Controller Design for a Magnetic Levitation Kit using OpenModelica's Integration with the Julia Language

Bernhard Thiele[1] Bernt Lie[2] Martin Sjölund[3] Adrian Pop[3] Peter Fritzson[3]

[1]Institute of System Dynamics and Control, DLR, Germany, bernhard.thiele@dlr.de
[2]University of South-Eastern Norway, bernt.lie@usn.no
[3]PELAB, Linköping University, Sweden, {martin.sjolund,adrian.pop,peter.fritzson}@liu.se

Abstract

This paper presents a practical application of computer aided control systems design using a new OpenModelica API (OMJulia) which allows to conveniently operate on Modelica models from the Julia language. Julia is a rather young language (Julia 1.0 was released in August 2018) designed to address the needs of numerical analysis and computational science, in particular it already has decent support for the control community. The magnetic levitation application at hand demonstrates how control system design can benefit from a suitable integration between Julia and Modelica. It is based on a commercially available control education kit in which the original controller is replaced by our own digital controller developed in this work. There exists an accompanying but independent paper which introduces the complete OMJulia API.

Keywords: OpenModelica, OMJulia, control, magnetic levitation, Arduino, Julia, Modelica

1 Introduction

Modelica is a well established language for modeling complex technical systems supported by several convenient and powerful modeling and simulation environments. Distinguishing language characteristics are the focus on declarative system descriptions using mathematical equations and a specific approach to object orientation which allows encapsulating component behavior (given by data + equations) into reusable units which can be connected by suitable constraints (connect equations) to build complex systems from manageable building blocks, see e.g., (Modelica Association, 2017; Fritzson, 2015).

However, for many numerical analysis tasks an imperative language is well suited and suggests itself. Indeed, the most prevalent software for computer aided control systems design on the market, MATLAB/Simulink[1], has two parts: MATLAB, a numerical computing environment built around the imperative MATLAB scripting language, and Simulink, a primarily graphical block diagram language which is tightly coupled to MATLAB, for modeling and simulation.

Although the Modelica language also has an imperative part for writing algorithms, its support in tools as scripting language has so far remained limited and rather tool specific. Consequently, no rich ecosystem for typical numerical computing tasks like data analysis and advanced data visualization was developed within the community. There are notable exceptions like the LinearSystems library for linear system analysis and controller design (Baur et al., 2009). However, as of the latest release of the library (Modelica_LinearSystems2 v2.3.4[2]) full support of the library is still limited to the Dymola[3] tool.

OpenModelica[4], similarly to other Modelica tools, provides interfaces to dedicated scripting languages which provide the desired advanced scripting support, inclusive a rich ecosystem for numerical analysis and advanced visualization. Based on OMPython (Ganeson, 2012; Ganeson et al., 2012) an API was developed for simple operation on Modelica models from within Python (Lie et al., 2016). However, in the meantime the rather young language Julia[5] has matured (Julia 1.0 was released in August 2018) and has attracted a growing user base in the scientific computing community. The Julia language was originally designed to address the needs of numerical analysis and computational science, in particular it already has decent support for the control community. This motivated the development of OMJulia, an API for interacting with Modelica models from the Julia language. The OMJulia API is described in detail in an accompanying but independent paper (Lie et al., 2019).

The goal of this paper is to demonstrate the interaction between Julia and Modelica models using one of the most popular applications in control education: A *magnetic levitation system*; see, e.g., (Yoon and Moon, 2016; Lilienkamp and Lundberg, 2004; Craig et al., 1988; Wong, 1986). The intention is to present available tool support using a tangible example, it is *not* in the scope of the paper to propose and validate a controller that improves on existing designs.

[1]The MathWorks, https://mathworks.com.

[2]Modelica_LinearSystems2 library, https://github.com/modelica/Modelica_LinearSystems2.

[3]Dassault Systèmes, https://www.3ds.com.

[4]Open Source Modelica Consortium (OSMC), https://www.openmodelica.org.

[5]Julia language, https://julialang.org.

