PROJECT 2: DEBURRING WITH THE FANUC M16IB

Marie-Ange Janvier **Stefan Bracher**

ABSTRACT

The deburring of a four edge spherical bloc is simulated for a Fanuc M16iB robot. To achieve this goal, a trajectory file containing the joint angles of the robot has been computed offline using the resolved-motion rate algorithm. This algorithm solves the inverse kinematics in a efficient way, as it takes advantage of previous points and uses a numerically computed Jacobian. The results have been validated in an animation showing the tool moving around the sphere four times in two different resolutions.

INTRODUCTION

This work is a follow up of project one "Postures for a Fanuc [1] M16iB manipulator" [2]. In project one we calculated the direct and inverse kinematics to position the robot to one specific edge point of a spherical block.
Starting from the same point, we will let the robot move around the spherical block along its edge in order to deburr it.

In this work we use a different method to calculate the inverse kinematics, as we now have to compute a whole path and not only a single point. This method uses the already know joint-angles of the previous point to calculate the next ones on the path. The method is know as resolved-motion rate projection and is included in our path control algorithm.

First we will present the Jacobian, the resolved motion rate algorithm and how the path was developed for this task.

THE JACOBIAN

The Jacobian J relates the joint angle velocities to the end effector velocities.

$$t = J\dot{\theta} \tag{1}$$

With t and $\dot{\theta}$ defined as

$$t \equiv \left[\omega^T \dot{p}^T\right]^T \in 2\times R^3,\ \dot{\theta} \equiv \left[\dot{\theta}_1 \ldots \dot{\theta}_n\right]^T \in R^n,$$

Because we are not interested in the symbolical solution of the Jacobian, we use a numerical approach to compute it for each point:

$$J = \left[\begin{array}{c|c|c|c} & & & \\ e_1 & e_2 & \ldots & e_n \\ & & & \\ e_1\times r_1 & e_2\times r_2 & \ldots & e_n\times r_n \end{array}\right] \tag{2}$$

With e_n being the z axes of the n-th joint expressed in base coordinate system of the robot and r_n the vector from the robot base to point n.

In detail this has to be done like the following:

$$\begin{aligned} Q_{11} &= Q_1 \\ Q_{12} &= Q_{11}\cdot Q_2 \\ &\ldots \\ Q_{1n} &= Q_{1(n-1)}\cdot Q_n \end{aligned} \tag{2.1}$$

Where Q is the orientation matrix obtained from the David Hartenberg Parameters, as used in [1].

Afterwards we compute :

$$\begin{aligned} e_1 &= k \\ e_2 &= Q_{11}k \\ &\ldots \\ e_n &= Q_{1(n-1)}k \end{aligned}$$

with k=[0 0 1]^T (2.2)

Then, all r_n are calculated with respect to the previous referential:

$$\begin{aligned} r_6 &= a_6 \\ r_5 &= a_5 + Q_{15}\cdot r_6 \\ &\ldots \\ r_1 &= a_1 + Q_{11}\cdot r_2 \end{aligned} \tag{2.3}$$

And each of the r_n is brought back to the robot base referential by multiplication of the corresponding orientation matrices

$$r_n = Q_{1(n-1)} \cdot r_n \tag{2.4}$$

Where a_i are the translations of the coordinate transformation matrices of the systems attached to the robot members, provided by the David-Hartenberg parameters.

RESOVLED-MOTION RATE ALGORITHM

The resolved-motion rate algorithm allows to compute the joint angles of a new point close to a previous point, based on its joint angles. This algorithm is referred as the twist decomposition algorithm in [5].

This algorithm includes the orthogonal decomposition of twists as well as a gradient of an objective function (h) that allows to prevent the angles from reaching the joint limits.

Orthogonal decomposition of Twists

The twist projector T is used for the orthogonal decomposition. This is necessary in order to get good solutions for functionally redundant manipulators.
Functionally redundant manipulators are manipulators with dimensions of the operational space greater than the task space. [5].

As our task is fully 3D, T was chosen according to the projector vectors provided by [5] as:

$$T=\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3}$$

Objective function

To avoid the joint limits, the gradient of an objective function (h) is used.

$$h=-W*(t'-tmean) \tag{4}$$

with $W = diag(\theta_{\max} - \theta_{\min})$

(4.1)

and $tmean = 1/2 \cdot (\theta_{\max} + \theta_{\min})$

(4.2)

The reach function

Finally a reach function performing the algorithm was implemented the following way:

