ARGESIM Benchmark C11 'SCARA Robot' with Extended Trajectory Tracking Control: Comparison of Model Approaches and Simulation Results in MATLAB, Simulink and SimMechanics

Martin Batliner[2], Felix Breitenecker[1], Andreas Körner[1*], Horst Ecker[2]

[1]Inst. of Analysis and Scientific Computing, [2]Inst. of Mechanics and Mechatronics, TU Wien, Wiedner Hauptstrasse 8-10, 1040 Vienna, Austria; **Andreas.Koerner@tuwien.ac.at*

SNE 31(1), 2021, 43 - 51, DOI: 10.11128/sne.31.bne11.10557
Received: October 25, 2018; Revised: July 15, 2019;
Revised: December 15, 2020; Accepted: January 20, 2021
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. Modelling of mechatronic systems often results in implicit model description. Simulation systems provide different strategies to deal with this kind of problem, resulting in different modelling approaches. For this purpose, Mathwork's MATLAB system offers in its basic MATLAB system at programming level implicit ODE solvers, and in its toolboxes Simulink and SimMechanics (on basis of the Modelica-like system SimScape) graphical modelling environment. This Benchmark Study compares these modelling approaches and the simulation efficiency and results on basis of the ARGESIM Benchmark C11 'SCARA Robot'. Additionally, the contribution's investigations present a tuning of the PID control for point-to-point movement with and without collision prevention and introduce a trajectory tracking control with collision prevention, which improves the performance essentially.

Introduction

Classical derivation of models for mechanic systems results first in implicit models for the generalized coordinates $\vec{q}$, with mass matrix $M(\vec{q},\dot{\vec{q}})$, and with generalized forces $\vec{f}(\vec{q},\dot{\vec{q}},\vec{u})$ with input (feedforward control $\vec{u}$):

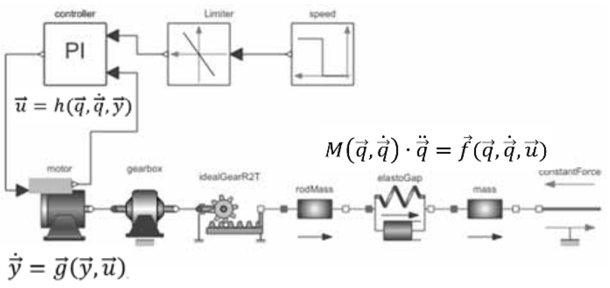
$$M(\vec{q},\dot{\vec{q}})\cdot\ddot{\vec{q}} = \vec{f}(\vec{q},\dot{\vec{q}},\vec{u})$$

Second, feedback control with actuators adds (state) equations for the actuators and feedback functions for control: $\dot{\vec{y}} = \vec{g}(\vec{y},\vec{u}),\ \ \vec{u} = h(\vec{q},\dot{\vec{q}},\vec{y})$

And third, these implicit equations have to be transformed into at least semi-linear state space form with 'combined' state vector $\vec{x}$, to be understood by an ODE solver of a 'basic' simulation system:

$$\widehat{M}(\vec{x},\vec{u})\cdot\dot{x} = \vec{\hat{f}}(\vec{x},\vec{u}),\ \ \vec{x} = (\vec{q},\dot{\vec{q}},\vec{u})^T$$

On the other side, today simulation environments offer graphical physical modelling with mechanical elements for the mechanical part following the Modelica concept [1] and graphical signal-oriented block modelling for the control part, as sketched in the following figure:


controller
PI
Limiter
speed
$\vec{u} = h(\vec{q},\dot{\vec{q}},\vec{y})$
$M(\vec{q},\dot{\vec{q}})\cdot\ddot{\vec{q}} = \vec{f}(\vec{q},\dot{\vec{q}},\vec{u})$
motor
gearbox
idealGearR2T
rodMass
elastoGap
mass
constantForce
$\dot{\vec{y}} = \vec{g}(\vec{y},\vec{u})$

1 Mechatronics in MATLAB

MATLAB [7] offers modelling and simulation on all these above cited levels, from basic use of ODE solvers for the semi-linear state space form in basic *MATLAB* directly, via signal-oriented graphical modelling for the control part as well as for the mechanical part in *Simulink*, up until to use of *SimMechanics* for the mechanicals in combination with *Simulink* for the control part.

The user clearly expects that all approaches give the same simulation results. But as 'higher' modelling techniques generate equations 'automatically' (a big comfort for the user), the models may differ in comparison with a 'laborious manually' model setup. Consequently, sub-results may differ, especially if discrete elements and/or boundaries are implemented into the overall system.

The following investigations discuss alle three *MATLAB* modelling techniques in a case study with a specific SCARA robot and its environment, as defined in *ARGESIM Benchmark C11* ‘SCARA Robot’, dealing with mechanics, control, and collision prevention for such type of robot [2]. Additionally, control is extended from point-to-point control to trajectory tracking control, increasing the complexity of both control and mechanical model.

2 SCARA Robot Benchmark

2.1 System Definition

Following the definition of the *ARGESIM Benchmark C11* ‘SCARA Robot’, the three axis SCARA robot type as shown in Figure 1 has two vertical revolute joints and one vertical prismatic joint. The axes of all three joints are parallel to the z-axis.

