A Tutorial-oriented Approach to ARGESIM Benchmark C11 'SCARA Robot' in MATLAB, Simulink and Stateflow

Johannes Leindecker, Maximilian Zechmeister-Machhart, Felix Gauss, Philipp Wiegard

Inst. of Mechanics and Mechatronics & Inst. of Analysis and Scientific Computing, TU Wien, Wiedner Hauptstrasse 8-10, 1040 Vienna, Austria; *johannes.leindecker@tuwien.ac.at*

SNE 31(2), 2021, 101 - 110, DOI: 10.11128/sne.31.bne11.10569
Received: January 13, 2021; Revised April 25, 2021;
Accepted: May 15, 2021
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. This Educational Benchmark Note presents a tutorial-oriented approach to 'ARGESIM Benchmark C11 SCARA Robot', an SNE Student Note compiled by master students of Mechanical Engineering. Students have described for students necessary state space models for the SCARA robot, and algorithmic preparations for implementation in MATLAB and Simulink, including proper state limitations. Furthermore, for collision handling of the robot movement, benefits of a state automata–based implementation in Stateflow is given. The simulations compare explicit and implicit model versions and efficiency of different ODE solvers, and present a basic animation.

Model Description

Mechanical System. The mechanical system, that is investigated in this Benchmark, is a three-axes robot arm, called SCARA (Selective Compliance Assembly Robot Arm). It has two vertical revolute joints and one vertical prismatic joint, with which it can change the position of its mounted tool tip.

One can describe the equations of motion in compact form of an implicit second order system of differential equations:

$$\boldsymbol{M}\ddot{\vec{q}} = \vec{b} \qquad (1)$$

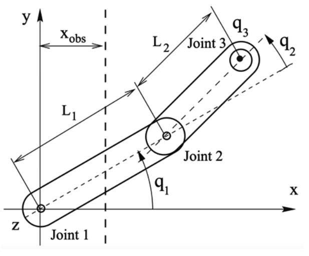


Figure 1: Mechanical structure of the SCARA robot [1].

In this case, vector $\vec{q}$ includes the joint angles q_1 and q_2 and the joint distance q_3. The mass matrix $\boldsymbol{M}$ is a 3x3 block-diagonal matrix:

$$\boldsymbol{M} = \begin{pmatrix} ma_{11} & ma_{12} & 0 \\ ma_{21} & ma_{22} & 0 \\ 0 & 0 & ma_{33} \end{pmatrix} \qquad (2)$$

To calculate the moments of inertia (eq. (8) - (10)), we assume that there is a homogeneous mass distribution of the two rods (mass m_1 and m_2) along the length L_1 and L_2, the mass of the load and the vertical drive motor is known and the moment of inertia of the rotating parts is defined.

$$ma_{11} = \Theta_1 + 2\Theta_2 \cos(q_2) + \Theta_3 \qquad (3)$$
$$ma_{12} = \Theta_2 \cos(q_2) + \Theta_3 \qquad (4)$$
$$ma_{21} = ma_{12} \qquad (5)$$
$$ma_{22} = \Theta_3 \qquad (6)$$
$$ma_{33} = m_{3L} + \Theta_{3mot} u_3^2 \qquad (7)$$
$$\Theta_1 = \left(\frac{m_1}{3} + m_2 + m_3\right) L_1^2 \qquad (8)$$
$$\Theta_2 = \left(\frac{m_2}{2} + m_3\right) L_1 L_2 \qquad (9)$$
$$\Theta_3 = \left(\frac{m_2}{3} + m_3\right) L_2^2 \qquad (10)$$
$$m_3 = m_{3A} + m_{3L} \qquad (11)$$

Vector $\vec{b}$, which describes the right-hand side of the differential equation, is a function of the joint torques $T_1(t)$ and $T_2(t)$ and the joint force $T_3(t)$.

$$b_1 = T_1 + \Theta_2(2\dot{q}_1\dot{q}_2 + \dot{q}_2^2)\, sin(q_2) \qquad (12)$$
$$b_2 = T_2 - \Theta_2 \dot{q}_1^2\, sin(q_2) \qquad (13)$$
$$b_3 = T_3 - m_{3L} g \qquad (14)$$

Servo Motor and PD-Control. Due to the usage of a PD-control (Proportional-Derivative), which controls position errors and joint velocities, there is a limitation for the mechanical system that has to be observed. This limitation is the armature voltage (eq. (15)), restricted by $\pm U_{ireg}^{max}$ (eq. (16)) during the regular operation of the SCARA. The armature voltage is a function of the proportional gains P_i, the derivative gains D_i and the target and current value of the joints:

$$U_i = P_i(\hat{q}_i - q_i) - D_i \dot{q}_i, \quad i = 1,2,3 \qquad (15)$$

$$U_{ai} = [-U_i^{max} \leq U_i \leq U_i^{max}], \quad i = 1,2,3 \tag{16}$$

In case of an emergency, the armature voltage has a bigger restriction zone ($\pm U_{imax}^{max}$) and therefore allows a higher armature voltage. This emergency feature is used in the collision avoidance later on.

Another limitation of the mechanical system is the armature current, due to the engineering property of a servo motor. Using a first order differential equation (eq. (17)), the electrical relationship of the armature is expressible with the servo motor constant k_{Ti}, gear ratio u_i, resistance R_i and inductance L_i:

$$\dot{I}_i = \frac{(U_{ai} - k_{Ti} u_i \dot{q}_i - R_{ai} I_{ai})}{L_{ai}}, \quad i = 1,2,3 \tag{17}$$

$$I_{ai} = [-I_i^{max} \leq I_i \leq I_i^{max}], \quad i = 1,2,3 \tag{18}$$

I_i^{max} (eq. (19)) can be computed with the maximum permitted torque T_i^{max} and the servo motor constant k_{Ti}:

$$I_i^{max} = T_i^{max} \left(\frac{\sqrt{3}}{2} k_{Ti}\right)^{-1}, \quad i = 1,2,3 \tag{19}$$

The armature current is proportional to the joint torques (T_1, T_2) and the joint force (T_3):

$$T_i = u_i \frac{\sqrt{3}}{2} k_{Ti} I_{ai}, \quad i = 1,2,3 \tag{20}$$

Obstacle definition & collision avoidance. Creating an obstacle, the tool-tip has to avoid, generates the collision avoidance task. The obstacle is defined as

$$h = h_{obs} \forall \ x \leq x_{obs} \tag{21}$$

$$h = 0 \forall \ x > x_{obs} \tag{22}$$

with $h_{obs} = 0.2m$ and $x_{obs} = 0.25m$ (23) (24)

This SCARA is equipped with a distance measuring sensor, which allows regular movement of the tool tip, if the distance between tool tip and obstacle is above the critical distance $d_{crit} > 0.1m$. In case of an undercut of the critical distance d_{crit}, the armature voltage can have a maximum of $\pm U_{imax}^{max}$ to decelerate faster and prevent a crash.