1. The inputs of the function are: The desired location (pf) and orientation of the end effector (Qf), as well as the joint angles (t) of the last previous know position.
2. Based on these joint angles (t), the old points position (p) and Jacobian (J) are calculated with Eq(2) and the direct kinematics developed in [2].
3. Now actual position and orientation error are computed

   $$dp=a*(pf'-p) \tag{5}$$

   $$dq=a*Q*vect(Q'*Qf) \tag{6}$$

   Giving the error twist
   $$dx=[dq;dp] \tag{7}$$
4. The functional redundancy equation gives us new thetas leading the end effector closer to the desired position and orientation.

   $$t=t+dt' \tag{8}$$

   With
   $$dt=(pinv(J)*T)*dx + pinv(J)*(I-T)*J*h \tag{9}$$

   Using h and T as defined in Eq.(4) and (3)

5. The error between the desired and current parameters, given by the new obtained thetas is evaluated.
   If this error satisfies the minimal allowed error criteria, the thetas are returned. If not, steps 2 to 5 are repeated until the criteria are met or the maximum iteration allowed is reached.

PATH DEVELOPMENT

It is necessary to compute a trajectory file for the robot. The trajectory file contains the joint angles corresponding to the points on the path described in the introduction.

To get the x and y coordinates of the points on the path we use the projection of the edge of the sphere in the x-y plane, which is simply a square. We go in steps around this square (see Fig. 1) and find the corresponding z coordinate for each point with the sphere equation. (See Appendix A, functions "path" and "nextpoint")

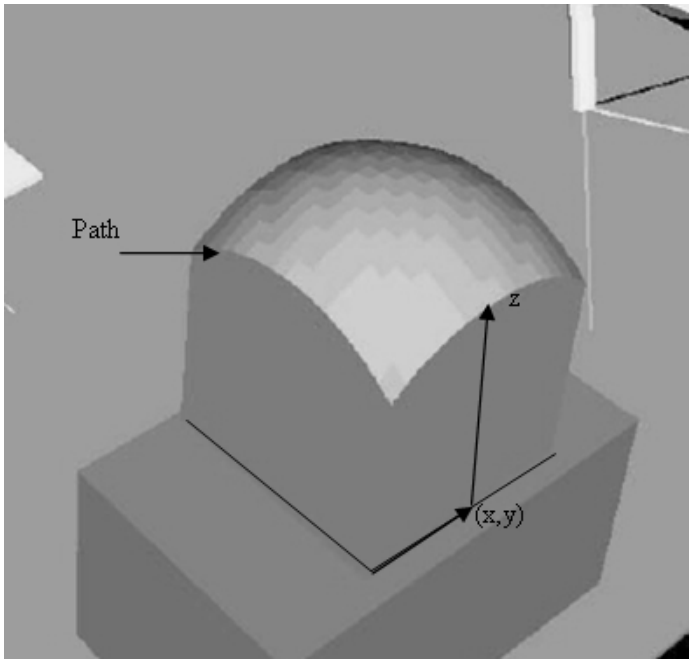

Figure 1. The edge point coordinates is found according to the x and y coordinates of the square.

Once the coordinates of a point are obtained, the correlating orientation matrix Qf of the tool is computed, so that the tool will be perpendicular to the sphere surface. This is done with simple vector algebra:

The vector z is the middle of the sphere minus the point coordinate, divided by the sphere radius:

$$vz=(n-pf)/0.25 \tag{10}$$

x and y axes are found with the crossproducts:

$$\begin{aligned} vx&=\text{cross}(xo, vz)\\ vx&=-vx/\text{norm}(vx)\end{aligned} \tag{11}$$

$$\begin{aligned} vy&=\text{cross}(vz,vx)\\ vy&=vy/\text{norm}(vy)\end{aligned} \tag{12}$$

Using an "arbitrary" direction xo=[0 -1 0]. Arbitrary in parentheses as this determines the orientation of the tool around its own axis. Several different orientations have been tried, but this one proved best practical results. (No reach errors, no joint limit reaching, no penetration of any physical bodies).

Giving Qf=[vx', vy', vz'] (13)

These values serve as input to the reach function, which provides the corresponding joint angles that are written in the trajectory file. However to use this method the inverse kinematics has to be solved for one point on the path. This can be done with the method described in [2] or guessing the angles and using the resolved motion rate algorithm to find the exact values.

RESULTS

The resulting .trj file can be found in ANNEX B. It can be used with [3] to simulate the deburring process. The animation shows the tool moving around the block four times. Figure 2 shows the solutions of all the joint angles with a resolution of 400 points per turn, while Figure 3 shows the same solutions for a 4000 points per turn.

