Mechatronic Systems Engineering
Faculty of Applied Science
SIMON FRASER UNIVERSITY

MSE429 – Advanced Kinematics for Robotic System –
- Individual Project #4 Report -

Jacobian and Dynamics Simulation of the 6-DoF manipulator using MATLAB

December 7th, 2020

Course Instructor:

Flavio Firmani

Author:

Dmitrii Gusev 301297008

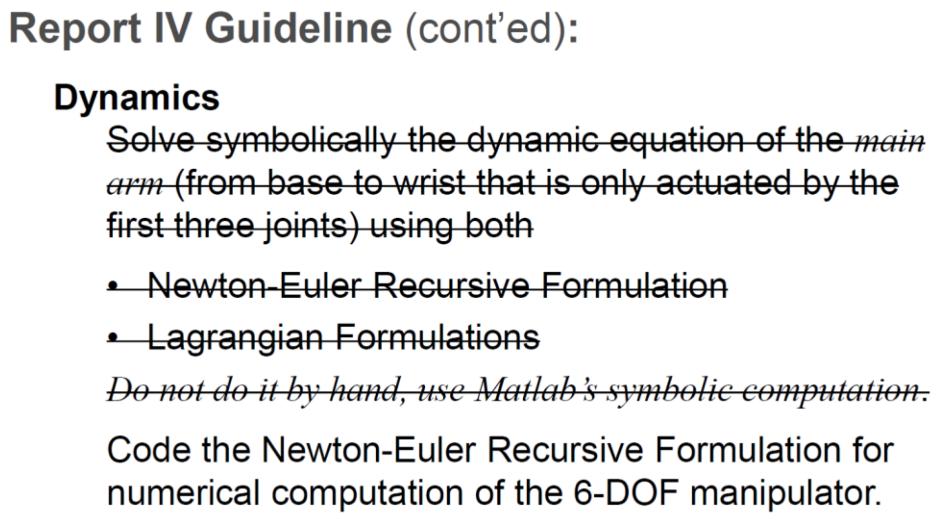
Abstract:

Throughout my experience of working in industrial field as electrical and controls engineer, I have noticed a huge issue related to the overall productivity of the industrial automation teams. Nowadays, we live in the age of automation, however automated machines themselves are hand-built most of the time. Electricians and mechanical assembly people need to work countless hours since early morning till late evening to ensure proper quality of the designed machine. At the same time, Controls Engineers need to stay at the office during this time to ensure that every design challenge electricians face can be fixed in documentation and PLC code.

Assembly of the industrial electrical panel is a messy and time-consuming task that requires electricians and engineers to perform same routine tasks repeatedly: wiring terminal blocks, PLCs, and many other components. I decided to come up with a machine that can simplify this process by using a precise robotic manipulator controlled by the GPU-accelerated computer vision. This report covers the trajectory generation methods for my robotic manipulator.

Project Guidelines (07 / 12 / 2020):

1. Calculate Jacobian and determine Singular Configuration(s)
2. Model Links
3. Newton-Euler Recursive Formulation: Numerical 6-DoF solution
4. Plot forces and torques along the trajectory from the Project III.



Report IV Guideline (cont'ed):

Dynamics

~~Solve symbolically the dynamic equation of the *main arm* (from base to wrist that is only actuated by the first three joints) using both~~

- ~~Newton-Euler Recursive Formulation~~
- ~~Lagrangian Formulations~~

~~*Do not do it by hand, use Matlab's symbolic computation*~~.

Code the Newton-Euler Recursive Formulation for numerical computation of the 6-DOF manipulator.

Problem Formulation:

Typical industrial control panel includes multiple components (e.g. PLCs, relays, switches etc) mounted on a DIN rail with wiring flowing to the top and bottom of the components directly into wiring trays as shown on the picture below. This wiring is usually specified by the electrical and controls drawing done in AutoCAD Electrical or EPLAN. Wires are numbered in according to the design guideline and it is usually straight forward how to interconnect all the components of the electrical panel. Even though, electrical panel may include tasks that requires additional attention from electricians and engineers, most of the time wiring is just a time-consuming routine both for engineering team and assembly team. I would like to find a solution that will allow to optimize assembly time by automating the easiest repetitive tasks such as wiring of the terminal blocks, relays and PLCs by using a robotic manipulator controlled by the GPU-accelerated computer vision.

This report does not focus on the controls part of the project (code, motor control etc), however main goal of this project is to design the forward kinematics model of the desired machine. Solution of this problem requires the following steps to be done:

1. Apply knowledge of the Forward and Inverse Kinematics to identify robot's movement
2. Perform Inverse Kinematic calculations in order to change joint displacements based on the desired end-effector position
3. Use Inverse Kinematics method to generate the movement trajectory for the robotic arm

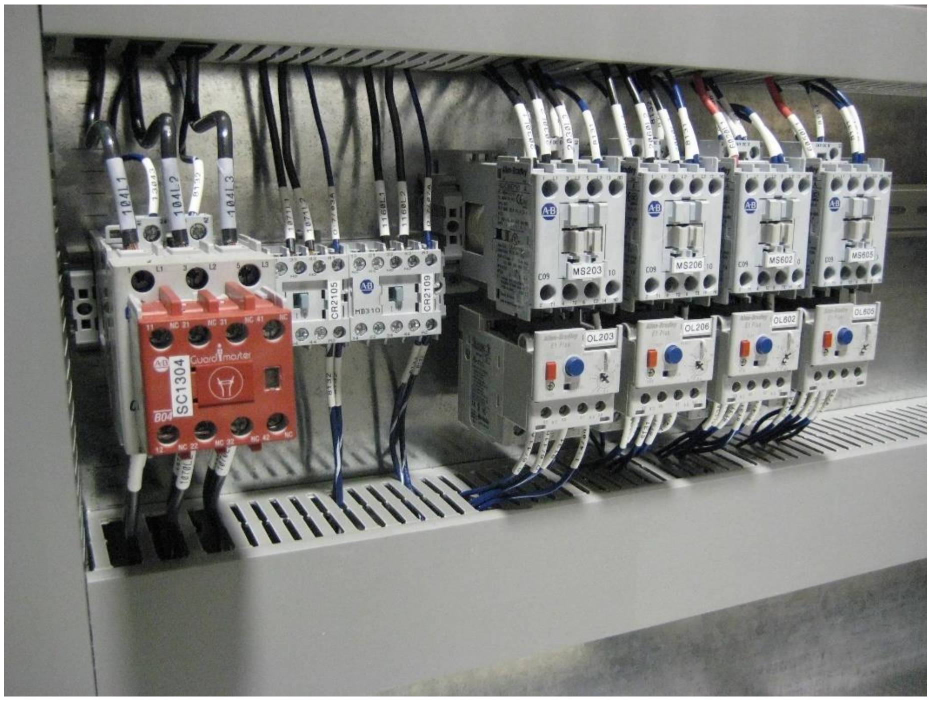


Figure #1: Typical Industrial Electrical Panel

Jacobian Calculation:

Manipulator general design is shown below. P (perp) R2 || R3 (perp) (R4 || R5 || R6)sph:

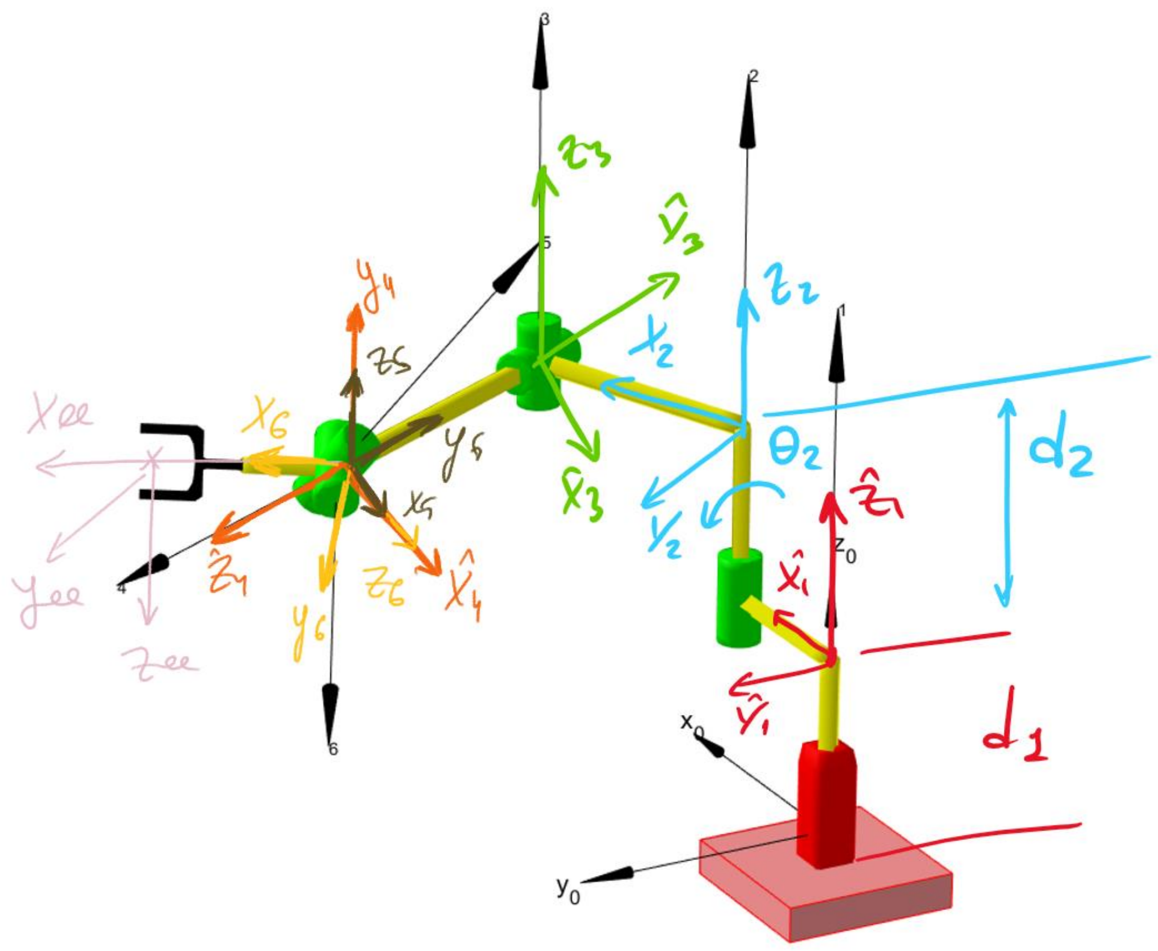
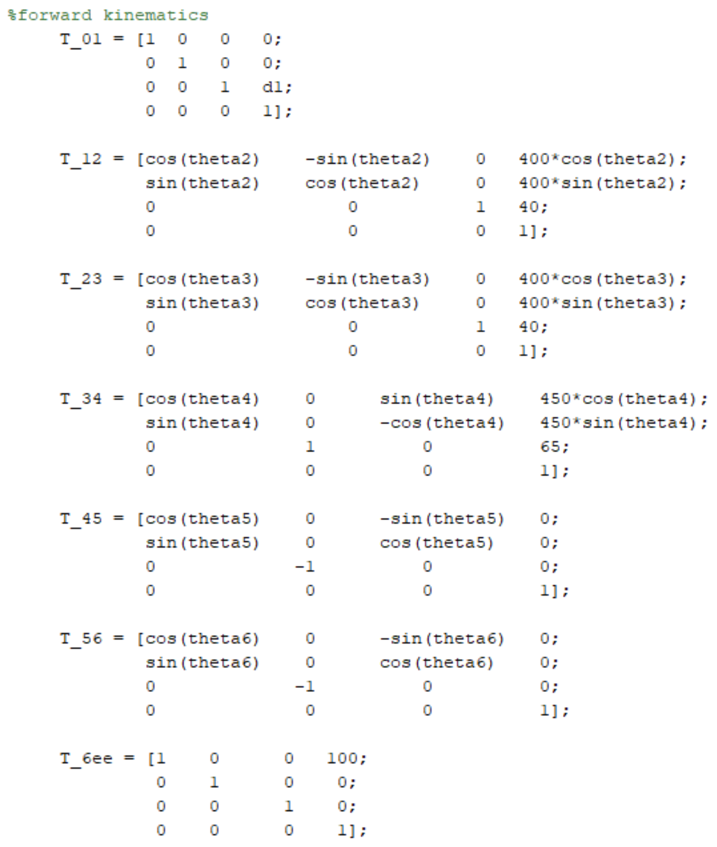