Additional Range Control. Due to the mechanical structure of the SCARA, we need to implement additional range control for the joint angles and the distance q_i, to achieve a model that pictures the real-life SCARA even better:

$$q_1 \leq \pi \tag{25}$$

$$|q_2| \leq \frac{5\pi}{6} \tag{26}$$

$$0.4m \geq q_3 \geq 0m \tag{27}$$

We only let our SCARA robot move its tool tip in a positive y-axis area (eq. (25)). That is why the limitation for the angle q_1 is defined as $q_1 \leq 180°$. Angle $|q_2|$ has a limit of under 180°, because the second rod is above the first one, which means that the tool-tip would crash into the first rod if $|q_2|$ would reach an angle close to 180° (eq. (28)). The length of the vertical rod q_3 also needs to be limited to ensure a realistic simulation (eq. (27)).

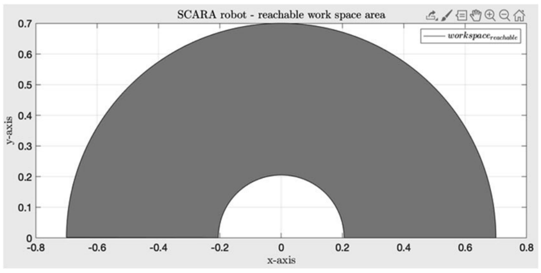


Figure 2: Reachable area of the tool-tip in the x-y-level.

1 MATLAB Implementation

Now we create a model of the SCARA movement using MATLAB. At first, we implement the given values and the equations of the definition, such as equation (3) – (15), (17), (19) and (20) – (27). These equations provide the needed values to implement the limitation of the current I_i from equation (17) and the limitation of the voltage U_i from equation (18). The implementation of the mass matrix (eq. (2)) differs, when we compare the explicit with the implicit model. Therefore, the mass matrix implementation gets described in chapter 1.1 and 1.2.

These mentioned implementations are straight forward and didn't cause any problems, but limitating the current I_i is not as simple as it may seem. By reason of the inability of the ODE-solver to limit the current itself, we limitated the voltage U_i and the derivative of the current $\dot{I}_i$. To carry out the limitation for the voltage U_i, we simply use ‚if' loops that constantly compare the current value of the voltage U_i with the maximum allowed voltage $\pm U_{ireg}^{max}$. In case of an exceeding, the value for U_i gets limited to the maximum allowed voltage $\pm U_i^{max}$.

The limitation of the current $\dot{I}_i$ is more difficult (shown in code-snippet 1). We constantly have to compare the current value of the current I_i with the maximum allowed current $\pm I_i^{max}$. If the current value I_i is higher than its maximum $+I_i^{max}$ and the derivative of the current $\dot{I}_i$ negative, the derivative of the current $\dot{I}_i$ is not limited because the negative inclination of the current I_i leads to a decrease of the current I_i. Consequently, the current I_i will drop below its maximum $+I_i^{max}$.

Vice versa if the current I_i is lower than its minimum $-I_i^{max}$ and the derivative of the current $\dot{I}_i$ is positive. If the current I_i is higher (lower) than its maximum $+I_i^{max}$ ($-I_i^{max}$) and the derivative of the current $\dot{I}_i$ positive (negative), the value of the derivative of the current $\dot{I}_i$ gets limited to zero.

```
if (q(7) > I1max) && ( ((U1-kt1*u1*q(1)-Ra1*q(7))/La1) < 0)
    b(7) = (U1-kt1*u1*q(1)-Ra1*q(7))/La1;
elseif (q(7) < -I1max) && ( ((U1-kt1*u1*q(1)-Ra1*q(7))/La1) > 0)
    b(7) = (U1-kt1*u1*q(1)-Ra1*q(7))/La1;
elseif (abs(q(7)) > I1max)
    b(7)=0;
else
    b(7) = (U1-kt1*u1*q(1)-Ra1*q(7))/La1;
end
```

Code-snippet 1: Limitation of the derivative of the current $\dot{I}_i$.

There are two possibilities to stop the calculation of the ODE-solver. The computing time can be limited to a fixed value, which is not very accurate when it comes to comparing different solving-times because there might be not enough time to reach the target position or the solver keeps on calculating, even though the SCARA is already very close to the target point or already reached it.

The second possibility uses an implementation of an event-function, that automatically stops the program, as soon as the tool tip reaches the target position. Therefore a so-called event-function "reached_target", which is shown in code-snippet 2, is used to constantly compare the current position of the the tool tip and the target position. With a difference of $q_{diff} \leq 0.001m$ between the actual and the target position of the tool tip, the event gets triggered and stops the calculation of the ODE-solver.

```
%event function: ODE-solver stops if target point is reached
function [value,isterminal,direction] = reached_target(~,qe)
global L1 L2 q_target q_diff
    value = (3 - (abs((cos(qe(4))*L1 + cos(qe(4)+qe(5))*L2)...
        - (cos(q_target(1))*L1 + cos(q_target(1)+q_target(2))*L2)) ...
        <= q_diff) - (abs((sin(qe(4))*L1 + sin(qe(4)+qe(5))*L2)...
        -(sin(q_target(1))*L1 + sin(q_target(1)+q_target(2))*L2))...
        <= q_diff) - (abs(qe(6)-q_target(3)) <= q_diff));
    isterminal = 1;         %stop the solver if value is zero
    direction = 0;
end
```

Code-snippet 2: Event that stops the code, as soon as the tool-tip reaches the target position.

```
%coordinates x, y and z of the target point
x_target = cos(q_target(1))*L1 + cos(q_target(1)+q_target(2))*L2;
y_target = sin(q_target(1))*L1 + sin(q_target(1)+q_target(2))*L2;
z_target = q_target(3);

%conditions for the target point
if q_target(1)>pi
    disp('The limit for q1 of the target point is pi!Try an other value.')
    return
elseif abs(q_target(2))>(5*pi/6)
    disp('The limit for |q2| of the target point is 5*pi/6! Try an other value.')
    return
elseif q_target(3)>0.4 || q_target(3)<0
    disp('The upper boundary for q3 is 40 cm, the lower one is the ground (0)! Try an other value.')
    return
elseif y_target<0
    disp('This SCARA-Robot can not reach behind itself! Try other values.')
    return
end
```

Code-snippet 3: Loop, to display a warning for unreachable target points.

With ‚if' loops, we can display a notification in case of an unreachable target position. The notification also gives a short description of the reason for the error, shown in code-snippet 3.