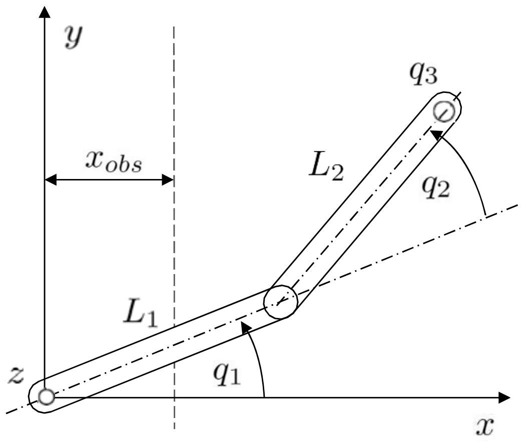


Figure 1: Kinematic structure of a SCARA robot.

The joint vector $\vec{q}$ consists of the joint angles q_1 and q_2 and the joint distance q_3. The equations of motion are given in the form

$$M(\vec{q}) \cdot \ddot{\vec{q}} = \vec{b}(\vec{q}, \dot{\vec{q}})$$

The calculation of the moments of inertia is based on the assumption that the rods have a homogeneous mass distribution. The right hand side of the dynamic equation includes the joint torques and joint forces. The electric relationship of the armature of a robot servo motor is given by the first order differential equation

$$I = \frac{1}{L_{ai}} \cdot (U_{ai} - k_{Ti} \cdot u_i \cdot \dot{q}_i - R_{ai} \cdot I_{ai})$$

where U_{ai} is the applied armature voltage. The resulting armature current I_i is limited to maximum values I_{max}.

The joint torques (forces) of a motor are proportional to the armature current I_{ai} and given by

$$T_i = u_i \frac{\sqrt{3}}{2} k_{Ti} \cdot I_{ai}$$

The detailed equations and numerical data are taken from the definition *ARGESIM Benchmark* C11, see [2].

2.2 Implementation

First, the simulators and their respective solvers are shortly portrayed and their handling of implicit systems. Then the basic model is implemented with the three simulators Simulink, SimMechanics and in *MATLAB* with solver ode15i.

2.3 Point-to-point motion

In order to control the point-to-point motion of the robot a single-axis PD-controller is employed for the control voltage:

$$U_i = P_i \cdot (\hat{q}_i - q_i) - D_i \cdot \dot{q}_i$$

Initial values are $q_1 = q_2 = q_3 = 0$, and the target values are $q_1 = q_2 = 2$ and $q_3 = 0.3$. The velocities should be zero at start and end.

2.4 Collision avoidance

Based on the point-to-point motion now an obstacle has to be avoided. The obstacle is given by an elevated area with the height h_{obs} and the borderline x_{obs} (Figure 1). The border represents an obstacle for the end-effector of the robot arm. Possible contact has to be avoided and must be detected during robot motion.

Therefore, an obstacle sensor should measure the distance between end-effector and obstacle and if that falls below a critical value d_{crit} the controls of rotational drives must be changed until the tool tip has cleared the obstacle height. Maximum voltage may be used in this situation for motor 1 and motor 2 to obtain maximum deceleration.

2.5 Trajectory tracking control

Additionally to the tasks presented in *ARGESIM Benchmark C11*, we now introduce another task: a trajectory tracking control should be implemented to keep the end-effector on a desired trajectory. The robot is assumed to be in the same environment with the obstacle as before.

The foreseen trajectory should not only execute the point-to-point motion, but also bypass the obstacle providing a path which is planned from desired accelerations. Both a PD and a PID controlled are implemented and compared.

3 Implementation

3.1 Simulink

Simulink is a widely used tool for modelling and simulating multidomain dynamic systems. It is integrated in the *MATLAB* environment and can be scripted from it. *MATLAB* is used here to parameterise the model and analyse data.

Simulink provides as primary interface a graphical block diagram tool. The blocks used treating this problem can be found in the model library. The chosen ODE solver is the variable step size, one step solver ode45, which is based on the Dormand-Prince pair. It is based on an explicit Runge-Kutta formula and calculates the 4th and 5th order accurate solution and takes the difference to be the fourth order error to estimate the adaptive step size [8].

Implicit systems cannot be solved directly. Nevertheless, the implicit equation $M(\vec{q}) \cdot \ddot{\vec{q}} = \vec{b}(\vec{q}, \dot{\vec{q}})$ was implemented directly, but an algebraic loop break had to be added (Figure 2), which finds a solution for the system for every integration step. This is also possible in the presence of a hit-crossing block which is used in Section 5 and works similar to the zero-crossing in the event function discussed in Section 3.3 [3].

Additionally, the model was implemented in an explicit form: the mass matrix *M* was inverted manually using the symbolic computation in *MATLAB* and consequently the system $\ddot{\vec{q}} = M^{-1}(\vec{q}) \cdot \vec{b}(\vec{q}, \dot{\vec{q}})$ was set up in Simulink using the graphical block diagrams (Figure 3).

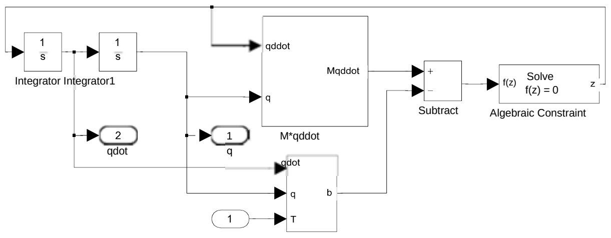


Figure 2: Implicit model description with *Simulink*.