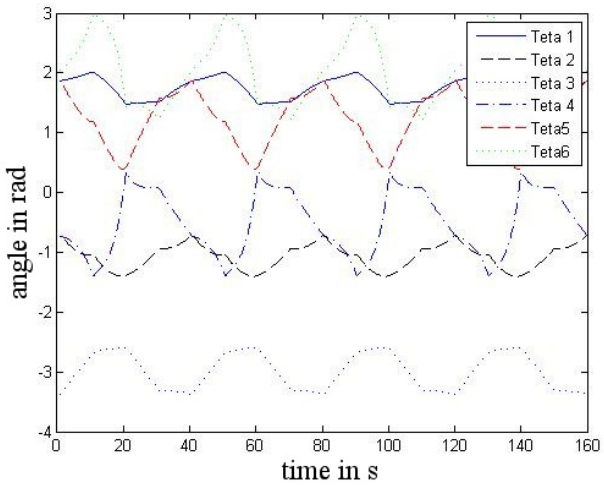

Figure 2. The joint angles with respect to the time of the path is presented for 4 turns with each 400 points. The step time represents 0.04s.

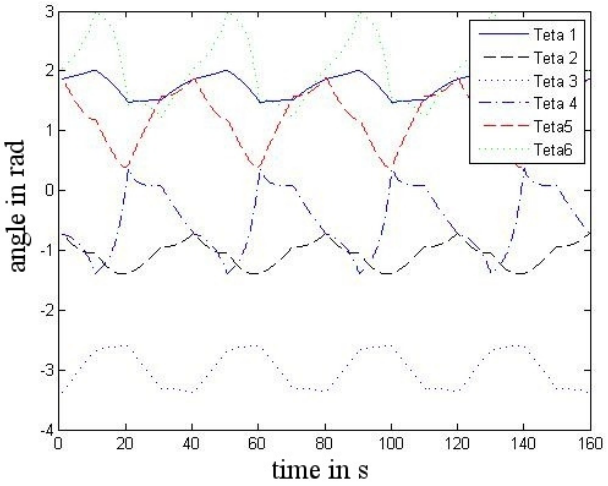

Figure 3. The joint angles with respect to the time of the path is presented for 4 turns with each 4000 points. The step time represents 0.04s.

Unfortunately we cannot show the full animation in this paper. Therefore only a few snapshots are presented in the following figures.

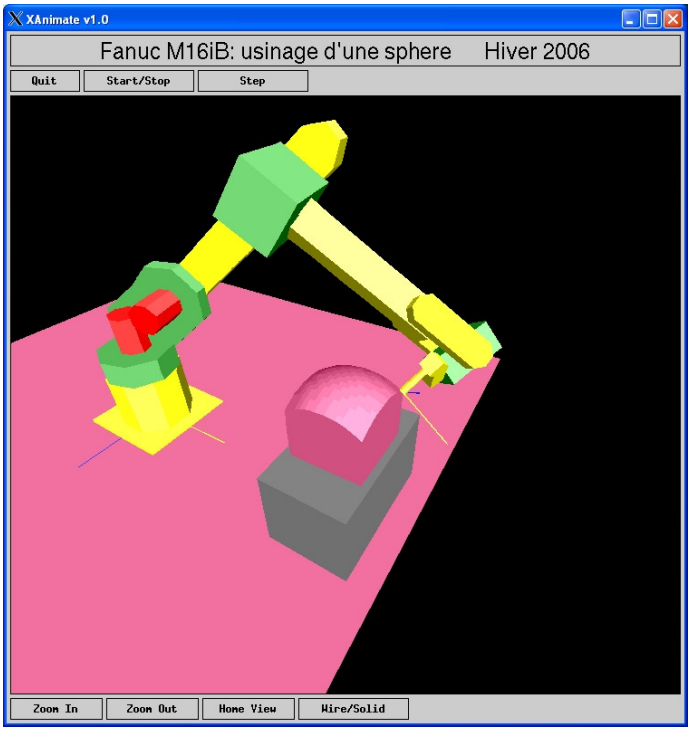

Figure 4. Initial position.