1.1 Excplicit Point-to-Point model

To solve the point-to-Point motion explicitly, we create a 9-row-vector $\dot{\vec{q}}$ that contains the joint velocities $\dot{q}_i$, the angles and the distance of the joints q_i and the currents of the servo motors I_i, written as vector $\dot{\vec{q}} = [\dot{q}_i, q_i, I_i]$. Additionally, the mass matrix M needs to be inverted and multiplied with $\vec{b}$ to receive the joint's angular and directional acceleration $\ddot{\vec{q}}$:

$$\ddot{\vec{q}} = \vec{b}M^{-1} \tag{28}$$

This step is necessary because the ODE-solver in the explicit model can't work with a mass matrix, provided by the $odeset()$ command. Extending vector $\vec{b}$ with $\dot{q}_i$ and the limited values of $\dot{I}_i$, we also get a 9-row-vector, which basically is the derived $\vec{q}$ vector.

In Figure 3 the changes of the angles and the heigth (q1, q2, q3) over time of the explicit point-to-point model are pictured. So, it can be observed that q1 and q2 have the value 2 and q3 the value 0.3 when the event-function stops the calculation. Furthermore, figure 4 visualises the movement of the tool tip in the x-y-z-system.

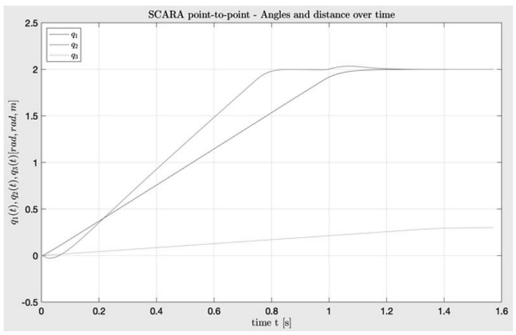


Figure 3: Plot of $q_1(t)$, $q_2(t)$ and $q_3(t)$, depending on t (explicit point-to-point model, target point q_{target} = [2 2 0.3], ode45).

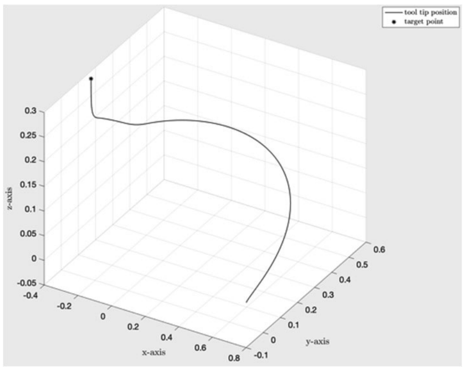


Figure 4: Plot of the movement of the tool tip (explicit point-to-point model, target point q_{target} = [2 2 0.3], ode45).

1.2 Implicit Point-to-Point model

Solving the point-to-point movement implicitly, we need to extend the original 3x3 mass matrix into a 9x9 mass matrix (eq. (29)), so the dimensions fit the right side of equation (1).

$$M = \begin{pmatrix} ma_{11} & ma_{12} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ ma_{21} & ma_{22} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & ma_{33} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{29}$$

The ODE-solver in the implicit model can work with a mass matrix, provided by the $odeset()$ command, which means that we don't have to invert our mass matrix M anymore. Hence, we provide the ODE-solver the maximum step size, the 9x9 mass matrix and the event for reaching the target by the $odeset()$ command. Limiting the current I_i and stopping the program happens in the same way as in the explicit model. The numerical accuracy is the same as in the explicit model. Hence, the graphical visualitaion is equal to Figure 3 and Figure 4 and not shown for the implicit model again.

1.3 Results of the Point-to-Point movement

We compare the ODE solvers ode45, ode23, ode23t, ode23s, ode23tb, ode113, and ode15s for the point-to-point movement, target point $q_{target} = [2 \;\; 2 \;\; 0.3]$:

ODE-solver	explicit solving-time	implicit solving-time
ode45	0.756971s	0.826150s
ode23	0.572606s	0.838872s
ode23t	0.270100s	0.485513s
ode23s	3.851616s	-
ode23tb	0.299691s	0.595774s
ode113	0.878708s	1.069086s
ode15s	0.244320s	0.358710s

Table 1: Explicit and implicit solving-times of different ODE-solvers for the point-to-point movement with target point $q_{target} = [2 \;\; 2 \;\; 0.3]$.

The measured explicit system is always faster than the implicit one. Another factor for the solving-time is the step size control of the solver itself. Through a high step size control, like in the ode23t, ode23tb and ode15s, the solving-time is very short. Thereupon, a solver like the ode45, which has a low step size control, has a longer solving-time. In addition, a correlation between the step size control and the accuracy of a solver exists. For example, ode23t, ode23tb and ode15s are fast but also have a low accuracy. Besides, the ode45 has a medium accuracy, due to its low step size control.

1.4 Explicit collision avoidance model

The explicit collision avoidance works in the same way as the explicit point-to-point movement, using the inverted mass matrix M. But to implement the collision avoidance, we extend the explicit point-to-point model with various cases which are automatically selected, regarding the dimensions of the obstacle and the target position.

An ‚if' loop, which is partly shown in code-snippet 4, chooses the right sequence of event functions. In this code snippet 4 case no. 1, 2 and the beginning of case no. 3 are pictured. The selected case depends on where the target point is located. If an event function detects a zero transition it stops the current ODE-solver and another one, with a new dynamical structure, has to be started using the last entries of the stopped ODE-solver as starting values for the new one. All times and values of the different used ODEs are attached in the vector te and vector qe to get a gapless simulation.

We differ now from case no. 1 to case no. 5. The differences are described in the following.

Case no. 1. In case no.1 (Figure 5), the target position is located **behind the obstacle** and **above the obstacle's height**. It starts the regular point-to-point solver until the tool-tip reaches the critical distance $d_{crit} = 0.1m$. After the event of the critical distance between the obstacle and the tool tip occurs, a new solver is called which only allows movement of q_3 (shown in code-snippet 5). This means that the SCARA drives vertically upwards, until the tool-tip is higher than the height of the obstacle.
Then, the standard point-to-point solver is called again to reach the target position.

