming skills using specific robot languages. Both complementary learning aspects can be benefited by simulation tools. The robot modelling has a high geometric and mathematical complexity, which is added to the difficulties (time and budget required) for students working with real robots.

For several years we have proposed a learning project-based experience (Romeo 2008) in which, students must model and solve kinematics for a commercial model of robot manipulator, design some aspects of its control system and program the robot in an industrial application. Simulation tools are essential for student's motivation and understanding of the problem, which gives good learning results. The goal of this paper is to present *RobotScene*: the simulation tool developed for this project-based course.

RobotScene is a specific software tool that provides a graphical interpretation for all the geometrical concepts involved at the design stage of an industrial robot (solids, joints, reference frames, Denavit-Hartenberg parameters, etc). Moreover, *RobotScene* provides a framework in which programming the previously created robots. It's composed by three modules specialized on *solid*, *robot* and *scene* creation respectively. There are many robotic platforms that provide simulation frameworks in which users can develop robotic applications (Micorsoft 2008, Mellado 2003), but they are not specifically oriented to robot manipulator design. Although, other platforms provide tools useful for robot kinematics and dynamics simulation, as the *Robotic Toolbox for Matlab* (Corke 1996) or *Spacelib* (Legnani 2006), but they are not elaborated graphical interfaces and don't provide robot

programming tools. Some projects as *OROCOS* (Bruyninckx 2001) and *ROBOOP* (Gourdeau 1997) allow covering this empty space, but they haven't an easy graphical interface and so, their use requires important learning efforts. Furthermore, they need an external compiler in order to perform any simulation. In addition, there are several simulation platforms as *ROBOGUIDE* (Fanuc 2008) and *RobotStudio* (Abb 2008) developed by robot manufacturers in order to provide off-line programming tools specifically designed for their robots. Nevertheless, *RobotScene* provides modules for creating solids, robots and robotic scenarios in an easy manner. It has been developed using *GLScene* (a graphic motor for *Delphi* based on *Open GL*). The programming capabilities (needed for both inverse kinematics implementation and robot programming) are provided by *PascalScript*, a *Delphi* compatible interpreter that adds Pascal-based scripting support to the *Robot constructor* module and the *Scene Constructor* one.

2. ROBOT MODELLING: KINEMATICS

The first stage of the learning project is to model and solve the kinematics of a commercial robot. To achieve this, students must follow the Denavit-Hartenberg (DH) convention (Denavit 1995), which allows obtaining forward kinematics in a systematic way.

In order to achieve this stage, students must trace three steps using two *RobotScene* modules: In first, by using the *Solid Constructor Module*, students must create all the solids that compose the robot manipulator, assigning each it a reference frame compatible with the DH rules. In a second step, they must mount the robot by using the *Robot Constructor Module*, which allows defining all data related to joints (type, range, DH parameters, home value, etc). Furthermore, this module can be used for creating robotic tools as grippers, welding equipment, etc. Once all the joints are created, students can implement the inverse kinematics equations using a specific programming interface over *PascalScript* and run it in order to validate them.

2.1 Modelling the Robot Solids

Once the students have analyzed the morphology of their assigned manipulator, identifying its joints and its solids, and assigning them all their reference frames according to the DH rules, they can begin to model their robot using *RobotScene*. The first task that must be accomplished in order to modelling a robot manipulator is defining the solids that compose it. The *Solid Creator Module* provides basic tools for modelling solids in an easy fashion. It contains a library that includes the basic geometries (prisms, cylinders, pyramids, spheres, etc) those composition makes possible the design of complex solids. Moreover, the module contains specific tools for modelling volumes generated by extrusion and revolution, taking into account the relevance of these geometries in the most of manipulators models. Fig. 1 shows an example of complex solid composed by two cylinders and a solid of revolution.