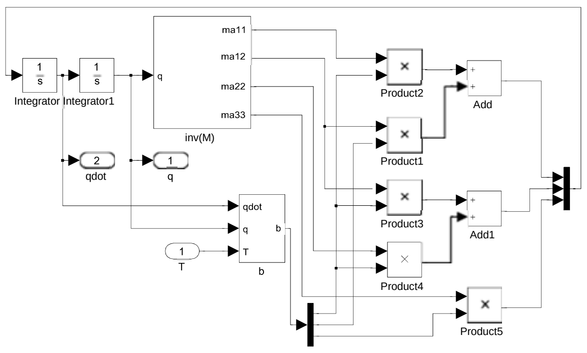


Figure 3: Explicit model description with *Simulink*.

3.2 SimMechanics

SimMechanics extends *Simulink* with tools for modelling three-dimensional mechanical systems within *Simulink* – being based on the general Modelica-like modelling system *SimScape*. Instead of deriving and programming equations this multibody simulation tool can build models composed of bodies, joints, constraints and force elements which reflect the structure of the system. It is integrated in *Simulink* and can be treated as a *Simulink* block, enabling physical model and control environment in one single system [6].

Consequently, simulating a SimMechanics model is a cooperative effort and consists of four steps:

- model validation
- machine initialisation
- force analysis and motion integration
- stiction mode iteration

whereby although the last step being negligible for the given problem as stiction is not treated.

The first two steps occur before the machine motion actually starts and checks all data entries, connections, assembly tolerances and validates the geometries and model topology. Moreover, *machine initialisation* cuts every closed loop and replaces it with a cut joint, constraint or driver block and checks all constraints and drivers for mutual consistency and eliminates redundant constraints. In *force analysis mode and motion integration*, *Simulink* steps up in simulation time and solves the system for every step while *SimMechanics* imposes assembly tolerances and a constraint solver [5]. Unfortunately, an analytic model cannot be derived from the *SimMechanics* block diagram.

The problem was implemented by setting up the block model in Figure 4. Rigid bodies are connected through joints which can be selected from the model library. The model properties such as mass, inertia and geometry can be set in the model blocks. The blocks, however, must be steered and measured with actuators and joint sensors which are connected to the free docks in Figure 4. These can be integrated into the Simulink workflow.

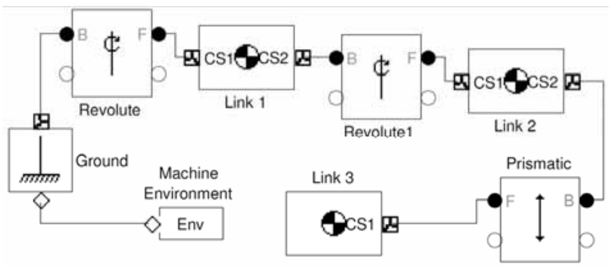


Figure 4: *SimMechanics* model.

3.3 MATLAB

For comparison a model in *MATLAB* was implemented. The solver ode15i in *MATLAB* can solve fully implicit differential and differential algebraic equations of the form $F(t, y(t), y'(t)) = 0$.
Additionally, to the initial values the solver needs the initial derivatives and the initial conditions must be consistent, meaning $F(t_0, y_0, y_0') = 0$. If needed they can be computed using the function decic in *MATLAB*.

The solver uses a fixed leading coefficient implementation of the BDFs (backward differentiation formulas) in a Lagrangian form for the polynomial interpolation [6]. This leads to simple expressions, evaluated efficiently in *MATLAB* and convenient for event localisation.

For the detection of state events the 'Events' property, in options set to function events, solves the system while also finding where user defined functions, called event functions, are zero. The event functions used for this problem are displayed in Section 5.2. At this point the system can come to a halt or is terminated. The event function is defined over a zero crossing, thus enabling the detection of the crossing through a change of sign [7].

Integrating the equations the solver passes the zero-crossing, consequently missing the exact value of the zero-crossing. But the solver automatically goes back and iterates back and forth to find the exact whereabouts of the zero-crossing within a set tolerance.

The system had to be reduced to a set of first order differential equations. Both sides of the equations have to be one as shown below. Hereby, *T* and *O* denote the joint torque (force) and the non-linear terms of the right-hand side of the equation:

```
y=[q(4)-qdot (1);…
q(5)-qdot(2);…
q(6)-qdot(3); …
ma11*qdot (4)+ma12*qdot (5)-T(1)-O(1);…
ma21*qdot (4)+ma22*qdot (5)-T(2)-O( 2 );...
ma33*qdot (6)-T(3)-O(3) ]
```

4 Point-to-point Motion

4.1 Simulink/SimMechanics

For the point-to-point movement (Figure 5) both *Simulink* and *SimMechanics* use the same controller, as the SimMechanics model is treated as a Simulink block. The model with controller and plant, which consists of the electrical and the mechanical model, is displayed in Figure 6. A target vector is the input for submodel control model which contains the PD-controller.