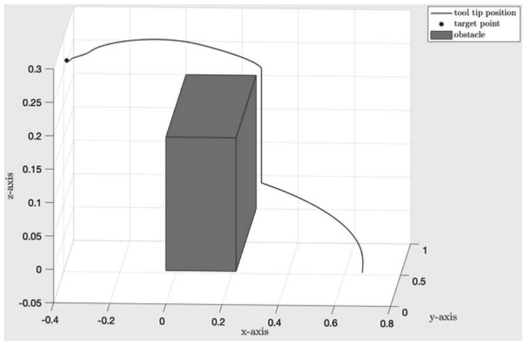


Figure 5: Plot of case no. 1 (explicit collision avoidance model, target point $q_{target} = [2 \;\; 2 \;\; 0.3]$, ode45).

```
%there are five different cases depending on the coordinates of the target point
if x_target<(-0.01) && z_target>(hobs+0.01) %case no.1
    %target point is located behind the obstacle and is higher than hobs
    tic
    [te, qe, ie] = ode45(@(t,q) dynamics(t,q), t_span, q_0, options1);
    %there is an event (avoidance1) -> start a new solver
    %begin with the last values of the old solver
    [te2, qe2, ie2] = ode45(@(t,q) dynamics2(t,q), [te(end) 2],...
        [qe(end, 1)*0 qe(end, 2)*0 qe(end, 3) qe(end,4)...
        qe(end,5) qe(end,6) qe(end,7) qe(end,8) qe(end,9)], options2);
    %interim results
    te = vertcat(te, te2);   %solutionvector is the solution of
    qe = vertcat(qe, qe2);   %the first solver and the second solver
    %there is an event (avoidance2) -> start a new solver
    %begin with the last values of the old solver
    [te3, qe3, ie3] = ode45(@(t,q) dynamics(t,q), [te(end) t_end],...
        [qe(end,1) qe(end,2) qe(end,3) qe(end,4) qe(end,5)...
        qe(end,6) qe(end,7) qe(end,8) qe(end,9)],options4);
    %solver stops if the target point is reached (event: reached_target1)
    toc
    %results te and qe: final solutionvectors
    te = vertcat(te, te3);
    qe = vertcat(qe, qe3);
elseif x_target>(-0.05) && z_target<(hobs+0.01) && x_target<=(xobs+0.01) %case no.2
    %target point is located inside or in a critical distance to the obstacle
    disp('Target point cannot be reached! Try other values.')
    return
elseif x_target<=(xobs+0.01) && z_target>=(hobs+0.01) && x_target>=(-0.01) %case no.3
    %target point is located right over the obstacle
    tic
    [te, qe, ie] = ode45(@(t,q) dynamics(t,q), t_span, q_0, options1);
    %there is an event (avoidance1) -> start a new solver
    %begin with the last values of the old solver
    [te2, qe2, ie2] = ode45(@(t,q) dynamics2(t,q), [te(end) 2],...
        [qe(end, 1)*0 qe(end, 2)*0 qe(end, 3) qe(end,4)...
        qe(end,5) qe(end,6) qe(end,7) qe(end,8) qe(end,9)], options2);
    %interim results
```

Code-snippet 4: Case no.1, 2 and partly 3, chosen depending on the position of the target point.

```
%first 3 equations have to be multiplied by the inverted mass matrix
a(1) = 0;
a(2) = 0;
a(3) = T3i-m3L*g;

%multiply left side with the inverted mass matrix to get explicit model
b=a/M;
```

Code-snippet 5: Limiting the movement of the SCARA robot to only joint 3.

Case no. 2. In case no. 2, the target position lies **within the obstacle** or **its critical surrounding**. A notification of the error gets displayed.

Case no. 3. In case no. 3, the target position is located **straightly above the obstacle's height**. The utilized procedure is the same as in case no. 1.

Case no. 4. In case no. 4 (fig. 6-8), the target position is located **behind the obstacle** and **below the obstacle's height**. Here, the solver for the point-to-point movement is used until the collision warning event gets triggered, followed by the vertical upward movement of the tool-tip as in case no. 1. As soon as the tool tip is higher than the obstacles height, another solver is used, which only allows movement in joint 1 and 2. This leads to a movement of the SCARA parallel to the obstacles top side. After the tool tip passes the obstacle, the solver for the point-to-point movement is used again to finally reach the target point.

Case no. 5. In case no. 5, the target position is located **in front of the obstacle**. No collision avoidance occurs, which makes this case to a simple point-to-point movement; events are used to stop the program when the tool tip reaches the target position.

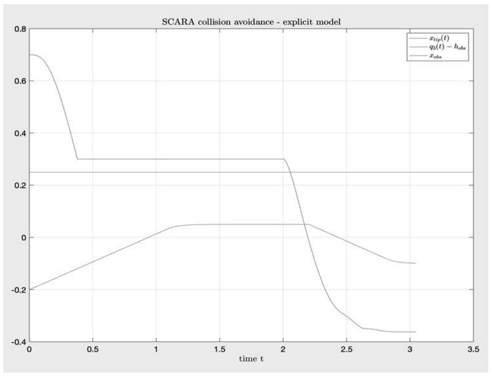


Figure 6: Plot of $x_{tip}(t)$, $q_3(t) - h_{obs}$ and x_{obs} over time t (case no. 4, explicit collision avoidance model, target point $q_{target} = [2\ \ 2\ \ 0.1]$, ode45).

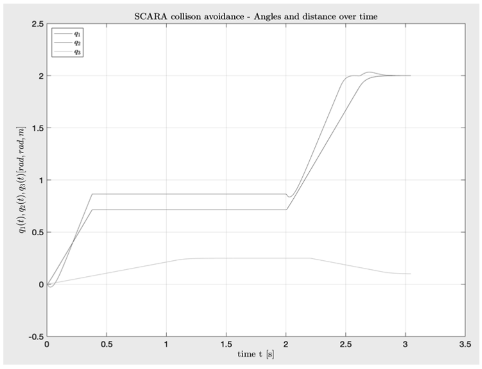


Figure 7: Plot of $q_1(t)$, $q_2(t)$ and $q_3(t)$, depending on t (case no. 4, explicit collision avoidance model, target point $q_{target} = [2\ \ 2\ \ 0.1]$, ode45).

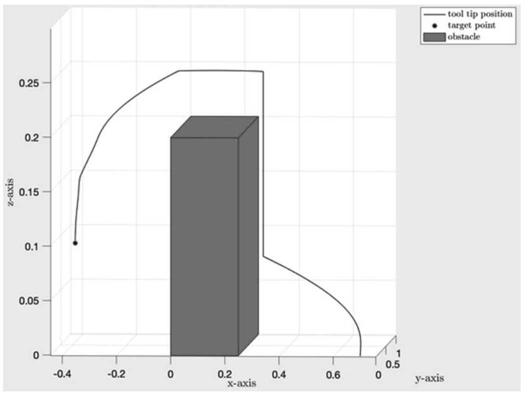


Figure 8: Plot of the case no. 4 (explicit collision avoidance model, target point $q_{target} = [2\ \ 2\ \ 0.1]$, ode45).

1.5 Implicit collision avoidance model

The implicit collision avoidance model uses the same structure as the explicit collision avoidance model, apart from the handling of the mass matrix. Here, the handling of the mass matrix is the same as in 1.2 Implicit Point-to-Point model. The solutions of the implicit model in case no. 1 are plotted in Figure 9 and Figure 10.

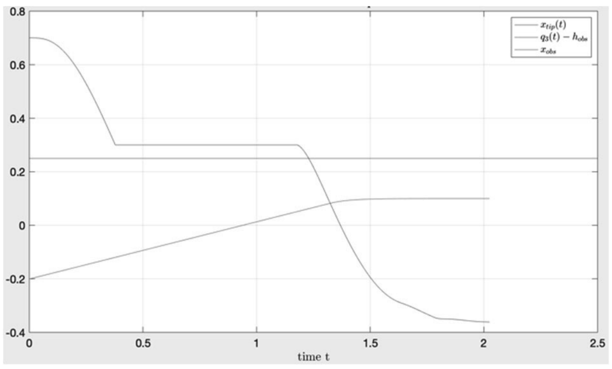


Figure 9: Plot of $x_{tip}(t)$, $q_3(t) - h_{obs}$ and x_{obs} over time t (case no. 1, implicit collision avoidance model, target point q_{target} = [2 2 0.3], ode45).