2 Digital Control for a Magnetic Levitation Kit

Magnetic levitation is a popular application for teaching control theory. A levitating object which apparently defies the law of gravity is an attractive gadget and the underlying physics (unstable plant dynamics) convincingly demonstrate the importance of feedback control. The application is based on a commercially available electromagnetic levitation kit[6] from Zeltom which is targeted at educational applications. The fully assembled unit is shown in Figure 1. The vertical position of the levitating magnet

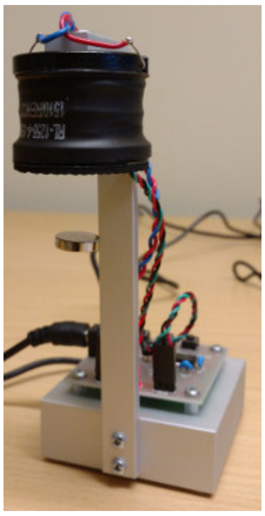

Figure 1. Zeltom's electromagnetic levitation kit.

is measured using a linear Hall effect sensor which is directly attached below the electromagnet. The kit includes a black box microcontroller for controlling the current in the electromagnet.

The goal is to replace Zeltom's controller by our own design.

3 Plant Model

A behavioral model describing the dynamics of the physical system is provided in a technical report by Zeltom (Zeltom LLC, 2009). A schematic diagram of the system is shown in Figure 2, where v is the voltage across the electromagnet, i is the current flowing through the electromagnet, R is the resistance and L the inductance of the electromagnet, e is the voltage across the Hall effect sensor, d is the distance between the Hall sensor and the levitating magnet, m is the mass of the levitating magnet, and f is the force on the levitating magnet generated by the electromagnet.

The nonlinear dynamic equations as described in (Zeltom LLC, 2009) are reproduced below.

[6]Zeltom Electromagnetic Levitation System, `http://zeltom.com/product/magneticlevitation`.

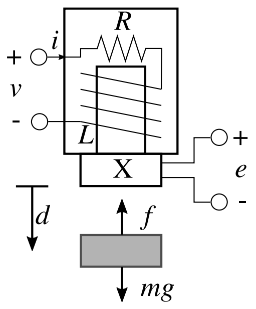

Figure 2. Schematic of the magnetic levitation system.

Approximated force from the electromagnet on the levitating magnet:

$$f = k\frac{i}{d^4}, \tag{1}$$

approximated voltage across the Hall effect sensor:

$$e = \alpha + \beta\frac{1}{d^2} + \gamma i, \tag{2}$$

Newton's second law:

$$m\frac{\mathrm{d}^2 d}{\mathrm{d}t^2} = mg - f, \tag{3}$$

Kirchhoff's voltage Law:

$$v = Ri + L\frac{\mathrm{d}i}{\mathrm{d}t}, \tag{4}$$

where k is a geometry dependent constant, α, β, γ are constants that depend on the Hall sensor and the geometry, and g is the standard gravity constant. The system's parameters are listed in Table 1; the values are from (Zeltom LLC, 2009) and from own measurements.

Table 1. System parameters.

Parameter	*Value*	*Unit*
k	$17.31 \cdot 10^{-9}$	$\mathrm{kg \cdot m^5/A \cdot s^2}$
α	2.44	V
β	$1.12 \cdot 10^{-4}$	$\mathrm{V \cdot m^2}$
γ	0.26	V/A
R	2.41	Ω
L	$15.03 \cdot 10^{-3}$	H
m	$3.02 \cdot 10^{-3}$	kg

Letting v be the control input and e be the measured output, these nonlinear equations can be readily transcribed into a Modelica model (condensed for saving space):

```
model MagLevNL
  parameter Real R=2.41, L=15.03e-3,
      m=3.02e-3, k=17.31e-9, alpha=2.44,
      beta=1.12e-4, gamma=0.26;
  input Real v;
  output Real e;
  Real i, d, d_der, f;
  constant Real g=9.81;
equation
  f = k*i/d^4;
  e = alpha + beta/d^2 + gamma*i;
  der(d) = d_der;
  m*der(d_der) = m*g - f;
  v = R*i + L*der(i);
end MagLevNL;
```