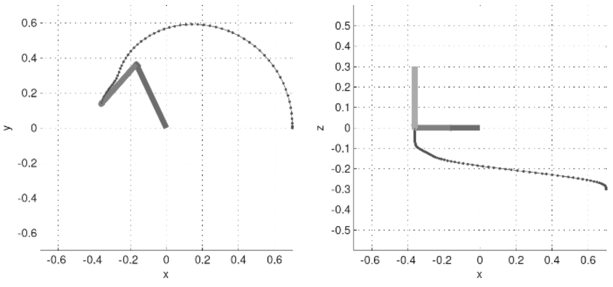


Figure 5: Point-to-point motion.

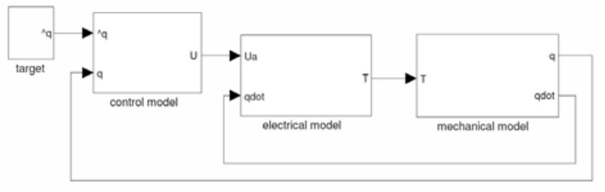


Figure 6: Control model in Simulink.

This was realised using the corresponding block from the Simulink library.

The output voltage *U* is fed into submodel electric model (Figure 6), which models the servo motor. The resulting torque *T* drives the mechanical system.

To incorporate the boundaries of voltages and currents and subsequently torque, *Simulink* offers the option to comfortably set the saturation limits as displayed in the electrical model in Figure 7 by a saturation block.

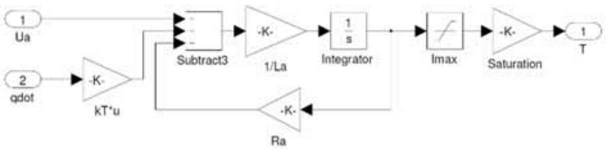


Figure 7: Electrical model.

4.2 MATLAB

For the point-to-point motion the model had to be extended by a control and servo motor model. These were integrated in the function deSCARA containing the robot equations. Moreover, armature current limits had to be implemented explicitly.

This was put into effect using simple if - clauses in the function containing the system equations. The function is called by

```
[ t q ] = ode15i (@deSCARA,…
        …[ t0 tend ], q0, qdot0 );
```

where the timespan, initial values and derivatives have to be implemented in consistent form.

5 Collision Avoidance

5.1 Simulink/SimMechanics

For collision avoidance (Figure 8) a block called state control was added to the submodel control model in Figure 9.

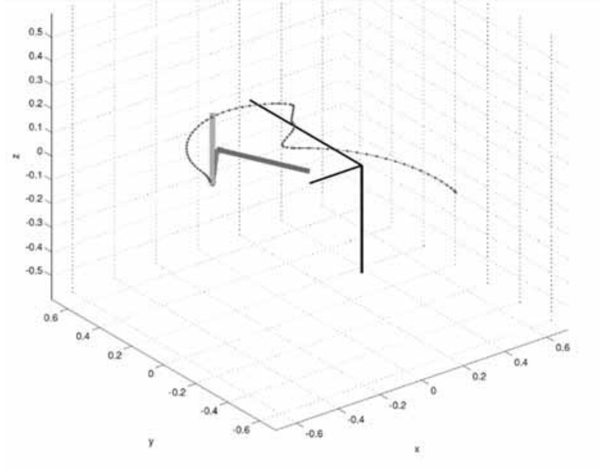


Figure 8: Collision avoidance.

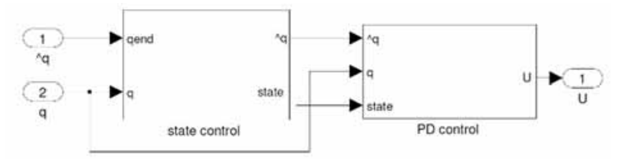


Figure 9: Extended control model.

The distance between the obstacle and the tool tip is permanently checked. If it falls under the critical distance the target positions for q_1 and q_2 are changed to the current position and the emergency maximum voltages are allowed. This is realised through a logic circuit which checks two conditions: One, is the robot in critical zone, and two, is the robot under h_{obs}. If both conditions apply new set-points for q_1 and q_2 are enforced. For robot arm 1 and arm 2 maximum voltage may be used to slow down and return to the position where the danger and subsequently a zero-crossing has been detected (as denoted in Section 3.1). Just after the tool tip of the robot has reached the height of the obstacle, at which only one of the two conditions applies, original target positions are reactivated for the arm 1 and arm 2.

How this is realised displays Figure 10 which shows in detail the state control from Figure 9. The block state control checks if one or both conditions mentioned before apply. If they do, they change their respective value from 0 to 1 and add up in block Add2. This sum now is the reference for the blocks threshold and Switch. Block threshold checks if this sum adds up to 2. If so, the switch forwards new set-points to the PD controller.

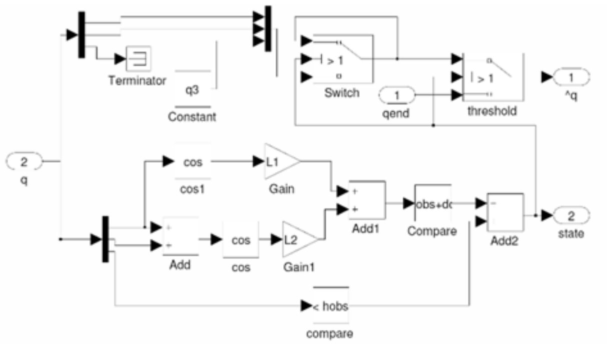


Figure 10: State control.

This point in time is determined accurately as the block enables zero-crossing detection.