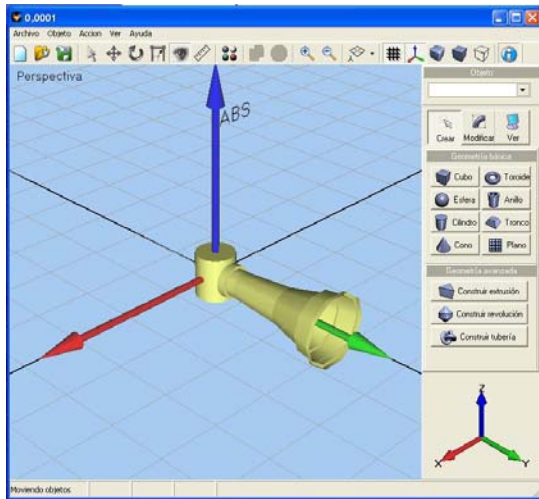


Fig. 1. Example of complex solid: two cylinders and a solid of revolution

It must be noted that even though each basic element has its associated reference frame (the local frame); robot solids must exhibit a unique one (the global reference frame), that must accomplish the DH assignment rules. For instance, the reference frame shown in Fig. 1 corresponds to this mentioned global reference frame. Note that it's about the fourth solid of an articulated robot (the first solid of its wrist) and so, its z axis corresponds to the fifth manipulator joint (the flex wrist joint), according to the DH convention.

The solid creation module includes also some tools that are useful for the solid design task:

- A tool for modifying all the geometry attributes.
- Tools for translating and rotating solids with respect to different (local or global) reference frames.
- A tool for designing the material attributes (colours, textures, etc).
- Tools for modifying the observer's location

Finally, the module enables saving the created solids to an ASCII file that contains all the solid attributes, enabling in this form its edition and modification (with or without *RobotScene*).

2.2 Assembling the Robot

Once all the manipulator solids have been created, students can assemble their robots using the *Robot Constructor Module*. For do this, students must have previously determined both the DH parameters as well as the equations that solve the inverse kinematics of their robot. According to these previous tasks, the robot creation stage exhibits two steps: the first one consists of defining in detail all the robot joints while robot is being assembling. The second step is related to the implementation of the equations that solve the robot inverse kinematics, and will be treated in the following subsection.

The complete modelling of each joint requires the definition of the following parameters: the joint type (rotational or translational), the constant DH parameters, the joint range, the home joint value, the joint max velocity and finally, the joint max acceleration. Fig. 2 shows the modelling of the fourth joint of an articulated robot. Note that, depending on the joint type, the variable DH parameter (θ_i in this case) will be represented in the table by mean of its corresponding variable identifier (q_i). As we can see in the same figure, the joint modelling is considered in the robot construction process as a part of the *solid addition procedure*. For the usual case of a six degree of freedom manipulator, this *addition procedure* must be done for seven times (from solid 0 to 6).

It's important to note at this point the role of the Denavit-Hartenberg convention in the robot modelling process with *RobotScene*: it determines not only the main part of the joint modelling, but the form in which each solid has been created as well.

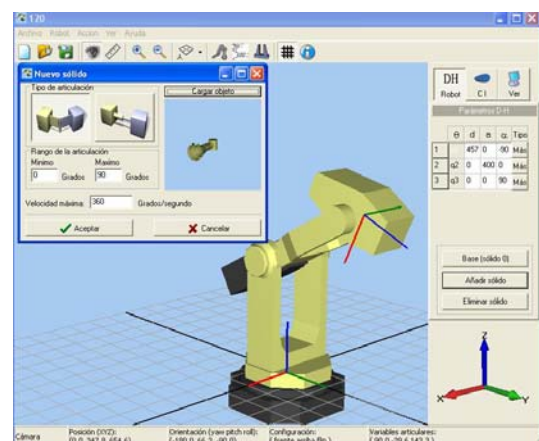


Fig. 2. Modelling the fourth joint by using the Robot Constructor Module

Once students have finished the robot assembly process, they can use a *robot guidance tool* that allows them to move the

robot by dragging the joint-associated cursors or by specifying robot destinations in joint coordinates. Note that they can't exploit the entire guidance tool potential (i.e. the guidance in user coordinates) because inverse kinematics is not yet implemented. Nevertheless, robot guidance in combination with reference frames visualization, can aid to the students to improve their understanding about the sense of DH parameters.

Likewise solids created using by the *Solid Creator Module*, robots can be saved in an ASCII file that contains all the joint attributes and the complete file path for all the solids that compose them.