For the purpose of controller design it is typically necessary to work with a linearized version of the plant dynamics. The goal for the magnetic levitation system is to design a controller which stabilizes the plant in an equilibrium position. Therefore, the system needs to be linearized around an equilibrium position of the nonlinear plant. Hence, the first step is to determine an equilibrium position. It would be convenient to have a direct OMJulia API function for this task, similar to

```
mlNL = OMJulia.OMCSession()
mlNL.ModelicaSystem("MagLevNL.mo",
    "MagLevNL")
state_e, u_e, y_e =
    mlNL.findEquilibrium(["d=0.02",
    "d_der=0"])
```

where `findEquilibrium(..)` would search for an equilibrium position under constraints that can be set as function arguments. The function would return the value of the state variables, as well as the value of the inputs and outputs at the equilibrium position. Here, an equilibrium is sought under the constraints that the levitating magnet, levitates at a distance of 2 cm below the sensor.

Unfortunately, such a function is not (yet) available in OMJulia[7]. However, it is possible to modify the Modelica model and impose the equilibrium constraints within a steady-state initialization problem as shown in the listing below. Notice that `input v` was turned into a parameter with unknown value (`fixed=false`) which has the effect that the value is determined during initialization[8]. This is needed since in Modelica a variable which is declared as input is treated as a known, which would result in an overspecified initialization problem below. In order to search for the (unknown) voltage `input` at which the system stays at an equilibrium with the prescribed constraints the Modelica tool needs to treat the voltage input as an unknown.

```
model MagLevNL_SteadyState
  parameter Real R=2.41, L=15.03e-3,
      m=3.02e-3, k=17.31e-9, alpha=2.44,
      beta=1.12e-4, gamma=0.26;
  parameter Real d0 = 0.02 "Prescribed
      equilibrium position";
  parameter Real v(start=0.5, fixed=false)
      "Unknown equilibrium voltage across
      the electromagnet";
  output Real e;
  Real i, d, d_der, f;
  constant Real g=9.81;
equation
  f = k*i/d^4;
  e = alpha + beta*1/d^2 + gamma*i;
  der(d) = d_der;
  m*der(d_der) = m*g - f;
  v = R*i + L*der(i);
initial equation
  d = d0;
  der(d) = 0;
  der(d_der) = 0;
  der(i) = 0;
end MagLevNL_SteadyState;
```

With this model the OMJulia API can be used to retrieve the plant's values at the equilibrium position and use them for linearizing the plant at this equilibrium position. Since the OMJulia API does not (yet) allow to conveniently set start values, the following small modification to the MagLevNL model is introduced, in order to set the start values as parameters:

```
model MagLevNL
  // ... same as previously
  parameter Real i0, d0, d_der0;
  Real i(start=i0,fixed=true),
      d(start=d0,fixed=true),
      d_der(start=d_der0,fixed=true), f;
  // ... same as previously
end MagLevNL;
```

Using this modified model the OMJulia API allows to retrieve the linearized representation of the plant model as shown in the listing below.

```
mlNLe = OMJulia.OMCSession()
mlNLe.ModelicaSystem(
    "MagLevNL_SteadyState.mo",
    "MagLevNL_SteadyState")
mlNLe.setParameters(["d0=0.02"])
mlNLe.simulate()
sol = mlNLe.getSolutions(["v", "i", "d",
    "d_der"])
v_e = sol[1][1] # input v at equilibrium
i_e = sol[2][1] # state i at equilibrium
d_e = sol[3][1] # must be equal to d0
d_der_e = sol[4][1] # must be 0

mlNL = OMJulia.OMCSession()
mlNL.ModelicaSystem("MagLevNL.mo",
    "MagLevNL")
mlNL.setInputs(["v=$v_e"])
mlNL.setParameters(["i0=$i_e", "d0=$d_e",
    "d_der0=$d_der_e"])
A,B,C,D = mlNL.linearize()
```

The final call to the `linearize()` function retrieves a tuple of 2D arrays (matrices) which encode the linearized model in a state space representation ($\dot{x} = Ax + Bu, y = Cx + Du$). The values can be easily inspected, e.g., by

[7] Tools like Wolfram Mathematica (Wolfram Research) or Maple (MapleSoft) support functions for finding local equilibrium points of nonlinear systems. For example Wolfram Mathematica 11.3 introduced a function named "FindSystemModelEquilibrium" which works with (imported) Modelica models and provides respective functionality.