The new set-points forwarded to the block threshold do not have to be specified by the user, but are determined through another circuit which localises the joint position of arm 1 and arm 2 when they enter critical zone. This is done by the block Switch. It has as input the current position and via a loop the last position. In critical zone, Switch forwards the looped values for q_1 and q_2 while looping the same value. If one of the two conditions is no longer fulfilled, the block threshold switches back to the original set-points.

For the saturation limits in the controller, the state of the system is forwarded trough the output state and the change of saturation limit in the PD-controller is subsequently realised with a switch block. This way the user is not required to specify way-points for the case that the robot enters the critical zone.

5.2 MATLAB

Additionally to the PD-controller in Section 4.2, collision avoidance has to be implemented employing the event property in the options of the solver shown in code below and implementing two event functions (cf. Section 3.3). The event functions for the given problem are:

$$x_{obs} + d_{crit} - L_1 \cdot \cos(q_1) - L_2 \cdot \cos(q_1 + q_2)$$
$$h_{obs} - q_3 = 0$$

Event specific parameters are returned from the function such as time and solution at a zero crossing and the corresponding type of event. This allows to stop the simulation at the occurrence of an event and restart the simulation with new setpoints.

Other than in the models implemented with *Simulink* and *SimMechanics,* these new set-points were added manually. Changing of set-point occurs twice: for entering critical zone and subsequent to reaching the admissible height, setting the setpoints back to the initial set-points.

The simulation runs until t_{end} when the counter equals 1 and the while-loop stops. Time and output vector are concatenated respectively every time the simulation is restarted:

```
options = odeset ( ' Events ' , @eventFun ) ;
while count==1
[ t q te xv ereig ]=ode15i ( @deSCARA,…
… [ t(end) tend ], q0, qdot0, options ) ;
qout=[ qout; q( : , 1 ) ];
tout=[ tout; t ];
if ( t( end)>=tend )
    count=0;
else
    telen=length( te );
    if ereig( telen )==1
        q1=1; q2=1;
    else ereig ( telen )==2;
        q1=2; q2=2;
    end;
end; end
```

6 Trajectory Tracking

6.1 Trajectory planning

The trajectory control problem in the joint space consists of following a given time-varying trajectory $q_d(t)$ and its successive derivatives which respectively describe the desired velocity and acceleration [9]. The planned path (Figure 11) will execute the point-to-point movement treated earlier with a way-point at $q_1 = q_2 = 1$ and $q_3 = 0.3$ to avoid collision with the obstacle described in the problem definition.

The velocities for start and end should be zero and naturally, the trajectories for the positions have to be smooth. For robot arm 1 two quadratic splines were chosen. These lead to linear velocities and constant accelerations. Robot arm 2 should have constant velocity, which leads to linear motion. To ensure smooth transition cubic splines were chosen for the path. Arm 3 has one cubic spline to the way-point and then stops. This results in linear acceleration. These variations in velocities and accelerations should unveil the capabilities and limits of the controller. For that purpose the *SimMechanics* model from the point-to-point motion is extended where the voltage restrictions still apply.

6.2 PD-Control

Based on PD-controller used earlier the tracking controller has the form

$$U_i = q_{i,d} + P \cdot (\ddot{q}_{i,d} - q_i) + D \cdot \dot{q}_i$$

where *d* marks the desired terms and the equations computes the control voltage from the position and velocity errors while feeding forward the desired acceleration.

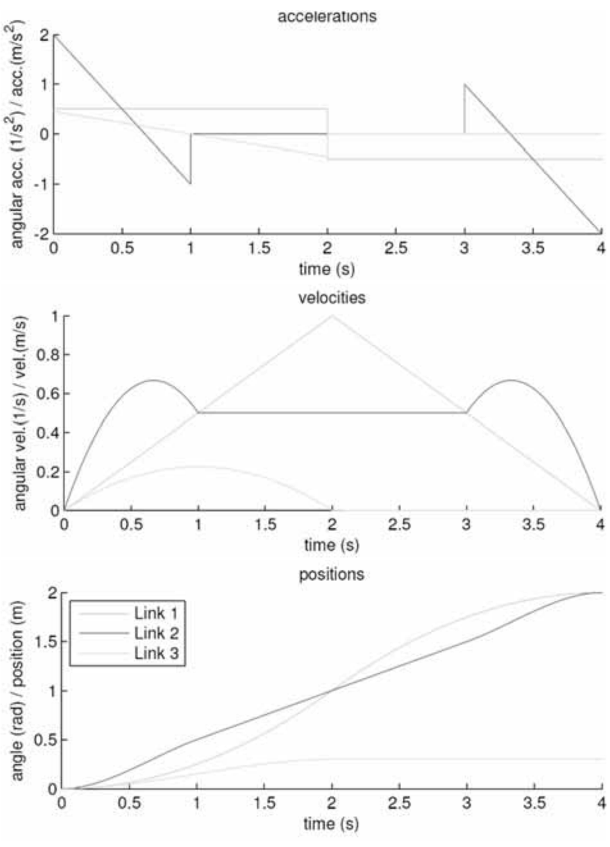


Figure 11: Planned trajectories.

Apart from the feed-forward acceleration it is the same closed loop behaviour as seen earlier and consequently the well-tuned parameters P_i and D_i from the point-to-point motion are used.