Figure #2: Model Layout with attached frames

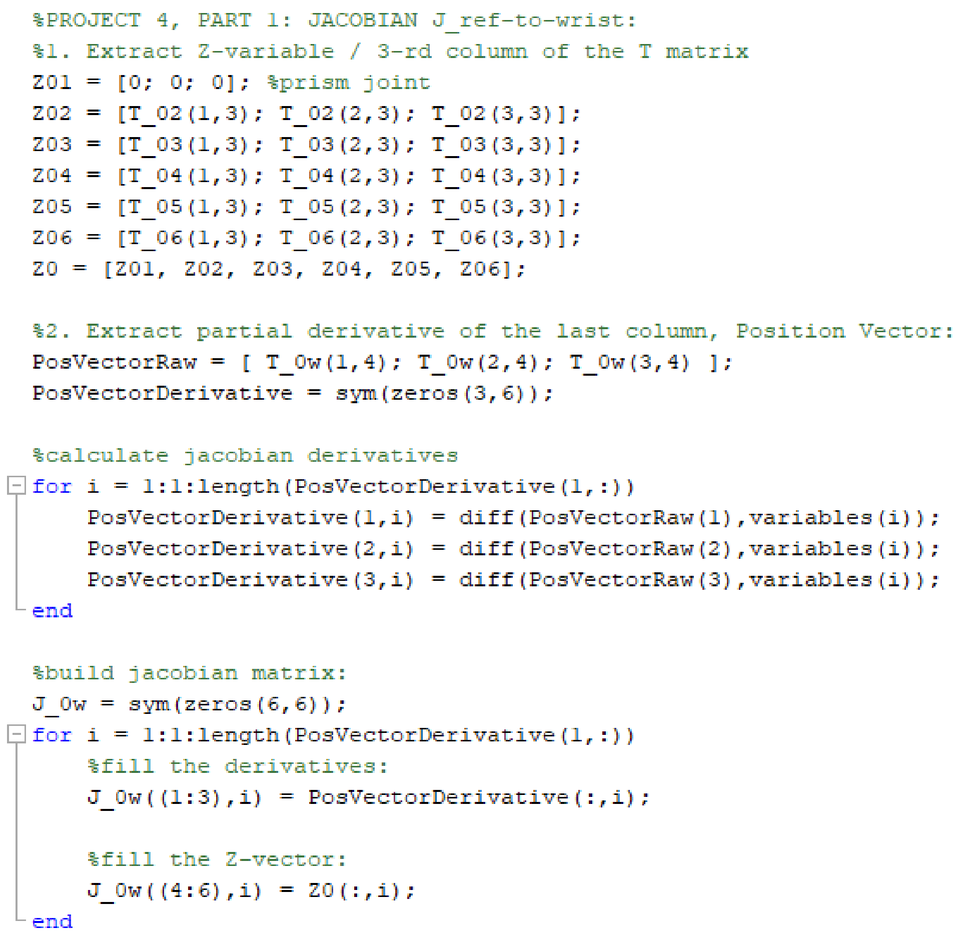
Jacobian calculator shown in this section uses transition matrices created in the Project Part I. They are shown below:



```
%forward kinematics
    T_01 = [1  0   0   0;
            0  1   0   0;
            0  0   1   d1;
            0  0   0   1];

    T_12 = [cos(theta2)    -sin(theta2)    0   400*cos(theta2);
            sin(theta2)    cos(theta2)     0   400*sin(theta2);
            0                  0           1   40;
            0                  0           0   1];

    T_23 = [cos(theta3)    -sin(theta3)    0   400*cos(theta3);
            sin(theta3)    cos(theta3)     0   400*sin(theta3);
            0                  0           1   40;
            0                  0           0   1];

    T_34 = [cos(theta4)    0      sin(theta4)     450*cos(theta4);
            sin(theta4)    0      -cos(theta4)    450*sin(theta4);
            0              1          0           65;
            0              0          0           1];

    T_45 = [cos(theta5)    0      -sin(theta5)    0;
            sin(theta5)    0      cos(theta5)     0;
            0             -1          0           0;
            0              0          0           1];

    T_56 = [cos(theta6)    0      -sin(theta6)    0;
            sin(theta6)    0      cos(theta6)     0;
            0             -1          0           0;
            0              0          0           1];

    T_6ee = [1    0      0    100;
             0    1      0     0;
             0    0      1     0;
             0    0      0     1];
```

Figure #3: Transition Matrices for the 6-DoF manipulator

I used material covered in the Chapter 7 to create the Jacobian Calculator. Z-vector uses Rotation Matrix located in the first 3x3 section of the Transition Matrix. Z-vector for the first joint is going to be [0;0;0] because first joint is a linear actuator. Other Z-vectors are derived directly from the transition matrices above. Jacobian calculator is shown on the next page.



```
%PROJECT 4, PART 1: JACOBIAN J_ref-to-wrist:
%1. Extract Z-variable / 3-rd column of the T matrix
Z01 = [0; 0; 0]; %prism joint
Z02 = [T_02(1,3); T_02(2,3); T_02(3,3)];
Z03 = [T_03(1,3); T_03(2,3); T_03(3,3)];
Z04 = [T_04(1,3); T_04(2,3); T_04(3,3)];
Z05 = [T_05(1,3); T_05(2,3); T_05(3,3)];
Z06 = [T_06(1,3); T_06(2,3); T_06(3,3)];
Z0 = [Z01, Z02, Z03, Z04, Z05, Z06];

%2. Extract partial derivative of the last column, Position Vector:
PosVectorRaw = [ T_0w(1,4); T_0w(2,4); T_0w(3,4) ];
PosVectorDerivative = sym(zeros(3,6));

%calculate jacobian derivatives
for i = 1:1:length(PosVectorDerivative(1,:))
    PosVectorDerivative(1,i) = diff(PosVectorRaw(1),variables(i));
    PosVectorDerivative(2,i) = diff(PosVectorRaw(2),variables(i));
    PosVectorDerivative(3,i) = diff(PosVectorRaw(3),variables(i));
end

%build jacobian matrix:
J_0w = sym(zeros(6,6));
for i = 1:1:length(PosVectorDerivative(1,:))
    %fill the derivatives:
    J_0w((1:3),i) = PosVectorDerivative(:,i);

    %fill the Z-vector:
    J_0w((4:6),i) = Z0(:,i);
end
```

Figure #3: Jacobian Calculator.