[8] Alternatively, it is possible to declare v as "`Real v(start=0.5, fixed=false)`" and add an equation "`der(v) = 0`".

printing them to the console window (number of digits truncated for readability):

```
julia> println("v_e=$v_e, i_e=$i_e,
    e_e=$e_e")$
v_e=0.66, i_e=0.27, e_e=2.79
julia> println("A=$A\nB=$B\nC=$C\nD=$D")
A=[0.0 1.0 0.0; 1962.0 0.0 -35.8237; 0.0
    0.0 -160.346]
B=[0.0; 0.0; 66.5336]
C=[-28.0 0.0 0.26]
D=[0.0]
```

4 Control Design

The Julia ecosystem provides various packages which can support a control design process. The OMJulia bridge to OpenModelica allows to combine the strength of those packages with the powerful modeling and simulation infrastructure of a Modelica tool. This section will demonstrate some possibilities.

The magnetic levitation system is open-loop unstable, which can be quickly checked using the ControlSystems.jl package[9]. Function `mlLin=ss(A, B, C, D)` creates a state-space model from the previously retrieved matrices of the linearized magnetic levitation model. Function `pole(mlLin)` returns its poles.

```
julia> using ControlSystems
julia> mlLin = ss(A,B,C,D)
julia> pole(mlLin)
3-element Array{Float64,1}:
    44.294469180700204   -44.2944691807002
       -160.346
```

It is known that a PD controller is capable of stabilizing a magnetic levitation system. Indeed, Yoon and Moon have shown in (Yoon and Moon, 2016) that the system at hand can be stabilized by a simple PD analog controller. A particular challenge in respect to stabilizing a magnetic levitation system is designing a reasonable *robust* controller. A measure for the robustness of a design is the sensitivity function S, which describes the transfer function from an external disturbance to the process output. Lower values of $|S|$ suggest further attenuation of the external disturbance (hence, lower is better). The following paragraphs briefly introduces the Julia code used for sensitivity analysis of the controlled system.

A PD controller is described by the transfer function

$$C_{PD}(s) = K_p(T_d s + 1), \tag{5}$$

where K_p is the proportional gain, and T_d is the derivative time parameter. The listing below uses the construct `s = tf("s")` to create a continuous-time transfer function `s`, which enables a convenient notation for creating transfer functions using standard mathematical operators like `PD = Kp*(Td*s + 1)`.

[9]ControlSystems.jl, https://github.com/JuliaControl/ControlSystems.jl.

The plant's state space representation from above can be converted into a transfer function representation $G(s)$ using function `tf(..)`, e.g., `G = tf(mlLin)`. Using the plant's transfer function $G(s)$, the open-loop transfer function is given by a serial connection of controller and plant,

$$P_{PDol}(s) = C_{PD}(s)G(s), \tag{6}$$

which can be achieved using the `series(..)` function in Julia. The sensitivity function is then given by

$$S(s) = \frac{1}{1 + P_{PDol}(s)}. \tag{7}$$

Since poles are not canceled automatically, the function `minreal(..)`[10] is used to obtain a minimal transfer function representation.