6.3 PID-control

The given problem with PD-control has global integral behaviour. This can be seen from the open-loop transfer function e.g. for arm 3 (gravity is unaccounted for)

$$G_0 = \frac{\frac{\sqrt{3}}{2 \cdot k_{T3} \cdot u_3}}{s \cdot (s^2 \cdot ma_{33} \cdot L_{a3} + s \cdot ma_{33} \cdot R_3 + k_{T3} \cdot u_3)}$$

or can be seen in the results of the point-to point movement in Section 7.2 as no steady-state errors remains.

To zero the velocity error for constant velocity [10] and consequently increase the overall performance a PID controller is implemented in the form

$$U_i = q_{i,d} + P \cdot (\ddot{q}_{i,d} - q_i) + I \int (\ddot{q}_{i,d} - q_i) dt + D \cdot \dot{q}_i$$

where the coefficients given in Table 1 were found through pole placement and subsequently were tuned manually.

Gains	q_1	q_2	q_3
Proportional	1500	1000	15000
Integral	10000	7500	5000
Dervivative	10	25	10

Table 1: Coeffcients for PID controller.

7 Results

7.1 Implementation

The three implementations differ considerably in their modelling approach. In the *Simulink* block diagram modelling approach the user has always the overview of the problem, even bigger problems can be well arranged, although, setting up equations by blocks is comparably cumbersome. Even more, *Simulink* unveils its strength in handling the control model, as it offers out of the box blocks for most applications like saturation limits, and ready controller blocks. Zero-crossing for state events are detected automatically and offer the user a rather comfortable experience.

For modelling a mechanical system from scratch, *SimMechanics* offers the quickest and most intuitive approach to model the physical system as there is not even the need to derive equations. However, other than the other simulators one has to deal with the geometric property of the model, as the geometric relations of the joints and bodies have to be specified explicitly in the coordinate system. In recent *MATLAB* versions *SimMechanics* has been based on *SimScape*, a general denominator for 'physical' modelling in mechatronics, electrical engineering and other domains, following the Modelica approach; as consequence, coordinate systems can be avoided. A big advantage is that SimMechanics is fully integrated over the actuators and sensors in the *Simulink* control environment where it is treated as another block. It therefore uses all the *Simulink* functionality regarding state detection, but it does in turn not offer full insight in the process running in the background setting up the mechanical model.

For a given problem with given equations basic *MATLAB* offers with solver ode15i a reliable and reasonably quick implementation for the experienced user to directly compute the given fully implicit equations. The state event property is well integrated in the solver, which is specified over the option set.

However, out of all three approaches the user probably spends the most time debugging in *MATLAB*, as one could easily lose track in the vast amount of variables to be defined. The control law and saturation limits have to be integrated additionally into the functions containing the set of equations which enlarge the formulas and lessens the overview.

7.2 Point-to-point motion

All three approaches execute the point-to-point motion as expected. The result for the Simulink model can be seen in Figure 12.

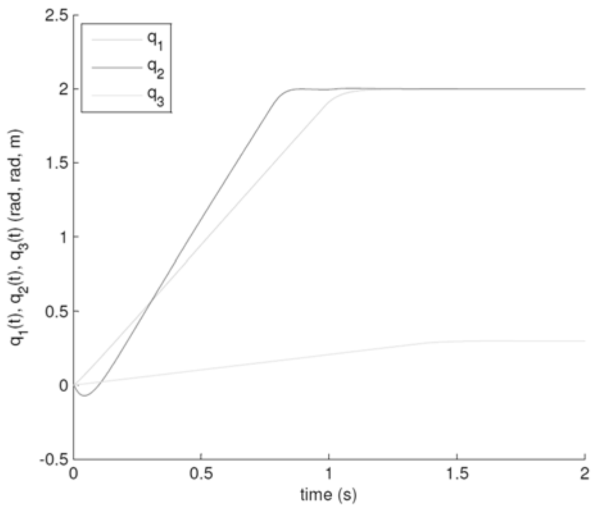


Figure 12: Point-to-point motion in Simulink.

Model description	*Norm. CPU-time*
Simulink explicit	1
Simulink implicit	3.7
MATLAB ode15i	4.6
SimMechanics	6.5 (4.2)

Table 2: Computation time.

Table 2 shows the normed computation time of the approaches for this task. The processing times were measured from *MATLAB* using the tic and toc command. Naturally, the explicit model description with *Simulink* with inverted mass matrix takes by far the least computation time. The implicit description from *Simulink* with algebraic loop breaking and *MATLAB* with implicit solver ode15i are comparable. However, the model implemented with *SimMechanics* almost doubles the time of the *Simulink* implicit model although it uses the same solver and no model manipulation was done beforehand. Recent developments with the *SimScape* basis have done significant improvement and increased the speed (time in parenthesis).

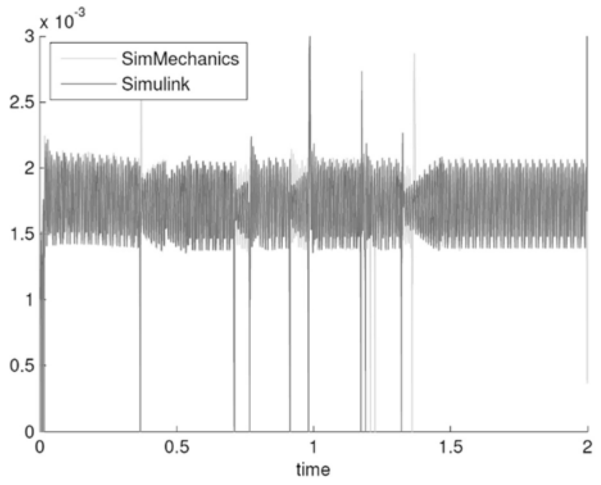


Figure 13: Comparison of stepsize.