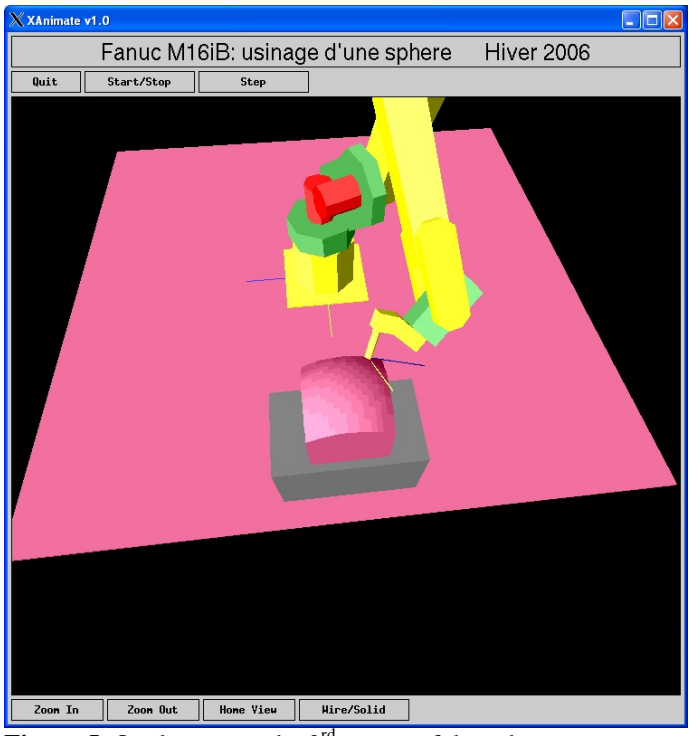

Figure 5. On the way to the 3rd corner of the sphere.

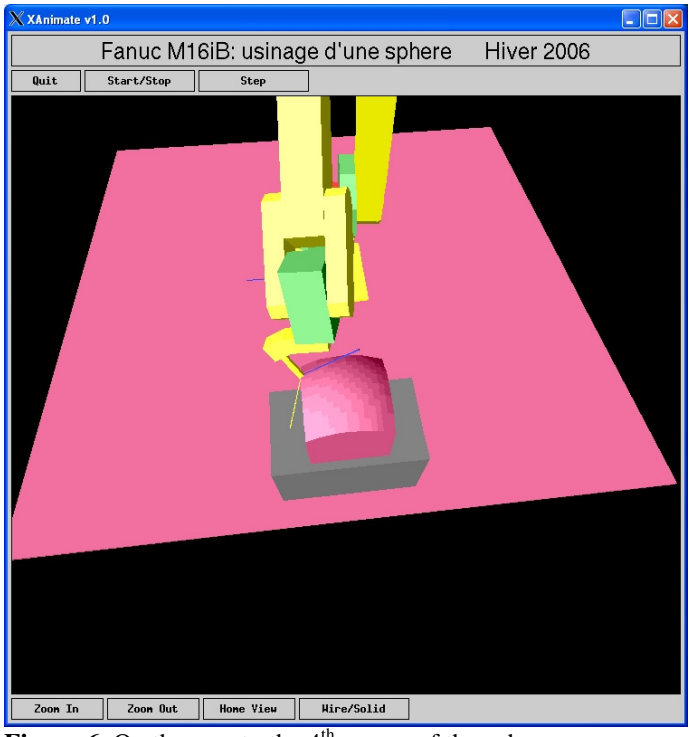

Figure 6. On the way to the 4th corner of the sphere**.**

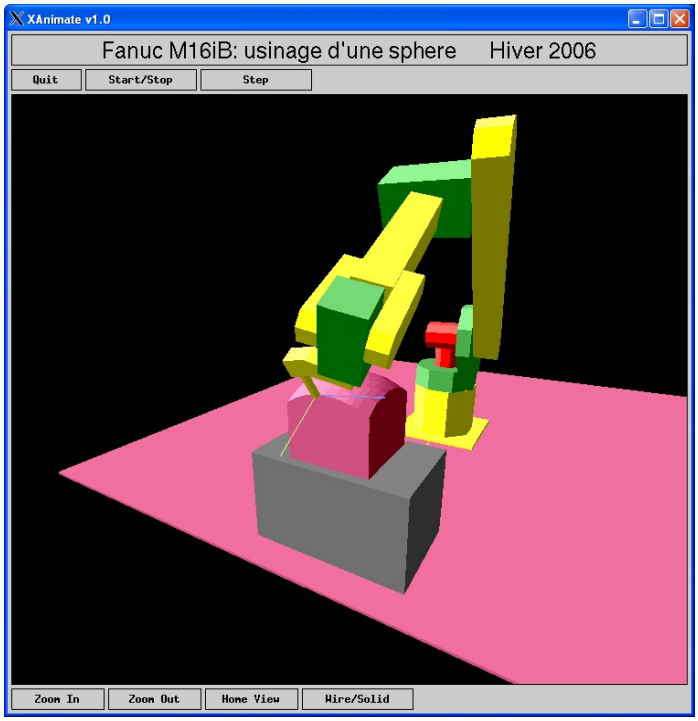

Figure 7. Back to the initial point.