Function `bodeplot(..)` is used for plotting the magnitude of the sensitivity function. Using the Julia Interact.jl package[11] together with functions from ControlSystems.jl allows for interactive plots in which the controller's parameters can be tuned experimentally. The Interact package provides means to create small GUIs in Julia based on web technology. It defines the macro `@manipulate` which sets up sliders for varying the parameters within the specified range.

```
using Interact
s = tf("s")
@manipulate for Kp=3:.5:20, Td=0.01:.01:0.1
  PD = Kp*(Td*s + 1)
  mlLinPDol = series(PD,tf(mlLin))
  mlLinPDSensitivity =
      minreal(1/(1+mlLinPDol))
  bodeplot(mlLinPDSensitivity,
      plotphase=false, yscale=:identity,
      yticks=0:0.1:2, title="Sensitivity")
end
```

Evaluating the above code in an IJulia/Jupyter session gives a result as depicted in Figure 3. The two sliders at the top allow to change the PD controller's parameters. When the parameters are changed, the plot is immediately updated.

5 Nonlinear Closed-Loop Model

After an acceptable design (based on the linearized model) has been found, the controller can be tested and further tuned by plugging it into the nonlinear Modelica model.

In the present example the PD controller can be easily transcribed into Modelica code and can be added appropriately to the MagLevNL model in order to close the loop between controller and plant. Let the resulting model be named "MagLevNLPD" (the complete listing is given in Appendix A).

[10]Function `minreal(..)` creates a minimal transfer function representation by canceling pole-zero pairs.

[11]Interact.jl, https://github.com/JuliaGizmos/Interact.jl.

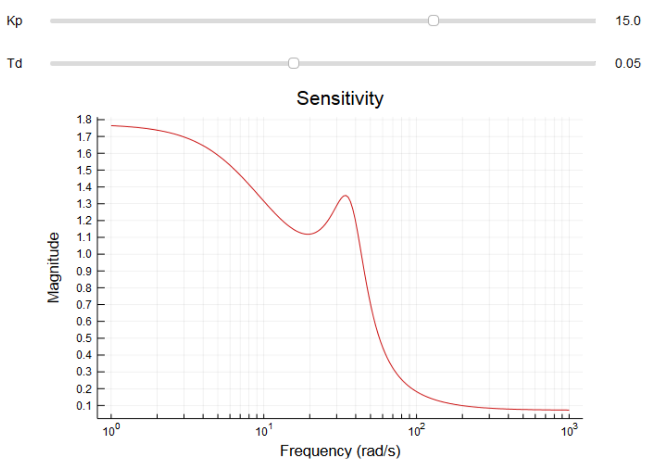

Figure 3. Interactive sensitivity plot for the magnetic levitation system in which the controller parameters can be varied using sliders.

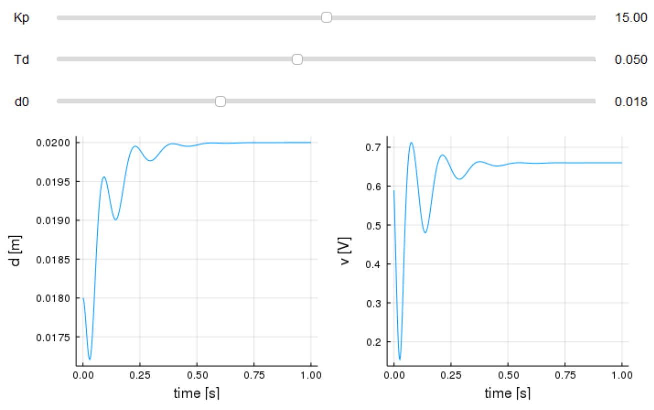

Figure 4. Simple interactive GUI with sliders for setting parameters of the closed-loop nonlinear magnetic levitation Modelica model using the OMJulia interface. Changing a slider will immediatly trigger a new simulation and update the plots.