2.3 Implementing the Inverse Kinematics Equations

In order to complete the robot modelling stage, students must implement the equations that solve the inverse kinematics of their robot. These equations must have been previously derived by using either geometric or algebraic approaches. For making possible the mentioned implementation, the *Robot Constructor Module* provides a specific programming tool based on *PascalScript*, that allows editing, performing syntactical checking and compiling Pascal source code. As starting point, students have a source file that includes comments with useful information about the function syntax and how to access all the needed data (user coordinates, DH parameters and robot configuration data). During this programming phase, students must pay attention to several problems inherent to inverse kinematics, as ill-conditioned equations and singularities detection and their treatment. Appendix A shows an extract of an inverse kinematics implementation, that shows the wrist singularity detection and its treatment as a predefined error.

Once inverse kinematics is implemented, students can check its correctness by using the robot guidance tool in the following way: in first, they must move the robot to a destination specified in joint coordinates. Next, they must commute to user coordinates mode in order to obtain them from the forward kinematics. Finally, they must move the robot to these user coordinates, having selected previously the adequate robot configuration. If the robot has not moved during the last step, students can assure the correctness of their inverse kinematics solution (the equations and their subsequent implementation).

The inverse kinematics implementation is saved in a specific source code file, and its existence is annotated on the associated robot file.

3. ROBOT APPLICATION PROGRAMMING

The second stage of the learning project-based experience consists in designing a robotic task by programming the previously created robot. For do it, students must firstly create a scenario that contains both the taking part application objects and the robot. Secondly, they must design and mount the appropriate robotic tool for performing the entrusted task and finally, they must program the robot by using the

provided language. The following subsections describe individually the mentioned steps.

3.1 Creating the Scenario

The scenario construction step requires the use of both the *Object* and the *Scene Constructor* modules. By using the first one, students must create all the objects belonging to the scene. This process is similar to one described in detail in the subsection 2.1. The main difference is about the object reference frame definition: for a robot solid, its reference frame should be compliant with the DH convention, whereas for non-robot objects, the choice of their reference frame depends on other factors. It also must be pointed that some objects could take active part in the task, whereas some others could be merely decorative objects.

Once objects and robots have been created by using their corresponding module, students must place them into the scene by defining the localization of their associated reference frame. For do this, the *Scene Constructor Module* allows defining these localizations in a structured form, by selecting the most adequate reference frame with respect to localizations are expressed. This means creating and maintaining a hierarchy that reflects the dependencies between the objects in the scene. Fig. 3 shows a simple structured environment consisting of a robot and a table in which there are three different objects. The corresponding scene hierarchy showed at the right side (named the *scene management tool*) of the figure shows that *Robot* and *Table* localizations have been defined with respect to the *Scenario* frame reference, whereas *Cube* and *Cil2* localizations have been defined with respect to the *Table* one. Finally, *Cil1* has been defined with respect to the *Cube* reference frame.

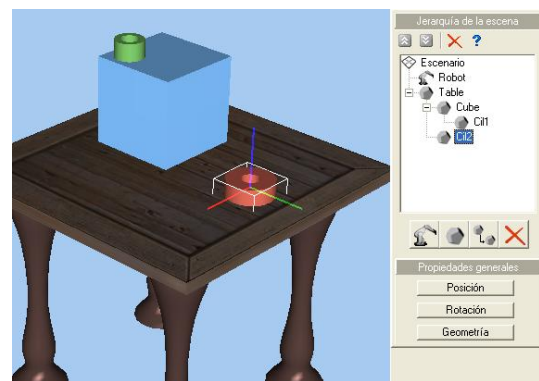


Fig. 3. Creating a simple localization hierarchy during the scene creation phase

The *scene management tool* allows the users not only to select any element (robot or object) in order to change its localization, but to modify its role in the hierarchy by dragging and dropping it as well.

3.2 Designing the Robot Tool

The robot tool is strongly related to the robotic application to perform and it can be designed by using the specific procedure provided by the *Robot Constructor Module*. Two types of tools have been considered according to the existence of mobile parts: *static tools* and *dynamic tools* (i.e. grippers). Tool design requires the modelling of its different parts as individual objects and the definition of its *tool reference frame* by specifying the corresponding transformation that will be added to kinematical chain. For the usual case of grippers, users must also define the opening range. Fig. 4 shows the tool definition window for a typical gripper consisting of one fixed base and two mobile fingers. Its corresponding reference frame has been defined by specifying its position and orientation with respect to the last robot reference frame (normally, the sixth reference frame located on the tool mounting plate).