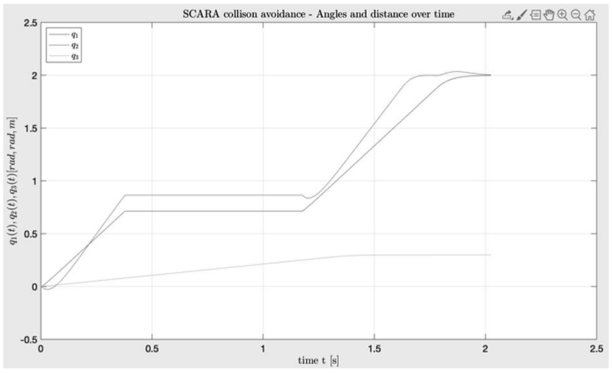


Figure 10: Plot of $q_1(t)$, $q_2(t)$ and $q_3(t)$, depending on t (case no. 1, implicit collision avoidance model, target point q_{target} = [2 2 0.3], ode45).

1.6 Results collision avoidance movement

We compare the following ODE-solvers for the collision avoidance movement with the target point q_{target} = [2 2 0.3]: ode45, ode23, ode23t, ode23s, ode23tb, ode113 and ode15s.

ODE-solver	explicit solving-time	implicit solving-time
ode45	1.163843s	1.316385s
ode23	0.820086s	0.988098s
ode23t	0.458257s	0.581426s
ode23s	3.177899s	-
ode23tb	0.427696s	0.621490s
ode113	1.240606s	1.259776s
ode15s	0.422060s	0.422124s

Table 2: Explicit and implicit solving-times of different ODE-solvers for the collision avoidance movement with target point q_{target} = [2 2 0.3].

2 Simulink Implementation

In Simulink, a model for simulating the Point-to-Point movement in an explicit representation and two different models for simulating the Point-to-Point movement with collision avoidance also in an explicit representation were developed. On the one hand a collision avoidance model with the usual Simulink-blocks and a 'Switch'-block to switch between the Point-to-Point movement and the collision avoidance movement and on the other hand a Simulink solution with integrated 'Stateflow'-states, in which it is also possible to switch between these movements.

All constant parameters necessary for the simulation were given into Simulink over the MATLAB environment. The simulation results were transferred back into MATLAB through 'To Workspace'-blocks for plotting purposes. To solve the explicit Simulink models, the variable step size, one step solver ode45 was used, whereby the relative tolerance was reduced to 1e-6. Since a double integration is possible in Simulink by implementing two 'Integrator'-blocks connected directly one after the other, it is not necessary to transform the descriptive second order differential equations [2]. The resulting 3x3 mass matrix was implemented in the Simulink and Simulink / Stateflow models using a 'MATLAB Function'-block and then inverted using a 'Product'-block in inversion mode.

As in the MATLAB implementation in order to prematurely end the integration at the target point, a Simulation Abort Condition-subsystem was integrated in all models (see Figure 11). When a precession of 0.001 is reached, the simulation stops by means of a 'Stop'-block.

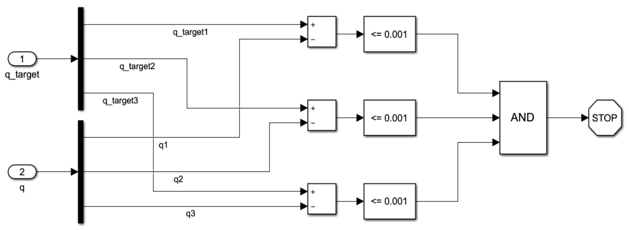


Figure 11: Simulation Abort Condition-subsystem with a logical combination of the differences between the components of the target point location vector q_{target} and the robot arm location vector q, with the 'Stop'-block being activated when a precession of 0.001 is reached.

2.1 Explicit Point-to-Point model in Simulink

Figure 12 shows the overall Point-to-Point Simulink model with built-in Control Model-subsystem (shown in Figure 13) and a 'MATLAB Function'-block called dynamics (which contains the equations 12 to 14).

The Control Model-subsystem includes a PD control-subsystem, which defines the relationship between the distance to the target point and the necessary voltage (see equation 15), as well as a servo motor-subsystem (shown in figure 14), which contains the first order differential equation for determining the current (see equation 17). The resulting current is limited via a limited output of the 'Integrator'-block. The current limit values are calculated using equation 19. Finally, the associated torque is determined in a 'MATLAB Function'-block. The torque is required to calculate the right-hand side of the system of differential equations (see equations 12 to 14).

The voltage is limited via a 'Saturation'-block, which is included in the PD control-subsystem. The voltage limit values, like the current limit values, are given constant parameters. The voltage and current limitation are much easier to implement in Simulink than in MATLAB.

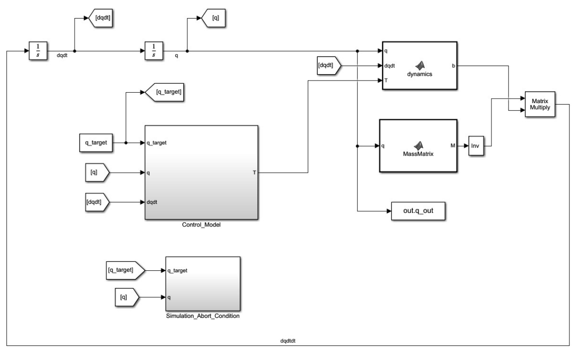


Figure 12: Explicit Point-to-Point Simulink model.

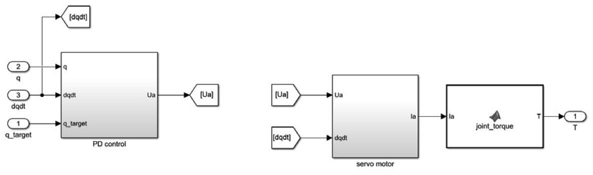


Figure 13: Control Model-subsystem.

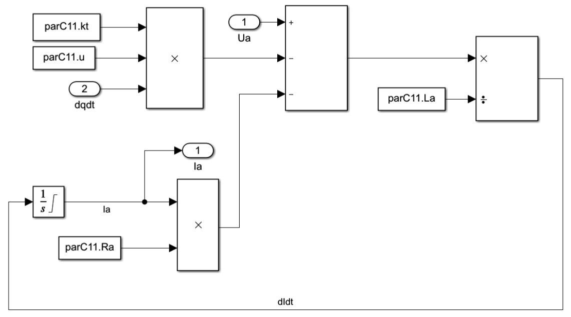


Figure 14: Servo motor-subsystem including an 'Integrator'-block with limited output.