Combining OMJulia with the Interact package allows to quickly create small GUIs for interactive experimentation with a Modelica model. The Julia code below creates sliders for varying the controller parameters, as well as to vary the initial distance d_0 of the levitating magnet. Since the controller is designed for keeping an equilibrium position at $d = 0.02\,\mathrm{m}$, it is interesting to explore how the closed system behaves for small displacements, where $d_0 \neq 0.02\,\mathrm{m}$.

```
using OMJulia, Plots, Interact
mlNLPD = OMJulia.OMCSession()
mlNLPD.ModelicaSystem("MagLevNLPD.mo",
    "MagLevNLPD")
@manipulate for Kp=7:0.5:23,
    Td=0.01:0.01:0.1, d0=0.015:0.0002:0.025
  mlNLPD.setParameters(["Kp=$Kp",
      "Td=$Td", "d0=$d0"])
  mlNLPD.simulate()
  sol = mlNLPD.getSolutions(["time", "d",
      "v"])
  time, d, v = sol[1], sol[2], sol[3]
  p1 = plot(time, d, label="",
      xlabel="time [s]", ylabel="d [m]")
  p2 = plot(time, v, label="",
      xlabel="time [s]", ylabel="v [V]")
  plot(p1, p2, layout=(1,2))
end
```

Figure 4 shows a screenshot of the resulting GUI when evaluating the above code in an IJulia/Jupyter session. The start value of d is set to $d_0 = 18\,\mathrm{mm}$, hence two millimeters closer to the electromagnet than the set reference distance of 20 millimeters. The left plot shows how the distance d starts at the prescribed start value and is regulated to the reference distance of 20 millimeters. The right plot shows the voltage v (the actuating variable) that the controller sets to the electromagnetic actuator. Notice that the voltage remains in reasonable limits (no actuator saturation). However, further exploration (using the same controller parameters) showed that the closed loop stabilization for the nonlinear model fails quickly when choosing start values d_0 which are greater than the reference distance of 20 millimeters.

6 Digital Control

For a practical implementation of the presented PD controller, the derivative “D” part is first approximated by a “DT$_1$” element before the controller is discretized in a second step. Finally, hardware characteristics of the target controller are considered in a nonlinear closed-loop, sampled-data model.

The “D” part can be approximated by $sT_d \approx \frac{sT_d}{1+sT_d/N_d}$, where N_d limits the gain at high frequencies (typically: $3 \leq N_d \leq 20$). Therefore, the structure of the controller becomes

$$C_{PDT_1}(s) = K_P \left(\frac{T_d s}{\frac{T_d}{N_d} s + 1} + 1 \right). \tag{8}$$

6.1 Discrete-Time Approximation

Using backward differences for approximation, transfer function (8) can be transformed into a pulse-transfer function by substituting s by s' using the formula

$$s' = \frac{z-1}{zh}, \tag{9}$$

where h is the sampling period and z is the Z-transform variable, resulting in the pulse-transfer function

$$C_{PDT_1}(z) = K_p \left(\frac{T_d N_d (z-1)}{(T_d + N_d h) z - T_d} + 1 \right). \tag{10}$$

The pulse-transfer function can be readily transformed into a recurrance relation which directly translates into Modelica code. The listing below shows a condensed version of the discretized controller using Modelica’s clocked synchronous language elements.

```
block Controller
  parameter Real Kp=15, Td=0.05, Nd=5,
      h=0.0005, v_e=0.66, e_e=2.79;
  input Real du_set "Setpoint delta
      voltage (=0 for d=>0.02)";
  input Real e "Measured voltage across
      the Hall effect sensor";
  output Real v "Output voltage to the
      electromagnet";
protected
  Real Dpart(start=0), de_e, du(start=0),
      dy, ad, bd;
equation
  // Measured delta voltage at OP
  de_e = e - e_e;
  // input to PD(T1) control law
  du = du_set - de_e;

  // Control law
  ad = Td/((Td + Nd*h));
  bd = Td*Nd/(Td + Nd*h);
  Dpart =  ad * previous(Dpart) + bd * (du
      - previous(du));
  dy = Kp*(du + Dpart);

  // Output voltage to electromagnet
  v = dy + v_e;
end Controller;
```

6.2 Target Hardware

The popular Arduino Uno board[12] is used as implementation hardware for the control algorithm. It is based on the Microchip ATmega328P microcontroller, has six analog inputs supporting 10-bit analog-to-digital conversion (ADC) for input voltages between zero and five volts, and 14 digital input/output pins of which six can be used as pulse-wide modulation (PWM) outputs. The frequency of the PWM outputs is configurable and a simple interface exists in which the PWM duty cycle can be set with a resolution of 8-bit.