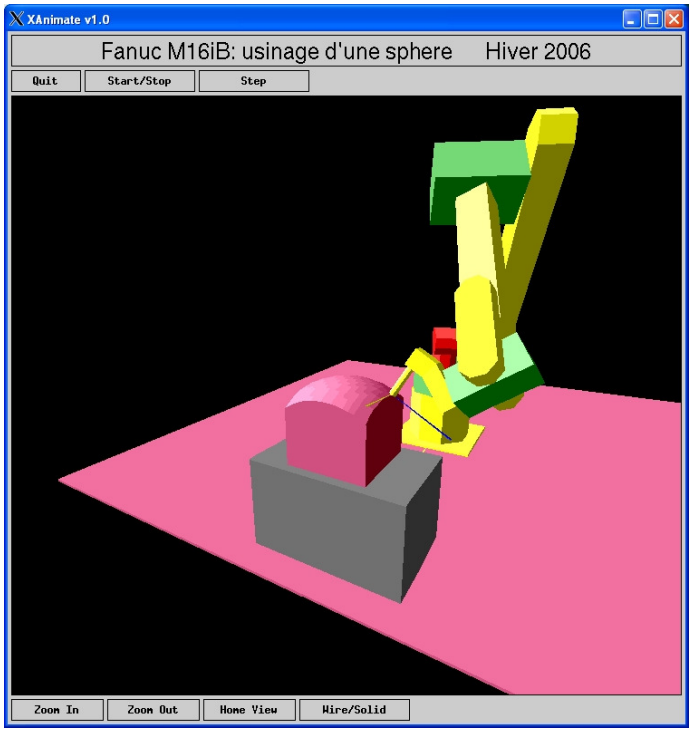

Figure 8. Second turn of the robot.

As you can see, the tool is always normal to the surface and the orientation around its own axis remains always the same, as forced by Eq.(10), (11) and (12).

DISCUSSION AND CONCLUSION

The path algorithm to deburr the spherical block edge was successful. We were able to move the robot tool along the path four times. The reach function proves to be position error stable, as it was able to solve the task as well for 4000 points per turn resolution as well as for 400 points per turn resolution. Even 90 degree angular shift around the tool axis could be forced in one step. (But this was is not used in the current path).

On the other hand, the initial definition of the tool axis orientation is very critical. Some orientations led to non solvable configurations. Careful testing is thus mandatory before using a full path on a real robot. Runtime calculation of the path in a real robot seems very unwise without other safety measures. For pre-recorded (and tested) paths, this is however a good solution.

The numerical Jacobian matrix was found quite useful in the computation of the joint displacement compared to a symbolic solution. It was found to be much faster and still provided enough accuracy for the task.

As for the resolved-motion rate projected, it can be doubted in this task that it was necessary since the orientation of the tool was forced in each point (Qf is always known).

Finally, the robot is able to complete a path with the resolved-motion projection algorithm at a good point accuracy.

REFERENCES

[1] FANUC Robotics America Inc. www.fanucrobotics.com
[2] Bracher, S., Janvier, M.A., École Polytechnique de Montréal Postures for a Fanuc M16iB manipulator
[3] Luc Baron, École Polytechnique de Montréal, Simulation Software of FANUC robot
[4] Luc Baron, École Polytechnique de Montréal, MEC6503 Task Project2 and class notes
[5] Baron L., Huo L., 'Kinematic Inversion of Functionally-Redondant Serial Manipulators : Application to Arc-Welding'.