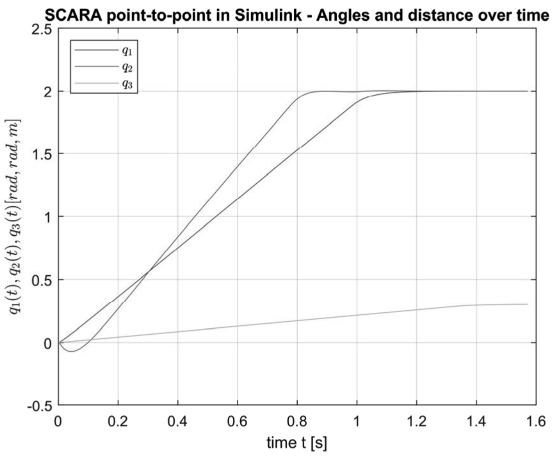


Figure 15: Point-to-Point simulation in Simulink - time course of q_1, q_2 and q_3.

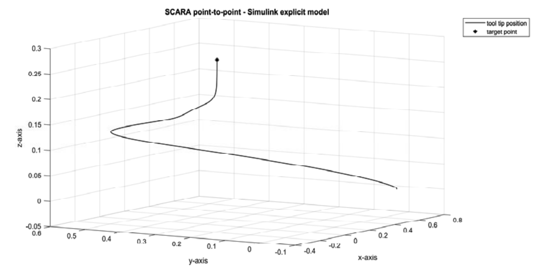


Figure 16: Point-to-Point simulation in Simulink - tool-tip path in three-dimensional space.

Results. Figure 15 shows the time course of the two angles q_1 and q_2 as well as the distance q_3 during the Point-to-Point movement (simulated in Simulink). Figure 16 shows the result of the Point-to-Point simulation in Simulink in three-dimensional space with initial value $q = [0; 0; 0]$ and target point $\hat{q} = [2; 2; 0.3]$. The Simulink Point-to-Point simulation runtime was 0.143362 seconds.

2.2 Collision avoidance model in Simulink and Simulink / Stateflow

Figure 17 shows the overall Point-to-Point with collision avoidance Simulink model with built-in Extended Control Model-subsystem. In order to avoid a collision with a defined object, an additional subsystem was included into the Control Model-subsystem (see Figure 18).
In this additional subsystem two 'Switch'-blocks for switching between Point-to-Point movement and obstacle avoidance are implemented.

If the condition $(x_{tip} - x_{obs}) \leq d_{crit} \wedge q_3 < h_{obs}$ is fulfilled, $\dot{q}$ is changed in such a way that $\dot{q}_{1,obs} = 0.00005 \cdot \dot{q}_1$, $\dot{q}_{2,obs} = 0.00005 \cdot \dot{q}_2$ and $\dot{q}_{3,obs} = \dot{q}_3$. $\dot{q}_{obs}$ is then passed on to the integrator and the dynamics-block (see Figure 17).

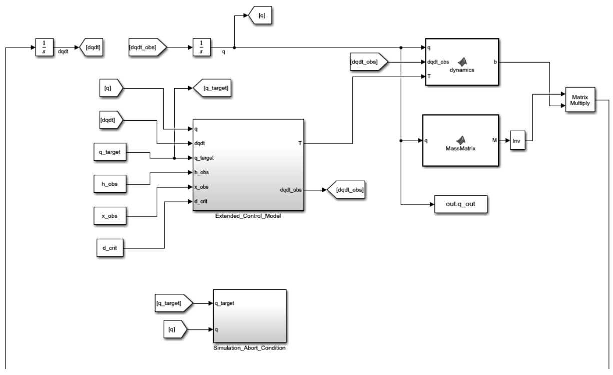


Figure 17: Explicit Point-to-Point with collision avoidance Simulink model.

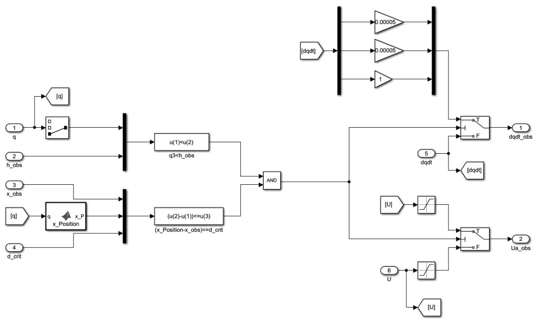


Figure 18: Additional subsystem with two 'Switch'-blocks for switching between different robot arm speeds $\dot{q}$ and maximum voltages.

In the second 'Switch'-block, which is controlled via the same condition, the permissible voltage limits are specified. In the case of obstacle avoidance, the normal maximum voltages (see chapter 2.1) are changed to the emergency maximum voltages.

Figure 19 shows the overall Point-to-Point with collision avoidance Simulink model with built-in 'Stateflow-Chart'-subsystem.

When using the Stateflow environment, the task is to define certain states and to specify conditions for when to switch between these states. Point-to-Point movement and collision avoidance were defined as states (see Figure 20). To define the states, 'Simulink state'-blocks were used, in which Simulink models like the one shown in Figure 12 with different $\dot{q}$ and maximum voltages were installed (see Figure 21 and Figure 22). 'State Writer'-blocks were used to transfer the integrated parameters from one state to another. The condition for a change from Point-to-Point movement to obstacle avoidance has not changed compared to the pure Simulink solution. In the opposite direction, the condition that the tool-tip has exceeded the obstacle height is sufficient. The definition of the conditions for changing between the states is much easier in Simulink / Stateflow than in a pure Simulink solution.

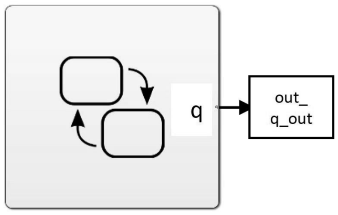


Figure 19: Explicit PtP with collision avoidance - Simulink model with 'Stateflow-Chart'-susystem.

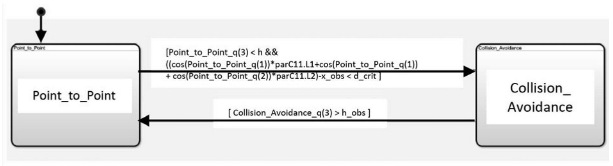

Point_to_Point
[Point_to_Point_q(3) < h &&
((cos(Point_to_Point_q(1))*parC11.L1+cos(Point_to_Point_q(1))
+ cos(Point_to_Point_q(2))*parC11.L2)-x_obs < d_crit]
[Collision_Avoidance_q(3) > h_obs]
Collision_
Avoidance

Figure 20: 'Stateflow-Chart'-subsystem with 'Simulink state'-blocks and conditions for a change of state.