Code shown above allows to generate the Jacobian 6 x 6 matrix. Calculated Jacobian uses Joint Directions and Derivatives method to calculate Jacobian from reference frame 0 to wrist. Result is shown below.

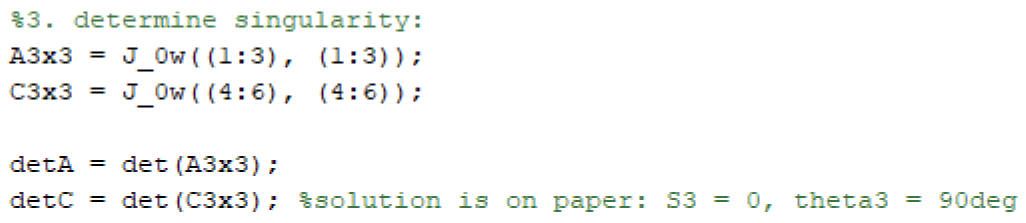
A_{3x3}

$$
{}^{0}J_w = \begin{bmatrix}
0 & -400S_2 - 400C_2S_3 - 400C_3S_2 & -400C_2S_3 - 400C_3S_2 & 0 & 0 & 0 \\
0 & 400C_2 + 400C_2C_3 - 400S_2S_3 & 400C_2C_3 - 400S_2S_3 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & C_4(C_2S_3 + C_3S_2) + S_4(C_2C_3 - S_2S_3) & -S_5(C_4(C_2C_3 - S_2S_3) - S_4(C_2S_3 + C_3S_2)) & -C_6(C_4(C_2S_3 + C_3S_2) + S_4(C_2C_3 - S_2S_3)) - C_5S_6(C_4(C_2C_3 - S_2S_3)) - S_4(C_2S_3 + C_3S_2)) \\
0 & 0 & 0 & S_4(C_2S_3 + C_3S_2) - C_4(C_2C_3 - S_2S_3) & -S_5(C_4(C_2S_3 + C_3S_2) + S_4(C_2C_3 - S_2S_3)) & C_6(C_4(C_2C_3 - S_2S_3) - S_4(C_2S_3 + C_2S_2)) - C_5S_6(C_4(C_2S_3 + C_3S_2)) + S_4(C_2C_3 - S_2S_3)) \\
0 & 1 & 1 & 0 & C_5 & -S_5S_6
\end{bmatrix}
$$

C_{3x3}

Figure #4: Jacobian of the 6-DoF manipulator

Code below allows to calculate singularities and we use hand-done calculations to find final form of the singularity.



```
%3. determine singularity:
A3x3 = J_0w((1:3), (1:3));
C3x3 = J_0w((4:6), (4:6));

detA = det(A3x3);
detC = det(C3x3); %solution is on paper: S3 = 0, theta3 = 90deg
```

Figure #5: Singularity calculator.

Using the fact that sum of the squared cosines and sines equals 1, we can simplify the singularity calculation. Resulting singularity solution means that our manipulator has two potential solutions depending on the angle Theta 3. Theta 4 to 6 are not matter because final position of the manipulator is mostly determined by the main arm up to wrist joint that contains links #1, #2 and #3.

2) Calculate singularity:

$$|{}^{0}J_w| = |A_{3\times3}||C_{3\times3}| = 0 \longrightarrow$$

$$|A_{3\times3}| = 160000 C_2^2 S_3 + 160000 S_2^2 S_3$$

$$\begin{aligned}|C_{3\times3}| = {} & S_6 C_2^2 C_3^2 C_4^2 C_5^2 + S_6 C_2^2 C_3^2 C_4^2 S_5^2 + S_6 C_2^2 C_3^2 C_5^2 S_4^2 + S_6 C_2^2 C_3^2 S_4^2 S_5^2 + \\ & + S_6 C_2^2 C_4^2 C_5^2 S_3^2 + S_6 C_2^2 C_4^2 S_3^2 S_5^2 + S_6 C_2^2 C_5^2 S_3^2 S_4^2 + S_6 C_2^2 S_3^2 S_4^2 S_5^2 + \\ & + S_6 C_3^2 C_4^2 C_5^2 S_2^2 + S_6 C_3^2 C_4^2 S_2^2 S_5^2 + S_6 C_3^2 C_5^2 S_2^2 S_4^2 + S_6 C_3^2 S_2^2 S_4^2 S_5^2 + \\ & + S_6 C_4^2 C_5^2 S_2^2 S_3^2 + S_6 C_4^2 S_2^2 S_3^2 S_5^2 + S_6 C_5^2 S_2^2 S_3^2 S_4^2 + S_6 S_2^2 S_3^2 S_4^2 S_5^2 = \\ = {} & S_6 C_2^2 C_3^2 C_4^2 + S_6 C_2^2 C_3^2 S_4^2 + S_6 C_2^2 C_4^2 S_3^2 + S_6 C_2^2 S_3^2 S_4^2 + \\ & + S_6 C_3^2 C_4^2 S_2^2 + S_6 C_3^2 S_2^2 S_4^2 + S_6 C_4^2 S_2^2 S_3^2 + S_6 S_2^2 S_3^2 S_4^2 = \\ = {} & S_6 C_2^2 C_3^2 + S_6 C_2^2 S_3^2 + S_6 C_3^2 S_2^2 + S_6 S_2^2 S_3^2 = \\ = {} & S_6 C_2^2 + S_6 S_2^2 = S_6\end{aligned}$$

$$\begin{aligned}|A_{3\times3}||C_{3\times3}| &= S_6 (160000 C_2^2 S_3 + 160000 S_2^2 S_3) = \\ &= 160000 S_3 S_6 = 0\end{aligned}$$

$$S_3 = 0 \implies \boxed{\theta_3 = \pm 90^\circ} \quad \boxed{\theta_{1,2} = 0^\circ}$$

Figure #6: Singularity solution

Resulting singular position is represented below using SolidWorks.