ANNEX A

MATLAB CODE

Main.m

```
%TP2 MAIN FILE
%
%Author: Marie-Ange Janvier and Stefan Bracher
%Date:   March 10th, 2006
%


%---------------simulation parameters------------------
points=100;    %number of intermediate steps
dt=0.04;       %robot steptime
turns=4;       %number of turns around the block
%------------------------------------------------------


% lets first determine a first point on the block
zpt= 0.3 + sqrt(0.25^2 -(-0.125 )^2 -(1.125 - 1)^2 );
%The z coordinate of a corner point
corner_sphere=[-0.125 1.125 zpt ];      % Corner point 1
r=0.25; xn= 0; yn= 1;zn= 0.3;       % The Sphere parameters
n=[xn yn zn];                       % The center of the sphere



%Cornerpoints and directions
pt1=[corner_sphere];        %Corner 1
direction1=[0 -1 0];        %To go from corner 1 to corner 2
pt2=pt1+0.25*direction1;    %Corner 2
direction2=[1 0 0];         %To go from corner 2 to corner 3
pt3=pt2+0.25*direction2;    %Corner 3
direction3=[0 1 0];         %To go from corner 3 to corner 4
pt4=pt3+0.25*direction3;    %Corner 3
direction4=[-1 0 0];        %To go from corner 4 to corner 1

% Creating void matrices to fill data in
trj=[]; tot_pts=[];path_final=[]; real_pts=[];



 for nt=1:turns             % Loop to go around several times

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%_____________________________________________________________
%----------------------for pt1 to pt2---------------------
disp('Trying to get from point 1 to 2 now');

direction=direction1;       % Direction to move on ground
nextdirection=-direction3;  % Axis Orientation of the tool
pf=pt1;                     % First point on track

t=[1.8639 -0.70875 -3.381 -0.73032 1.8847 2.0493]; %The initial joint
%angles from TP1


step=(pt1(2)-pt2(2))/points;    % Step length on ground

[trj1, label_all1]=path(points, step, pf, direction, nextdirection, n, t);
%Calculated the joint angles

trj1=[trj1,label_all1'];       %Storing it
% %----------------------saving for pt1 to pt2----------------------
%
 enregistre_trj(trj1(:,1:6), dt, 'simpt1_pt2.trj');% Save the partial track
%
%_____________________________________________________________
%----------------------for pt2 to pt3---------------------
disp('Trying to get from point 2 to 3 now');
direction=direction2;       % Direction to move on ground
```

```
nextdirection=-direction3;  % Axis Orientation of the tool
pf=pt2;                     % First point on track
t=trj1(end,1:6);            %The  initial joint angles
%
step=(pt3(1)-pt2(1))/points;  % Step length on ground
%
[trj2, label_all2]=path(points, step, pf, direction, nextdirection, n, t);
%Calculates the joint angles

trj2=[trj2,label_all2'];     %Storing it
% %----------------------saving for pt2 to pt3---------------------
% % works well with 1000 points
 enregistre_trj(trj2(:,1:6), dt, 'simpt2_pt3.trj');% Save the partial track
%
%_______________________for pt3 to pt4_____________________
% %----------------------for pt3 to pt4---------------------
disp('Trying to get from point 3 to 4 now');
direction=direction3;       % Direction to move on ground
nextdirection=-direction3;  % Axis Orientation of the tool
pf=pt3;                     % First point on track
t=trj2(end,1:6);             %The  initial joint angles
%
step=(pt4(2)-pt3(2))/points;  % Step length on ground

%
[trj3, label_all3]=path(points, step, pf, direction, nextdirection, n, t);
%Calculated the joint angles

trj3=[trj3,label_all3'];  %Storing it
% %----------------------saving for pt3 to pt4---------------------
% % works well with
 enregistre_trj(trj3(:,1:6), dt, 'simpt3_pt4.trj'); % Save the partial track

%_____________________________________________________________

% %----------------------for pt4 to pt1---------------------
disp('Trying to get from point 4 to 1 now');
direction=direction4;       % Direction to move on ground
nextdirection=-direction3;  % Axis Orientation of the tool
pf=pt4;                     % First point on track
t=trj3(end,1:6);            %The  initial joint angles
%
step=(pt4(1)-pt1(1))/points;  % Step length on ground

%
[trj4, label_all4]=path(points, step, pf, direction, nextdirection, n, t);
%Calculated the joint angles

trj4=[trj4,label_all4'];  %Storing it
% %----------------------saving for pt3 to pt4---------------------
% % works well with
 enregistre_trj(trj4(:,1:6), dt, 'simpt4_pt1.trj'); % Save the partial track
%_______________________for pt3 to pt4_____________________
% %----------------------for pt3 to pt4---------------------

%Merging the partial paths to one big file
if nt==1
 trj_tot1(:,1:6)=[trj1(:,1:6);trj2(:,1:6); trj3(:,1:6); trj4(:,1:6)];
 disp('turn 1 done');
elseif nt==2
 trj_tot2(:,1:6)=[trj1(:,1:6);trj2(:,1:6); trj3(:,1:6); trj4(:,1:6)];
 disp('turn 2 done');
elseif nt==3
 trj_tot3(:,1:6)=[trj1(:,1:6);trj2(:,1:6); trj3(:,1:6); trj4(:,1:6)];
 disp('turn 3 done');
elseif nt==4
 trj_tot4(:,1:6)=[trj1(:,1:6);trj2(:,1:6); trj3(:,1:6); trj4(:,1:6)];
 disp('turn 4 done');
 end;

 end;

 path_final(:,1:6)=[trj_tot1(:,1:6);trj_tot2(:,1:6);trj_tot3(:,1:6);trj_tot4(:,1:6)];


 % Figure of joint position with respect to time for projected approach
```

```
%  path_final=[trj1(:,1:6); trj2(:,1:6); trj3(:,1:6); trj4(:,1:6)];

enregistre_trj(path_final, dt, 'full_path4_100.trj');
x=linspace(1,160, points*4*4);
figure
plot(x', path_final(:,1)); hold on;
plot(x', path_final(:,2),'k--');
plot(x', path_final(:,3),':');
plot(x', path_final(:,4),'-.');
plot(x', path_final(:,5),'r-- ');
plot(x', path_final(:,6),'g:'); hold off;
legend('Teta 1','Teta 2','Teta 3','Teta 4', 'Teta5', 'Teta6' );

% comparaison with real pts and trajectory pts of the robot
x=linspace(1,40,1000);
```