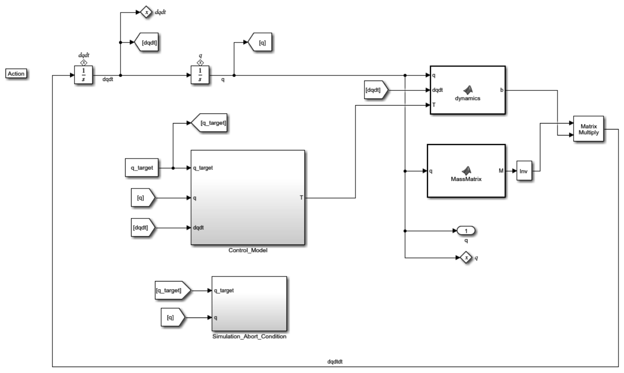


Figure 21: Point-to-Point subsystem.

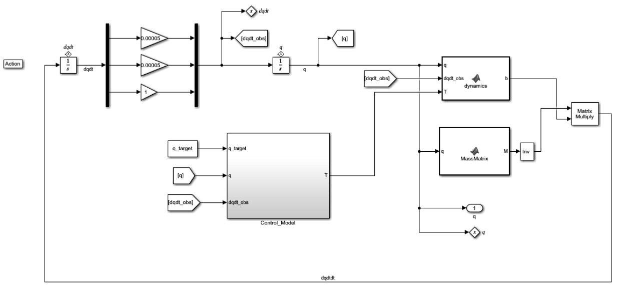


Figure 22: Collision avoidance subsystem.

Results. Figure 23 shows the time course of the two angles q_1 and q_2 as well as the distance q_3 during the Point-to-Point with collision avoidance movement (Simulink).

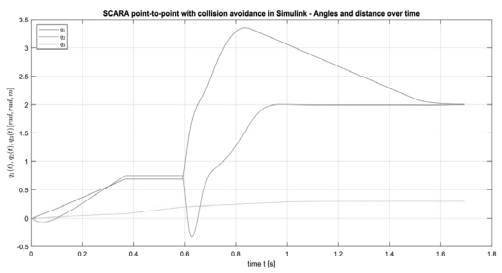


Figure 23: Point-to-Point with collision avoidance simulation in Simulink - time course of q_1, q_2 and q_3.

Figure 24 shows the result of the Point-to-Point with collision avoidance simulation in Simulink in three-dimensional space with initial value $q = [0; 0; 0]$, target point $\hat{q} = [2; 2; 0.3]$, obstacle height $h_{obs} = 0.2\,m$, obstacle distance $x_{obs} = 0.25\,m$ and critical distance $d_{crit} = 0.1\,m$.

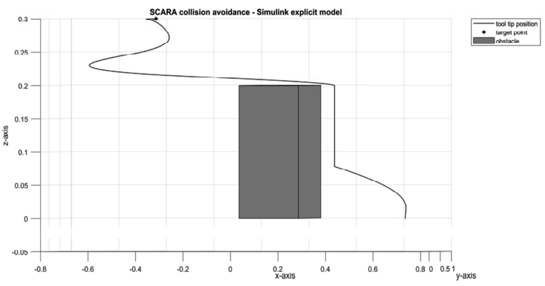


Figure 24: Point-to-Point with collision avoidance simulation in Simulink - tool-tip path in 3D space.

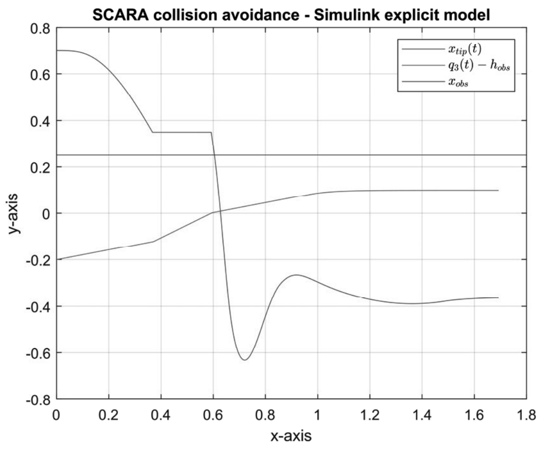


Figure 25: Representation of the Simulink collision avoidance maneuver in two-dimensional space.

The Simulink Point-to-Point with collision avoidance simulation runtime was 0.239948 seconds.

Figure 26 shows the time course of the two angles q_1 and q_2 as well as the distance q_3 during the Point-to-Point with collision avoidance movement (simulated in Simulink / Stateflow).

Figure 27 shows the result of the Point-to-Point with collision avoidance simulation in Simulink / Stateflow in three-dimensional space with initial value $q = [0; 0; 0]$, target point $\hat{q} = [2; 2; 0.3]$, obstacle height $h_{obs} = 0.2\,m$, obstacle distance $x_{obs} = 0.25\,m$ and critical distance $d_{crit} = 0.1\,m$.

The Simulink / Stateflow Point-to-Point with collision avoidance simulation runtime was 0.315776 seconds.

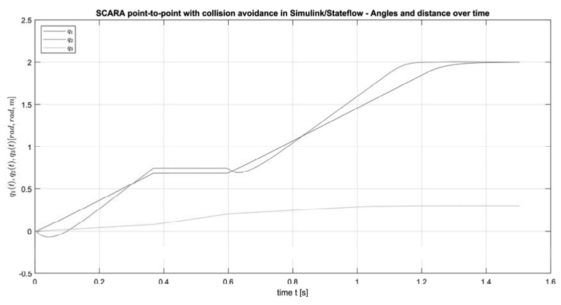


Figure 26: Point-to-Point with collision avoidance simulation in Simulink / Stateflow - time course of q_1, q_2 and q_3

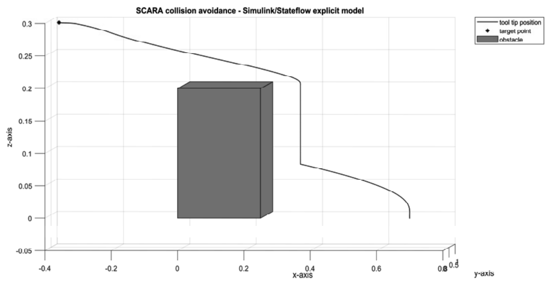


Figure 27: Point-to-Point with collision avoidance simulation in Simulink / Stateflow - tool-tip path in three-dimensional space.

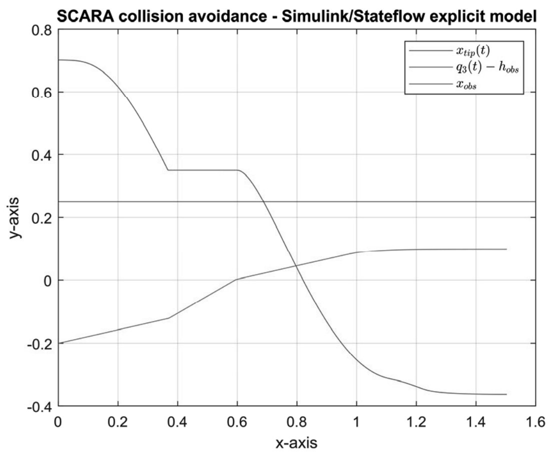


Figure 28: Representation of the Simulink / Stateflow collision avoidance maneuver in 2D space.