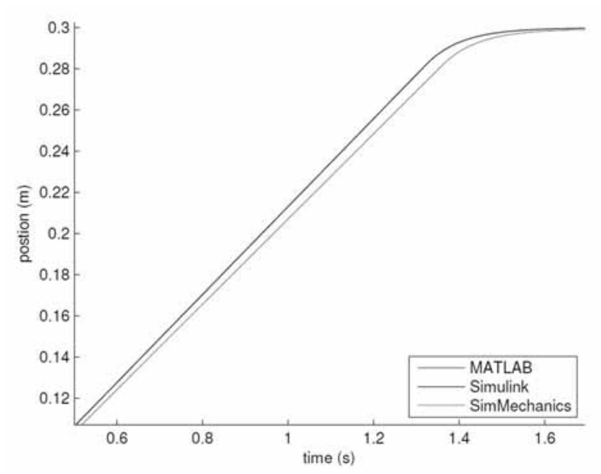


Figure 14: Comparison detail plot.

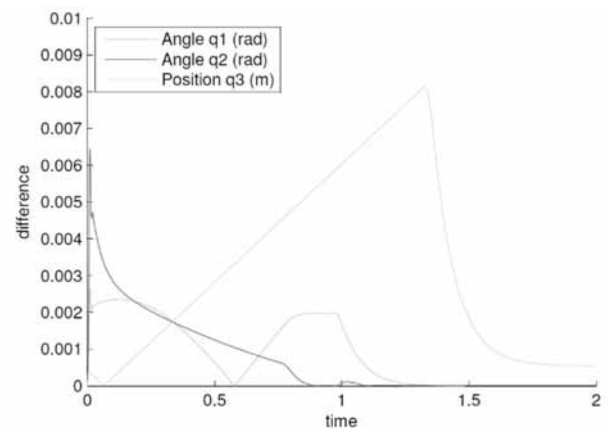


Figure 15: Differences Simulink - SimMechanics

Figure 13 shows the comparison of adaptive stepsizes the ODE solver ode45 used for the *Simulink* implicit model and the *SimMechanics* model. They not only use the same default solver but also have roughly the same step size.

Therefore, the higher computation time results from the premotion model validation and the cooperative solving effort discussed in Section 3.2.

Additionally, in Figure 14 a detail plot of coordinate q_3 for the point-to-point motion in *Simulink*, *SimMechanics* and *MATLAB* ode15i is shown. One can easily see that Simulink and *MATLAB* results are matching; interestingly, the *SimMechanics* result is a bit off – reason may be a difference in the equations fixed before or automatically derived, resp. A comparison of the three angles between Simulink and SimMechanics is depicted in Figure 15: Again, it shows the difference in q_3, which gives a hint for a different integration of the controller in the S*imMechanics* model.

7.3 Collision avoidance

Both *Simulink* and *MATLAB* deliver the same results within a numerical tolerance. This comes as no surprise for the implicit *Simulink* model and the implementation with *MATLAB* ode15i as they solve the same set of equations, but it verifies the *SimMechanics* model in its accuracy.

Figure 16 shows result of the collision avoidance in *SimMechanics* with the path of the tool-tip in *x*-direction and the overlay of the obstacle. The plot shows particularly well that the end-effector stabilises on the edge of the critical zone while robot arm 3 gains height. The adaptive stepsize of the solver ode45 can be seen from the Figure 17 which is a detail plot of Figure 16 and it clearly shows how the stepsize is reduced at a detection of a state event.

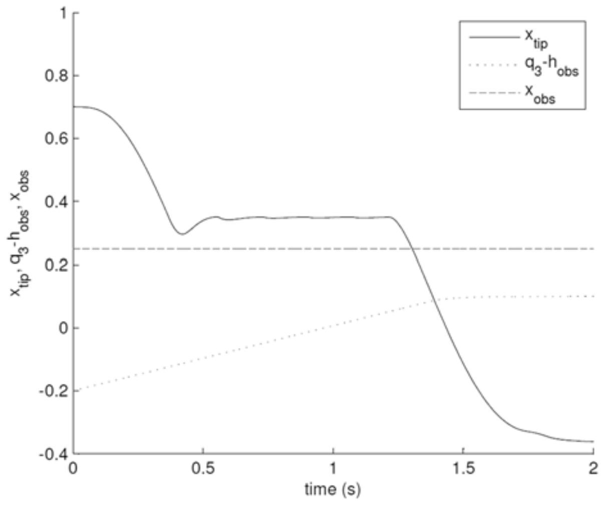


Figure 16: Collision avoidance in SimMechanics.

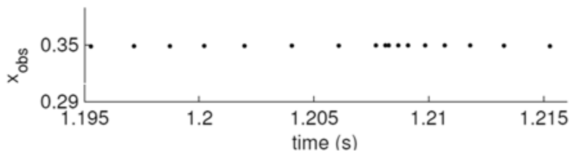


Figure 17: Step size at event detection.