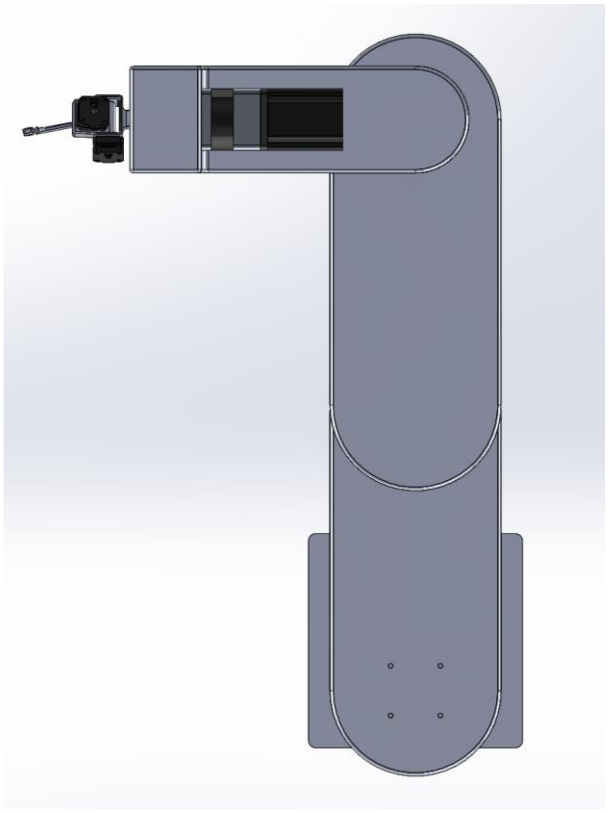
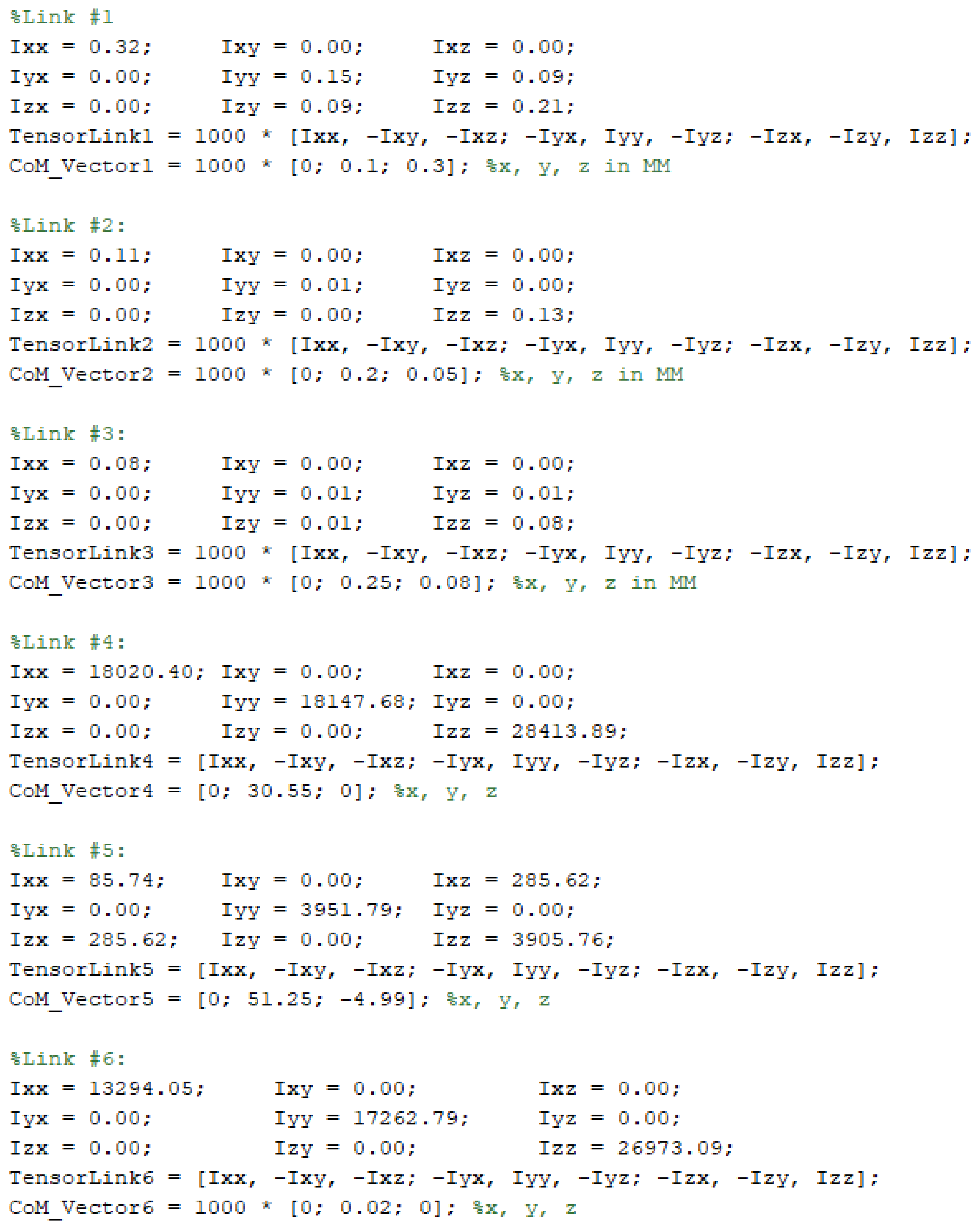


Figure #7: Singular Position (1 out of 2 possible)

Position #2 will be exactly same except for the Link #3 being turned by -90 degrees instead of 90 degrees.

Link Modelling:

Using SolidWorks we can model links by finding their center of mass, position vector from the center of joint rotation to the center of mass as well as find Inertia Tensors:



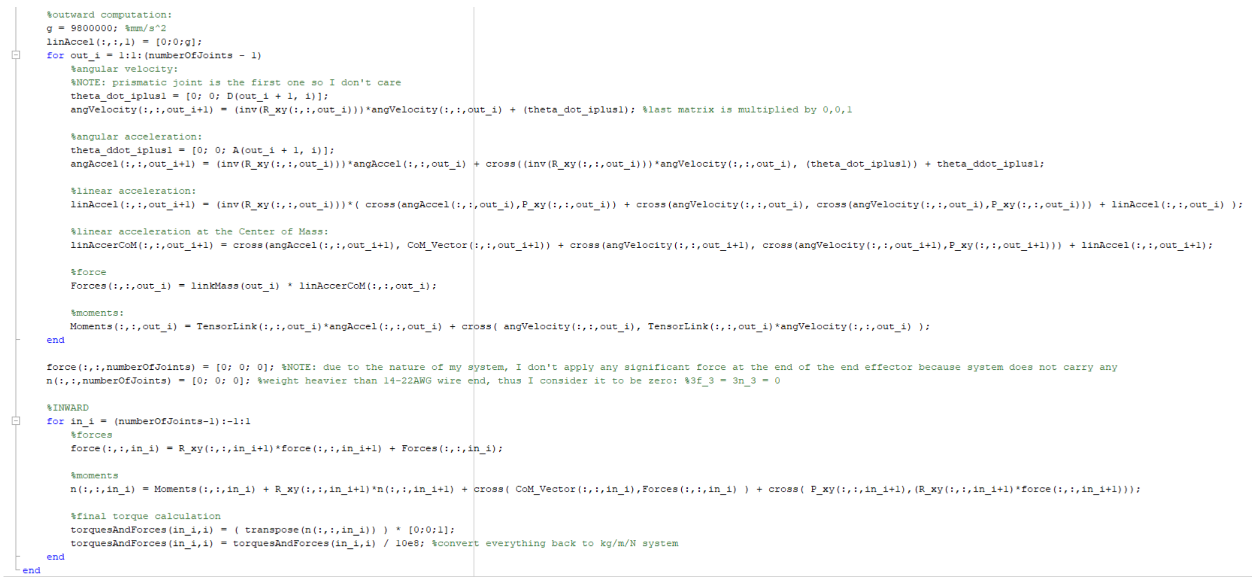
```
%Link #1
Ixx = 0.32;     Ixy = 0.00;     Ixz = 0.00;
Iyx = 0.00;     Iyy = 0.15;     Iyz = 0.09;
Izx = 0.00;     Izy = 0.09;     Izz = 0.21;
TensorLink1 = 1000 * [Ixx, -Ixy, -Ixz; -Iyx, Iyy, -Iyz; -Izx, -Izy, Izz];
CoM_Vector1 = 1000 * [0; 0.1; 0.3]; %x, y, z in MM

%Link #2:
Ixx = 0.11;     Ixy = 0.00;     Ixz = 0.00;
Iyx = 0.00;     Iyy = 0.01;     Iyz = 0.00;
Izx = 0.00;     Izy = 0.00;     Izz = 0.13;
TensorLink2 = 1000 * [Ixx, -Ixy, -Ixz; -Iyx, Iyy, -Iyz; -Izx, -Izy, Izz];
CoM_Vector2 = 1000 * [0; 0.2; 0.05]; %x, y, z in MM

%Link #3:
Ixx = 0.08;     Ixy = 0.00;     Ixz = 0.00;
Iyx = 0.00;     Iyy = 0.01;     Iyz = 0.01;
Izx = 0.00;     Izy = 0.01;     Izz = 0.08;
TensorLink3 = 1000 * [Ixx, -Ixy, -Ixz; -Iyx, Iyy, -Iyz; -Izx, -Izy, Izz];
CoM_Vector3 = 1000 * [0; 0.25; 0.08]; %x, y, z in MM

%Link #4:
Ixx = 18020.40; Ixy = 0.00;     Ixz = 0.00;
Iyx = 0.00;     Iyy = 18147.68; Iyz = 0.00;
Izx = 0.00;     Izy = 0.00;     Izz = 28413.89;
TensorLink4 = [Ixx, -Ixy, -Ixz; -Iyx, Iyy, -Iyz; -Izx, -Izy, Izz];
CoM_Vector4 = [0; 30.55; 0]; %x, y, z

%Link #5:
Ixx = 85.74;    Ixy = 0.00;     Ixz = 285.62;
Iyx = 0.00;     Iyy = 3951.79;  Iyz = 0.00;
Izx = 285.62;   Izy = 0.00;     Izz = 3905.76;
TensorLink5 = [Ixx, -Ixy, -Ixz; -Iyx, Iyy, -Iyz; -Izx, -Izy, Izz];
CoM_Vector5 = [0; 51.25; -4.99]; %x, y, z

%Link #6:
Ixx = 13294.05;     Ixy = 0.00;         Ixz = 0.00;
Iyx = 0.00;         Iyy = 17262.79;     Iyz = 0.00;
Izx = 0.00;         Izy = 0.00;         Izz = 26973.09;
TensorLink6 = [Ixx, -Ixy, -Ixz; -Iyx, Iyy, -Iyz; -Izx, -Izy, Izz];
CoM_Vector6 = 1000 * [0; 0.02; 0]; %x, y, z
```

Figure #8: Inertia Tensors (around Center of Mass) and Center of Mass position vector

Newton-Euler Recursive Numerical Formulation:

Next step of the project was to create recursive Newton-Euler numerical formulation to calculate the required torques and forces that must be applied to the joints in order to make manipulator work. It is important to note that my design's goal is to manipulate light wires of 14 to 22 AWG which weight nothing compare to the overall manipulator, thus I assumed 0 force being applied to the End Effector. Cycles for Outward and Inward computation shown below is repeated for the "number of joints = 6" (any other number can be specified in my code as long as transition matrices present for this). Overall joint computation cycle results with a number representing a torque or force required to be applied to the joint.