path.m

```
function [trj, label_all]=path(points, step, pf, direction, nextdirection, n, t)

%for number of int. point do
for ii=1:(points)

%nextpoint
if ii~=1 %We dont want to do this for the first point we already know
    pf=nextpoint(step,pf,direction); %find the new coordinates
end

% Calculation of the new Orientation Qf
vz=(n-pf)/0.25;
xo=nextdirection;
vx=cross(xo, vz);
vx=-vx/norm(vx);
vy=cross(vz,vx); vy=vy/norm(vy);
Qf=[vx', vy', vz'];



label=1;    %A variable so we can see if there is a reach error, per default 1

if ii~=1    %We dont want to do this for the first point we already know
[t, label]=reach(pf,Qf,t); %Calculates the joint angles

end

% In case angles are multiples of pi we set it back
 for ti=1:6
     while t(ti)>(2*pi)
         t(ti)=t(ti)-(2*pi);
     end
   while t(ti)<(-2*pi)
         t(ti)=t(ti)+(2*pi);
     end
 end

trj(ii,:)=t;            % writing the new angles to trj
label_all(ii)=label;    %Writing the labels
end
```

reach.m

```
function [t, label]=reach(pf,Qf, t)

n=25;       %nombre d`iterations
a= 0.75;    %amortisseur
ep=1;       %erreur positoin
ee=10;      %erreur d'orientation
```

```
[p,Q,J]=mgd_tot(t); %actual position, orientation and Jacobienne
e=Q(1:3, 3);        %Unit vector along the line
P=eye(3)-e*e'; L =e*e'; % Plane and Line projectors


T=zeros(6,6);
%Options for the twist projector T
% option 1 %  2 orientation, 3 positions
% T(1:3,1:3)=P; T(4:6,4:6)=eye(3);
% option 2 %  3 orientation, 3 positions
T(1:3,1:3)=eye(3); T(4:6,4:6)=eye(3);  % We are using this one
% % option 3 %  3 orientation, 2 positions
%T(1:3,1:3)=eye(3); T(4:6,4:6)=P;
% % option 4 %  2 orientation, 2 positions
% T(1:3,1:3)=P; T(4:6,4:6)=P;

tmax=[2.9671, 2.1817, 4.0143 ,3.4907, 2.4435, 7.8540]'; % max joint angles
%of the robot
tmin=-tmax; % minimal angles
tmean=1/2*(tmax +tmin); % medium angles
W=diag(tmax-tmin);      % Weightning matrix
h=-W*(t'-tmean);        % arbitrary vector

while(n>0 & ep>0.0001 & ee> 0.001)
% Repeat until solution found or task abandoned

    dp=a*(pf'-p)  ;      %Position error
    dq=a*Q*vect(Q'*Qf);  %Orientation error
    dx=[dq;dp];          %Error twist
    dt=(pinv(J)*T)*dx + pinv(J)*(eye(6)-T)*J*h; % change in angles
    t=t+dt';             %Next angle
    [p,Q,J]=mgd_tot(t); %actual position
    ep=norm(pf'-p);     %Position error real
    ee=asin(norm(Q*vect(Q'*Qf)));   %Orientation error real
    n=n-1;               % Decreasing n

end
label=1;                % No reach error -> label=1

if n==0
        disp('reach error');    % In case there was a reach error
        label=0;                % Label the error
end
```

nextpoint.m

```
function nxpoint=nextpoint(step, lastpoint, direction)

r=0.25; xn= 0; yn= 1;zn= 0.3; % center of sphere
nxpointongd=lastpoint+step*direction; % next point in the gound path
zpt=zn + sqrt(r^2  - (nxpointongd(1) - xn)^2  - (nxpointongd(2) - yn)^2 );
% corresponding z value
nxpoint=[nxpointongd(1) nxpointongd(2) zpt];  % new point on sphere
```