7.4 Trajectory tracking

The deviations of the desired path can be seen in Figure 18 for PD control. It can be seen that for constant velocity for link 2 the deviation is proportional to the velocities. The position error however, decreases with decreasing velocities to zero as the global integral behaviour of the plant suggests.

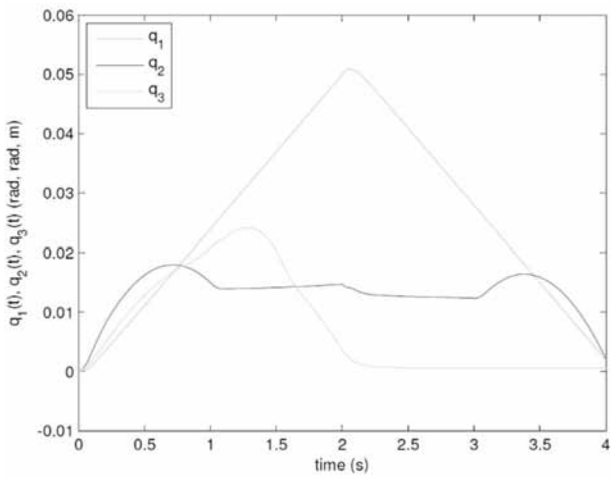


Figure 18: Position error PD-controller.

Figure 19 shows the deviations for the PID controller. When link 2 enters the part with constant velocity at $t = 1$, the position error for link 2 stabilises on zero after an initial overshoot. Here the PID-control comes into effect (as denoted in Section 6.3), zeros the velocity error for constant velocity and consequently the position error.

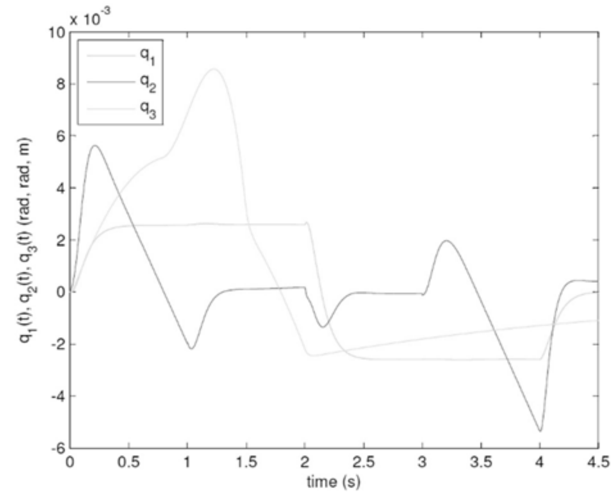


Figure 19: Position error PID-controller.

This can be shown with the response to a ramp input (Figure 20) for e.g. q_2 as the deviation for PID control to the ramp input vanishes [10].

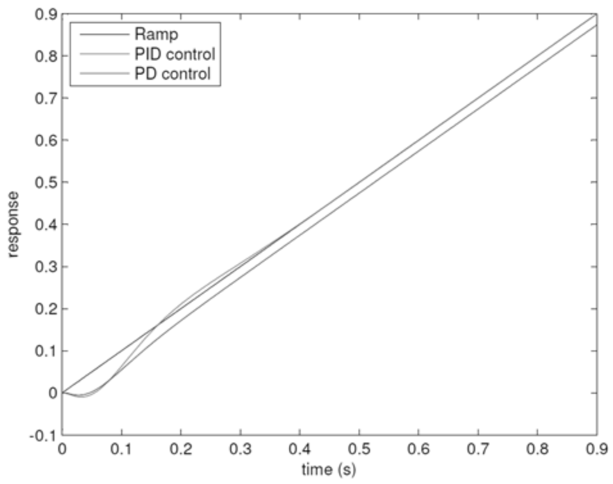


Figure 20: Ramp response.

As the two links are coupled, a little disturbance occurs when link 1 changes the sign of acceleration which is consequently absorbed by the controller. However, the position error of q_3 has an unexpected behaviour, which could result from the unaccounted gravity term. In Figure 19 it can be seen that at $t_{end} = 4$ for PID controller the model deviates from the trajectory, which consequently disappears as the controller integrates the position error and no steady state error remains.

Summing up it can be said that the error polynomials from Figure 18 are reduced one order to Figure 19 and the deviations are reduced approximately one order of magnitude due to the additional integrative term of the PID-controller. Hence, the overall performance has increased significantly.

References

[1] Fritzson P. Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach. ISBN 978-1-118-85912-4, 2014 Wiley-IEEE Press

[2] Ecker H, Breitenecker F. Comparison 11: SCARA Robot – Definition. SNE 8(22), 1998, 30-32.

[3] Simulink R2010/2016 Documentation

[4] SimMechanics User's Guide Version

[5] SimMechanics R2010b Documentation

[6] Shampine L. F. Solving 0 =F(t,y(t),y'(t)) in MATLAB. URL: http://faculty.smu.edu/shampine/cic.pdf, last retrieved Dec. 10, 2020.

[7] MATLAB R2010b/R2016a Documentation

[8] Wikipedia. Dormand-Prince method. Wikipedia, The Free Encyclopedia, 2020. [Online; acc. Dec. 10, 2020]

[9] Canudas de Wit C, Siciliano B. Theory of Robot Control. Springer, 1996.

[10] H.P. Joergl. Repetitorium Regelungstechnik. Band 1. Oldenbourg, 1993.