At the final step, every number is updated to the final value in terms of meters, kg and Newton because initial calculations were done in mm's for all previous sections of the report. We repeat this cycle about 154 times or as long as we have data in our via_points_match_VA.m solution for matrices D and A.



```
    %outward computation:
    g = 9800000; %mm/s^2
    linAccel(:,:,1) = [0;0;g];
    for out_i = 1:1:(numberOfJoints - 1)
        %angular velocity:
        %NOTE: prismatic joint is the first one so I don't care
        theta_dot_iplus1 = [0; 0; D(out_i + 1, i)];
        angVelocity(:,:,out_i+1) = (inv(R_xy(:,:,out_i)))*angVelocity(:,:,out_i) + (theta_dot_iplus1); %last matrix is multiplied by 0,0,1

        %angular acceleration:
        theta_ddot_iplus1 = [0; 0; A(out_i + 1, i)];
        angAccel(:,:,out_i+1) = (inv(R_xy(:,:,out_i)))*angAccel(:,:,out_i) + cross((inv(R_xy(:,:,out_i)))*angVelocity(:,:,out_i), (theta_dot_iplus1)) + theta_ddot_iplus1;

        %linear acceleration:
        linAccel(:,:,out_i+1) = (inv(R_xy(:,:,out_i)))*( cross(angAccel(:,:,out_i),P_xy(:,:,out_i)) + cross(angVelocity(:,:,out_i), cross(angVelocity(:,:,out_i),P_xy(:,:,out_i))) + linAccel(:,:,out_i) );

        %linear acceleration at the Center of Mass:
        linAccerCoM(:,:,out_i+1) = cross(angAccel(:,:,out_i+1), CoM_Vector(:,:,out_i+1)) + cross(angVelocity(:,:,out_i+1), cross(angVelocity(:,:,out_i+1),P_xy(:,:,out_i+1))) + linAccel(:,:,out_i+1);

        %force
        Forces(:,:,out_i) = linkMass(out_i) * linAccerCoM(:,:,out_i);

        %moments:
        Moments(:,:,out_i) = TensorLink(:,:,out_i)*angAccel(:,:,out_i) + cross( angVelocity(:,:,out_i), TensorLink(:,:,out_i)*angVelocity(:,:,out_i) );
    end

    force(:,:,numberOfJoints) = [0; 0; 0]; %NOTE: due to the nature of my system, I don't apply any significant force at the end of the end effector because system does not carry any
    n(:,:,numberOfJoints) = [0; 0; 0]; %weight heavier than 14-22AWG wire end, thus I consider it to be zero: %3f_3 = 3n_3 = 0

    %INWARD
    for in_i = (numberOfJoints-1):-1:1
        %forces
        force(:,:,in_i) = R_xy(:,:,in_i+1)*force(:,:,in_i+1) + Forces(:,:,in_i);

        %moments
        n(:,:,in_i) = Moments(:,:,in_i) + R_xy(:,:,in_i+1)*n(:,:,in_i+1) + cross( CoM_Vector(:,:,in_i),Forces(:,:,in_i) ) + cross( P_xy(:,:,in_i+1),(R_xy(:,:,in_i+1)*force(:,:,in_i+1)));

        %final torque calculation
        torquesAndForces(in_i,i) = ( transpose(n(:,:,in_i)) ) * [0;0;1];
        torquesAndForces(in_i,i) = torquesAndForces(in_i,i) / 10e8; %convert everything back to kg/m/N system
    end
end
```

Figure #9: Newton-Euler Recursive Formulation

Additional well-commented code is attached to this report. Please, download it and run if you would like to check my solution. Next section covers the plotting of the torquesAndForces vector calculated in this step.

Forces and Torques plots:

As mentioned in the previous step, our End Effector experiences close-to-zero load from the external materials, thus our forces and torques are determined only by the inertias of the whole manipulator. Results are shown below:

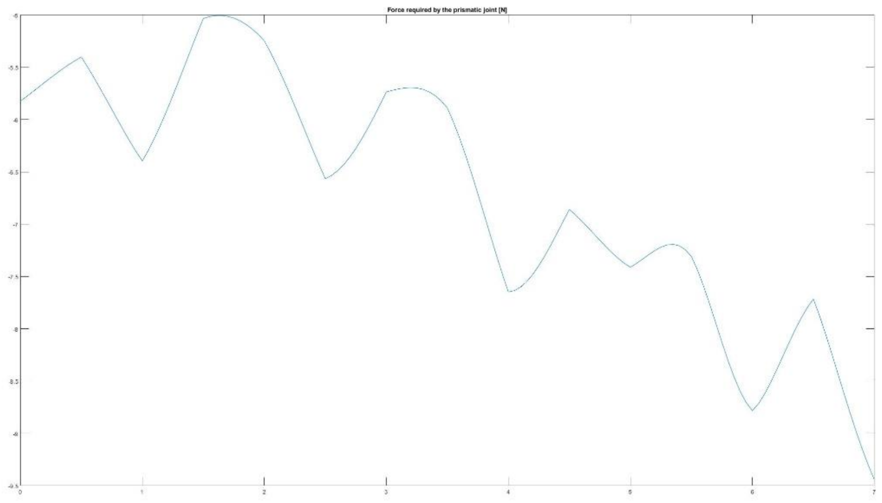


Figure #10: Force required by the prismatic joint (linear actuator) [N]

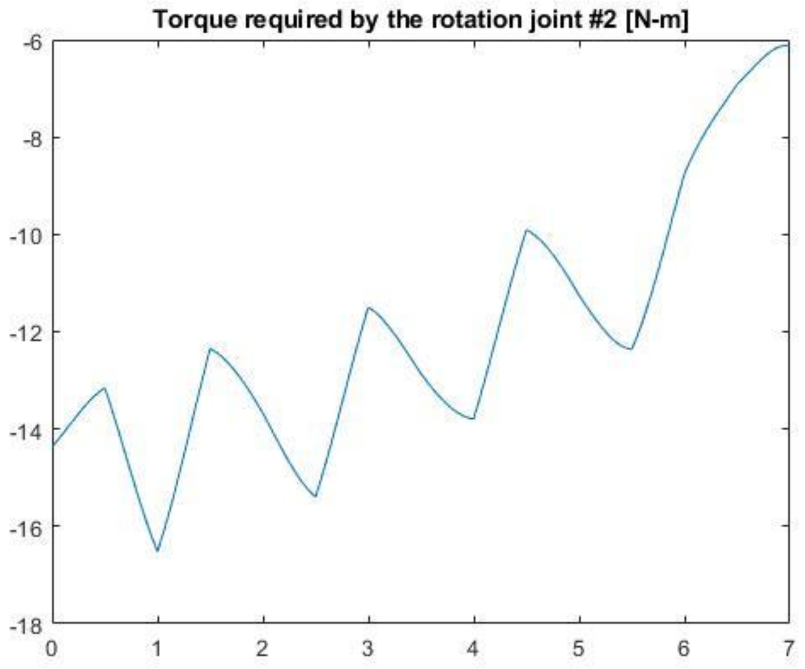


Figure #11: Torque required by the revolute joint #2 [N*m]