mgd_tot.m

```
function  [p,Q,J]=mgd_tot(t)
% Direct kinematics
teta=t;


% D-H parameters
lu=[cos(-pi/2),cos(0), cos(pi/2), cos(-pi/2), cos(pi/2),cos(0) ];
u=[sin(-pi/2),sin(0), sin(pi/2), sin(-pi/2), sin(pi/2),sin(0) ];
a=[0.15, 0.770, 0.10, 0, 0, 0];
b=[0.525, 0, 0, 0.740, 0, 0.10];

% Transformation outil
beta=-25.2*pi/180;
Teffe_tool=[cos(beta), 0, -sin(beta), -0.0785;...
            0, 1, 0, 0; ...
            sin(beta), 0, cos(beta), 0.154;...
            0, 0, 0, 1];

A1=matrix( angle( teta(1), lu(1), u(1) ),membrure( teta(1),a(1),b(1) )   );
A2=matrix( angle( teta(2), lu(2), u(2) ),membrure( teta(2),a(2),b(2) )   );
```

```
A3=matrix( angle( teta(3), lu(3), u(3) ),membrure( teta(3),a(3),b(3) )   );
A4=matrix( angle( teta(4), lu(4), u(4) ),membrure( teta(4),a(4),b(4) )   );
A5=matrix( angle( teta(5), lu(5), u(5) ),membrure( teta(5),a(5),b(5) )   );
A6=matrix( angle( teta(6), lu(6), u(6) ),membrure( teta(6),a(6),b(6) )   );

Atot=A1*A2*A3*A4*A5*A6*Teffe_tool;

% on trouve la position et l`orientation de l`outil
p=Atot(1:3,4); Q=Atot(1:3,1:3);

% on construit la matrice jacobienne
    % 1er- on definit le vecteur unitaire et les referentiels des vecteurs
    % unitaire par rapport à la base
    k=[0, 0, 1]';

    Q11=A1(1:3,1:3);        Q12=Q11*A2(1:3,1:3);
    Q13=Q12*A3(1:3,1:3);    Q14=Q13*A4(1:3,1:3);
    Q15=Q14*A5(1:3,1:3);    Q16=Q15*A6(1:3,1:3);

    e1=k;   e2=Q11*k; e3=Q12*k; e4=Q13*k; e5=Q14*k; e6=Q15*k;

    % on trouve les rayons ri definit par rapport au referentiel precedent

    r6=A6(1:3,4);
    r5=A5(1:3,4)+Q15(1:3,1:3)*r6;
    r4=A4(1:3,4)+Q14(1:3,1:3)*r5;
    r3=A3(1:3,4)+Q13(1:3,1:3)*r4;
    r2=A2(1:3,4)+Q12(1:3,1:3)*r3;
    r1=A1(1:3,4)+Q11(1:3,1:3)*r2;

    % on ramene tous ces rayons ri dans le meme referentiel de base

    r1=r1;
    r2=Q11*r2;
    r3=Q12*r3;
    r4=Q13*r4;
    r5=Q14*r5;
    r6=Q15*r6;

    % on construit la matrice Jacobienne
    J=zeros(6,6);
    J(1:6,1)=[e1;cross(e1,r1)];          J(1:6,2)=[e2;cross(e2,r2)];
    J(1:6,3)=[e3;cross(e3,r3)];          J(1:6,4)=[e4;cross(e4,r4)];
    J(1:6,5)=[e5;cross(e5,r5)];          J(1:6,6)=[e6;cross(e6,r6)];
```

Vect.m

```
function a=vect(A)
% computing vect A
a=1/2*[A(3,2)-A(2,3); A(1,3)-A(3,1); A(2,1)-A(1,2)];
```

enregistre.m

```
function etat=enregistre_trj(trj, dT, filename)
%Expects a dT, a filename and a  nx6 matrix in the following format
%[  teta1_1 teta1_2
%   teta2_1 teta2_2
%   teta3_1 teta3_2
%   teta4_1 teta4_2
%   teta5_1 teta5_2
%   teta6_1 teta6_2 ]  here with 6 tetas and 2 points

etat=0;
trj=trj';
[a, z]=size(trj);       % Dimensions of trj

if a==6

    fid=fopen(filename, 'w');  %trying to write the file
    if fid==0
            disp('Cant write the file:'); filename
        else
```

```
            fprintf(fid, '%10.0f ', z);      %Line numbers
            fprintf(fid, '%10.5f \n', dT);   %dT
            for i=1:z
            fprintf(fid, '%5.9f %5.9f %5.9f %5.9f %5.9f %5.9f \n', trj(:,i));
            % writing the lines with the angles
            end
            fclose(fid);
            etat=1;
    end
 else
    disp('the matrix does not have 6 angles!');
 end
```

angle.m

```
function Q=angle(teta, lu, u)
% Computing the orientation matrix
Q=[cos(teta), -lu*sin(teta), u*sin(teta);...
   sin(teta), lu*cos(teta), -u*cos(teta);...
    0, u, lu];
```

Membrure.m

```
function a=membrure(teta,a,b )
 % computing the a matrix

a=[a*cos(teta);a*sin(teta); b];
```

ANNEX B

FILES