3 Comparison & Results

A very essential difference between the MATLAB and the Simulink implementation is the computing time for each model. The direct comparison of the explicit point-to-point models shows, that the Simulink implementation is in our elaboration ≈ 5 times faster than the MATLAB implementation (both solving times result in the use of the ode45). This observation is confirmed, when we com-

pare the computing times of the explicit collision avoidance models (both solving times result in the use of the ode45). The Simulink simulation times are significantly shorter than those in MATLAB, since, in contrast to MATLAB, the right side of the differential equation system is mapped using a function and the code is not processed in lines.
On the other hand, using ‘MATLAB-function’-blocks in Simulink delays the simulation time.

Another important difference between these two implementation possibilities is the ability of being able to work with a second order differential equation system. MATLAB can’t work with a second order differential eqation system, in contrast Simulink can. This makes the Simulink implementation easier because the C11 definition needs no further conversion.

When it comes to the limitation of the voltage and current, the Simulink implementation is a lot easier and faster, compared to the MATLAB implementation. As shown in chapter 2.1, the limitation is done in Simulink with very few steps, while the limitation in MATLAB needs a small expansion, shown in chapter 1 and code-snippet 1.

Addendum

Additionally, an animation of the point-to-point movement and the collision avoidance movement is realised in the Matlab code as shown in Figure 29. Only with this animation we got a real feeling how the three rods were moving, and which relative movements are necessary to guid the tool tip of the SCARA to its target point. Furthermore, through the animation we recognized a long waiting time between the vertical and the horizontal movement when the obstacle height was reached so we adapted the setting of the PD control in order to reach the target even faster.

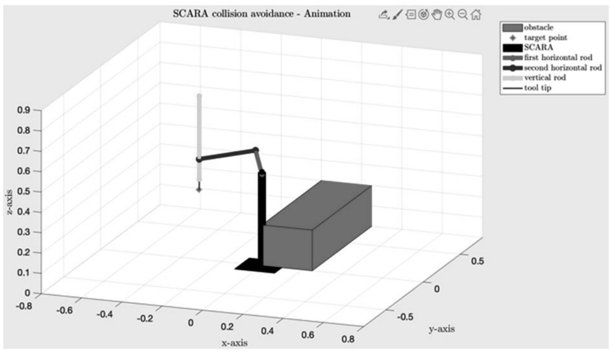
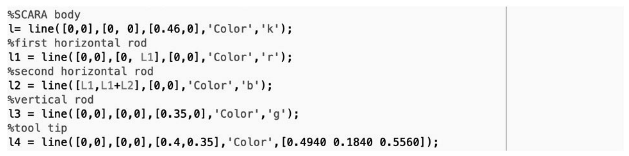


Figure 29: Animation of the collision avoidance movement of the SCARA in a 3D-model.

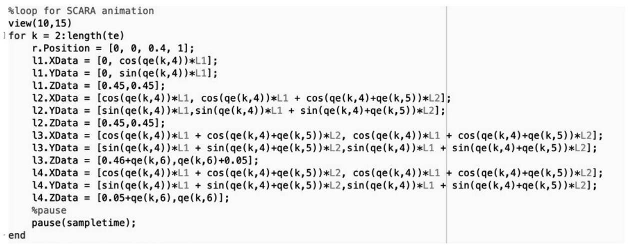
The animation was implemented with an explicit and with an implicit model, but there was no significant difference. So, further explanations refer to an implicit animation model. For this task the already described code of an implicit collision avoidance was used for this animation. The changes were made in the plot section.
First, the SCARA appearance itself had to be modeled. This was realised with different lines for the SCARA body, the rods and the tool tip as shown in the code-snippet 6.



```
%SCARA body
l= line([0,0],[0, 0],[0.46,0],'Color','k');
%first horizontal rod
l1 = line([0,0],[0, L1],[0,0],'Color','r');
%second horizontal rod
l2 = line([L1,L1+L2],[0,0],'Color','b');
%vertical rod
l3 = line([0,0],[0,0],[0.35,0],'Color','g');
%tool tip
l4 = line([0,0],[0,0],[0.4,0.35],'Color',[0.4940 0.1840 0.5560]);
```

Code-snippet 6: Shaping the appearance of the SCARA with different lines and their spatial position.

Finally, the before calculated values of q_i are now used to describe the position of the three rods in code-snippet 7. A ‚for‘ loop is used to plot every calculated position of the rods on their way to the target point with a short pause between the plots to get an animation of the motion of the SCARA.



```
%loop for SCARA animation
view(10,15)
for k = 2:length(te)
    r.Position = [0, 0, 0.4, 1];
    l1.XData = [0, cos(qe(k,4))*L1];
    l1.YData = [0, sin(qe(k,4))*L1];
    l1.ZData = [0.45,0.45];
    l2.XData = [cos(qe(k,4))*L1, cos(qe(k,4))*L1 + cos(qe(k,4)+qe(k,5))*L2];
    l2.YData = [sin(qe(k,4))*L1,sin(qe(k,4))*L1 + sin(qe(k,4)+qe(k,5))*L2];
    l2.ZData = [0.45,0.45];
    l3.XData = [cos(qe(k,4))*L1 + cos(qe(k,4)+qe(k,5))*L2, cos(qe(k,4))*L1 + cos(qe(k,4)+qe(k,5))*L2];
    l3.YData = [sin(qe(k,4))*L1 + sin(qe(k,4)+qe(k,5))*L2,sin(qe(k,4))*L1 + sin(qe(k,4)+qe(k,5))*L2];
    l3.ZData = [0.46+qe(k,6),qe(k,6)+0.05];
    l4.XData = [cos(qe(k,4))*L1 + cos(qe(k,4)+qe(k,5))*L2, cos(qe(k,4))*L1 + cos(qe(k,4)+qe(k,5))*L2];
    l4.YData = [sin(qe(k,4))*L1 + sin(qe(k,4)+qe(k,5))*L2,sin(qe(k,4))*L1 + sin(qe(k,4)+qe(k,5))*L2];
    l4.ZData = [0.05+qe(k,6),qe(k,6)];
    %pause
    pause(sampletime);
end
```

Code-snippet 7: Loop to plot the present position of the SCARA rods at every entry of q_i with a short pause between every plot to get an animation.

Information

MATLAB R2019b Update 4 (9.7.0.1296695) and MATLAB R2020a (9.8.0.1323502) was used on macOS Catalina 10.15.7 and Windows 10.0.18363.1139 to develop this elaboration.

References

[1] Horst Ecker. Comparison 11 (SCARA robot) - Definition. *EUROSIM - Simulation News Europe*. 1998; 22: 30-32

[2] Mathworks: Products and Services. https://de.mathworks.com (last access on 29th October 2020)