Once the robotic tool has been completely defined, users must add it to the robot. This step can be performed with the scene construction module, after the robot has been located in the scenario. The grasping operation is internally performed by using collision detection techniques provided by *GLScene*. In this sense, only scene objects and robot tool solids are considered for the collision detection analysis.

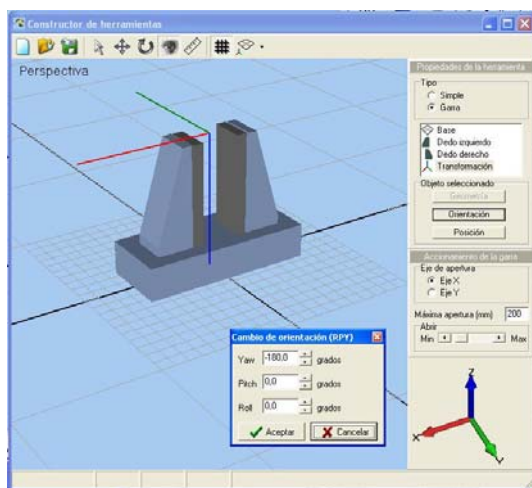


Fig. 4. Robot tool design

3.3 Robot Programming

Once the scene has been created, the last step to trace in order to design a robotic application task is programming the robot. For instance, Fig. 5 shows a complex scene in which, a SCARA robot must weld the yellow frames to the surfaces in which they are fixed.

For making it possible, the *Scene Creator Module* provides a robot programming language and a programming-oriented tool.

The language exhibits all the features of typical robot programming languages related to robot setup, robot motion, localisation data management, input/output and others.

Appendix B shows a summary of the language instructions set grouped by categories. Note that the language provides specific simulation domain instructions that allow importing the localization data present in the previously defined scene hierarchy. These instructions represent a useful connection between the scene creation phase and the robot programming one.

The *robot programming tool* consists of a programming interface in conjunction with *PascalScript*. The programming interface provides also editing, syntax checking and execution management capabilities.

Likewise the other modules, the scene and the robot program source code can be saved in their respective text files, making possible in this form their subsequent modification without *RobotScene*.

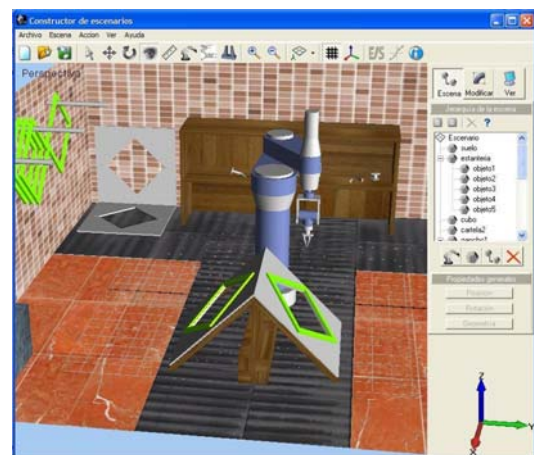


Fig. 5. Example of complex scenario that includes a SCARA robot

4. LEARNING IMPROVEMENTS

Simulation tools can be an highly useful instructional aid. In this case, two main learning aspects can be benefited from the use of the proposed simulation tool in the context of an industrial robotics related subject: the robot control and modelling in-depth comprehension, and the acquisition of robot programming skills. Lets treat each one separately.

The main learning improvements provided by the use of *RobotScene* during the robot modelling phase, are: the better comprehension of DH convention, and the in-depth understanding of the inverse kinematics problem. As mentioned before, all the robot design process, from the robot solids design to their subsequent assembly, is strongly influenced by the DH convention. *RobotScene* provides a framework in which students can check in a visual form their DH parameters, because any error will be reflected as a wrong solid assembly. They can also benefit from the frame reference viewing in conjunction with the joint guidance tool, that allow them to improve their comprehension of the DH parameters.