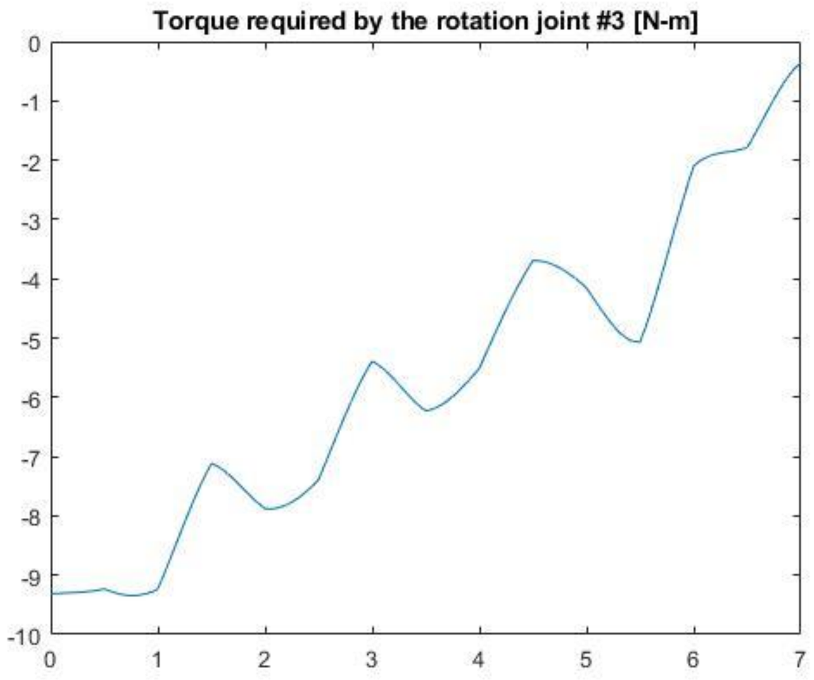


Figure #12: Torque required by the revolute joint #3 [N*m]

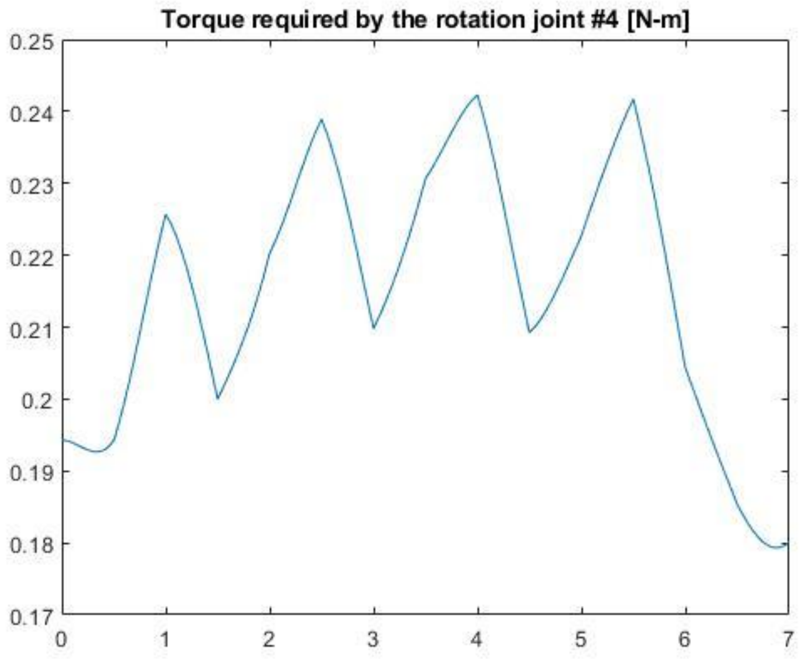


Figure #13: Torque required by the revolute joint #4 [N*m]

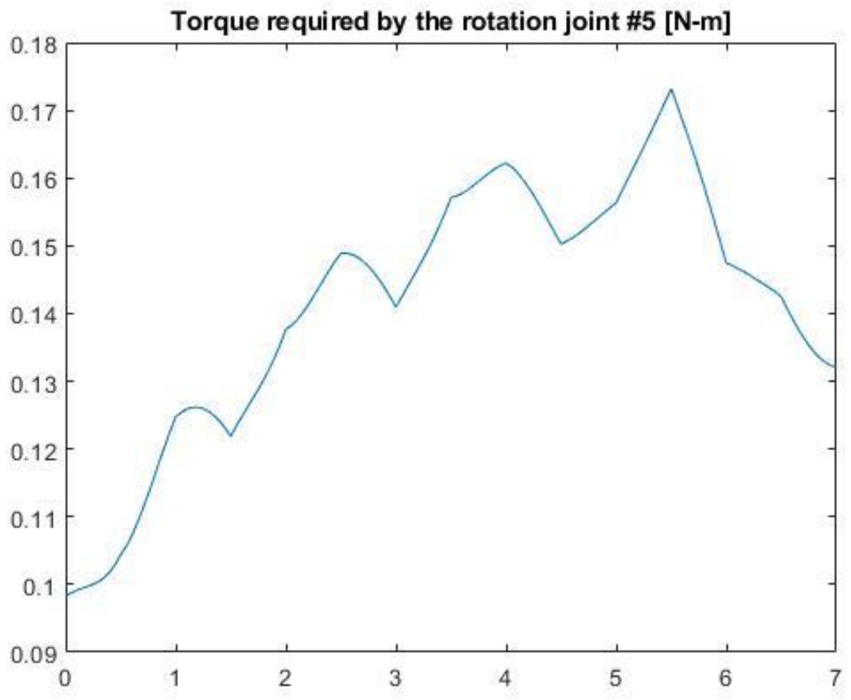


Figure #14: Torque required by the revolute joint #5 [N*m]

Additionally, we can compare the resulting theoretical torques with electric motors specified in the Report #1 to prove that initial motor selection was correct. Joint #1 is operated using linear actuator and linear actuator model was not initially specified. However, Joints 2 and 3 use NEMA-34 with gear reduction (if needed) and light Joints 4, 5 and 6 use widely used NEMA-17 with direct drive.

NEMA-34 can provide up to 63 kg-cm of torque. 63 Kg-cm = 6.1781895 N-m. We can note that joints #2 and #3 will require gear reduction of 2.5 to ensure proper torque can be applied by the NEMA-34.

NEMA-17 provides 3.2 kg-cm or 0.314 N-m of torque and Joints #4, 5 and 6 are 100% satisfied with the basic performance of the chosen stepper motor.

Conclusion:

This is the last report of the Individual Project series for the MSE429 course taken in Fall 2020. This report covers the development of the Jacobian and Dynamics for the 6-DoF robot designed to perform automated industrial electrical panel assembly. Overall, I managed to perform a transition of the SolidWorks design into Forward and Inverse kinematics using D-H parameters, perform motion simulation and find velocities and accelerations of the joints as well as calculate Jacobian in order to represent robot's dynamics in terms of required torques and forces. Finally, I have managed to compare my theoretical computation with initial robot's actuator selection in order to verify that design is fully valid.

Appendix:

MATLAB CODES ARE ATTACHED TO THIS REPORT WITHIN THE SAME ARCHIVE