On the other hand, the inverse kinematics programming allows understanding its authentic complexity, because

students must take into account some different problems related to its nature and implementation, as singularities detection and treatment, arising the use of ill-conditioned equations, or the robot configuration selection enabling. At this point it must be noted that the use of the robot in the programming stage requires a carefully inverse kinematics implementation.

Finally, *RobotScene* allows acquiring programming skills. Some of them are related to general good programming practices (readability, maintainability, etc). Nevertheless, some other are specific to robot programming aspects. A good example of these skills is related to the taking advantage from the adequate structuration of the robot environment. Note that the robot environment corresponds to the scenario whose hierarchy has been defined during the scene creation stage.

5. CONCLUSIONS

RobotScene is not only a robot programming framework. It allows constructing robots and robotic scenarios, avoiding the use of proprietary software. Its use improves the robot kinematics comprehension, and allows the students to acquire robot programming skills. In consequence, it can be a useful instructional aid in the teaching of Industrial Robotics.

REFERENCES

- Abb (2008). RobotStudio: of offline robot programming for ABB robots, <http://www.abb.com/>.
- 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.
- Denavit, J. And R. S. Hartenberg (1995). Kinematic notation for lower-pair mechanisms based on matrices *Transactions of the ASME, Journal of Applied Mechanics*, Vol. 23, pp. 215–221.
- Fanuc (2008). ROBOGUIDE: a family of offline robot simulation software, <http://www.fanucrobotics.com/>.
- Gourdeau, R. (1997). Object oriented programming for robotic manipulators simulation. *IEEE Robotics and Automation Magazine*, Vol.4, No.3.
- Legnani, G. (2006). SPACELIB: a software library for the Kinematic and dynamic analysis of systems of rigid bodies. U. di Brescia. <http://bsing.ing.unibs.it/~glegnani/>.
- Microsoft (2008). Microsoft Robotics Developer Studio, <msdn.microsoft.com/robotics/>.
- Mellado, M. C. Correcher, J.V. Catret, and D. Puig (2003). VirtualRobot: an open general-purpose simulation tool for robotics *The European Simulation and Modelling Conference (ESM2003)*, EUROSIS, Naples, Italy., pp. 271–350.
- Nethery, J., and M.W. Spong (1994). Robotica: a Mathematica package for robot analysis. *IEEE Robotics and Automation Magazine*, Vol. 1, No. 1, pp. 13–20
- Romeo, A. (2008) Teaching aid for the subject Industrial Robotics Anillo Digital Docente (ADD). Universidad de

Zaragoza

<http://moodle.unizar.es/course/view.php?id=552>.

Appendix A. INVERSE KINEMATICS EXAMPLE

```
//
// INVERSE KINEMATICS
//
// Returns the vector of joint coordinates that allows the tool
// reaching the desired localization. You have access to the
// Denavit-Hartenberg parameters in the following way:
// d,a,alfa,tita: array[1..num_joints] of single.
// In addition, the variable "configuration" will contain the
// previously selected configuration.
//
// DO NOT MODIFY THE FUNCTION HEADER

function inverse_kinematics(T:TTTransformacion):TArray;
var
  q:TArray;
// for friendly notation purposes
  px,py,pz,mx,my,mz,nz,nx,ny,oz,ox,oy,ax,ay,az : single;
  r13,r23,r31,r32,r33,fi,aa,bb,c,r:single; // auxiliary variables

begin
  px:= T.pos.x;
  py:= T.pos.y;
  pz:= T.pos.z;
  nx:= T.n.x;
  ny:= T.n.y;
  nz:= T.n.z;
  ox:= T.o.x;
  oy:= T.o.y;
  oz:= T.o.z;
  ax:= T.a.x;
  ay:= T.a.y;
  az:= T.a.z;
// the wrist position when robot tool is a symmetric gripper
  mx:= px - ax*d[6];
  my:= py - ay*d[6];
  mz:= pz - az*d[6];

  // First joint
  if configuration[1]='front' then
    q[1]:= Atan2(my,mx);
  else
    q[1]:= Atan2(-my,-mx);

  // Second joint
  if abs(q[1])-pi/2<0.1 then
    begin
      c:=d[4]*d[4]-a[1]*a[1]-a[2]*a[2]+2*my/Sin(q[1])*a[1]-
      mz*mz-my*my/(Sin(q[1])*Sin(q[1]));
      aa:=2*a[1]*a[2]-2*my/Sin(q[1])*a[2];
    end
  else
    begin
      c:=d[4]*d[4]-a[1]*a[1]-a[2]*a[2]+2*mx/Cos(q[1])*a[1]-
      mz*mz-mx*mx/(Cos(q[1])*Cos(q[1]));
      aa:=2*a[1]*a[2]-2*mx/Cos(q[1])*a[2];
    end;
end;
```

```

bb:=-2*mz*a[2];
r:=Sqrt(aa*aa+bb*bb);
fi:=Atan2(bb,aa);

if r/(2*a[2])>=(a[2]+d[4]) then
begin
    Kinematics_Error:=Too_Far;
    Exit;
end;

if configuration[2]='below'then
    q[2]:=fi+Acos(c/r)
else
    q[2]:=-fi-Acos(c/r)

// Third joint
if abs(q[1])-pi/2<0.1 then
    q[3]:=Atan2((my/Sin(q[1])-a[1]-a[2]*Cos(q[2])),-(mz-
a[2]*Sin(q[2]))) -q[2]
else
    q[3]:=Atan2((mx/Cos(q[1])-a[1]-a[2]*Cos(q[2])),-(mz-
a[2]*Sin(q[2]))) -q[2] ;

// Wrist joints
r13:=
ax*Cos(q[1])*Cos(q[2]+q[3])+ay*Sin(q[1])*Cos(q[2]+q[3])
+az*Sin(q[2]+q[3]);
r23:= ax*Sin(q[1])-ay*Cos(q[1]);
r33:=
ax*Cos(q[1])*Sin(q[2]+q[3])+ay*Sin(q[1])*Sin(q[2]+q[3])-
az*Cos(q[2]+q[3]);
r31:=
nx*Cos(q[1])*Sin(q[2]+q[3])+ny*Sin(q[1])*Sin(q[2]+q[3])-
nz*Cos(q[2]+q[3]);
r32:=
ox*Cos(q[1])*Sin(q[2]+q[3])+oy*Sin(q[1])*Sin(q[2]+q[3])-
oz*Cos(q[2]+q[3]);

if configuration[3] = 'Flip' then
    q[5]:= Acos(r33)
else
    q[5]:=-Acos(r33);

// Wrist singularity
if abs(q[5])<0.001 then
begin
    Kinematics_Error:=Wrist_Singularity;
    Exit;
end;

q[4]:=Atan2(r23/Sin(q[5]),r13/Sin(q[5]));
q[6]:=Atan2(r32/Sin(q[5]),-r31/Sin(q[5]));

// Radians to degrees
for i:=1 to 6 do q[i]:=q[i]*180/pi;
result:=q;
end;

```

Appendix B. ROBOT PROGRAMMING LANGUAGE SUMMARY

Basic operations with localization data

```

function NullTr:TTransform;

function Comp(T1,T2:TTransform):TTransform;

function Invert(T:TTransform):TTransform;

function Transf(x,y,z,yaw,pitch,roll:single):TTransform;

function RotYPR(yaw,pitch,roll:single):TTransform;

function RotEuler(alfa,beta,gamma:single):TTransform;

function Traslation(x,y,z:single):TTransform;

function Shift(T:TTransform; x,y,z:single):TTransform;

function ShiftRel(T:TTransform; x,y,z:single):TTransform;

procedure Speed(value:single);

function Where_Tool:TTransform;

function Joint_Values:TArray;

function
Import_Relative_Transform(name:string):TTransform;

function
Import_Absolute_Transform(name:string):TTransform;

```

Motion instructions

```

procedure Home;

procedure MovJoint(destination:Tarray);

procedure MovCoor(destination:TTransform);

procedure MovLin(destination:TTransform);

procedure MovRelCoor(destination:TTransform);

procedure MovRelLin(destination:TTransform);

procedure Drive(Num_Joint:integer;value:single);

procedure Stop(Miliseconds:integer);

```

Miscellaneous instructions (setup, I/O, etc)

```

procedure Open;

procedure Close(value:single);

procedure Tool_Transform(T:TTransform);

procedure Set_configuration(conf:string);

procedure Speed(value:single);

procedure Wait(i:integer);

procedure Signal(i:integer);

```