In our application the voltage across the Hall effect sensor is read using one of the analog inputs. The voltage to the electromagnet is set by a PWM output driving a MOSFET which is connected to a DC voltage regulator fed from an external power supply. A breadboard is used for the implementation of the supporting electronics (see Figure 8).

6.3 Sampled-Data Model

A model of the closed-loop, sampled-data system can be built conveniently with the help of the Synchronous library (Otter et al., 2012). Figure 5 shows a diagram view in which the nonlinear continuous-time magnetic levitation plant model is connected to the discrete-time (clocked) controller model using sample and hold blocks from the Synchronous library. The controller is activated by a periodic clock with a sampling period of 500 μs. The upper controller input specifies the setpoint of the controller. The setpoint is 0 V for the equilibrium position

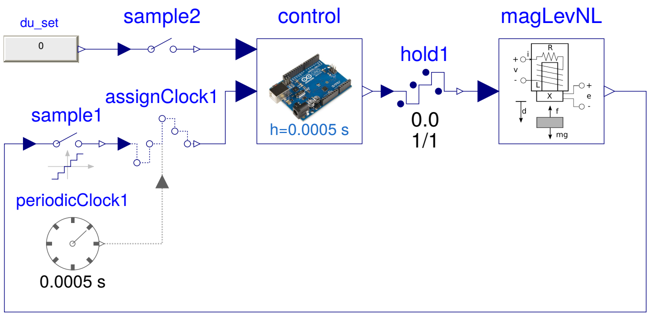

Figure 5. Closed-loop magnetic levitation system with clocked controller model.

for which the controller is designed (i.e., in the presented design the levitating object is at $d = d_e = 0.02\,\mathrm{m}$, the Hall effect sensor output is $e = e_e = 2.79\,\mathrm{V}$). Modifying the setpoint allows to influence the position of the levitating magnet.

The utilized sample and hold blocks allow modeling additional real-world effects like noise, quantization effects of digital-analog and analog-digital conversions, sensor and actuator limitations, and computational delays. In the displayed model the sample and hold blocks are parametrized so that they reflect the capabilities of the target hardware as described in Section 6.2. The measurement variable e is limited between $0\,\mathrm{V} \le e \le 5\,\mathrm{V}$ using 10-bit quantization. The actuating variable v is limited between $0\,\mathrm{V} \le v \le 1.3\,\mathrm{V}$ using 8-bit quantization.

Simulating the sampled-data model given above using the same scenario as for the nonlinear continuous-time model in Section 5 reveals a severe control degradation for the considered digital controller. Further simulation experiments reveal that this is mainly due to ADC quantization effects of the Hall effect sensor output. Figure 6 shows results plots[13] for simulating with different ADC settings, while the other settings, e.g., computational delay of one sampling period and 8-bit quantization of the actuating variable, are unchanged. The upper plot shows the distance d of the levitating object. The two lower plots show the sampled and quantized Hall sensor output e (= `sample1.y`) and the quantized actuating variable v (= `hold1.y`) in a narrow time window ($t \in [4.00\,\mathrm{s}, 4.03\,\mathrm{s}]$).

Although the levitating object can be stabilized in all simulated cases, it shows persisting oscillations for the case of an ADC with 10-bit resolution over the range $[0\,\mathrm{V}, 5\,\mathrm{V}]$. For this setting, the actuating variable exhibits large, high frequency oscillations. Increasing the quantization resolution mitigates this adverse effect and restores

[12] Arduino, `https://arduino.cc`.

[13] Apart from using OMJulia for controlling the complete simulation (as shown in Section 5), it is also possible to use the Julia CSV package for simply importing an OpenModelica (CSV-) result file into Julia for postprocessing. For example, plotting variable `magLevNL.d` from a CSV-result file can be achieved by:

```
using Plots, CSV
r = CSV.read("myresultfile.csv")
plot(r[Symbol("time")], r[Symbol("magLevNL.d")])
```

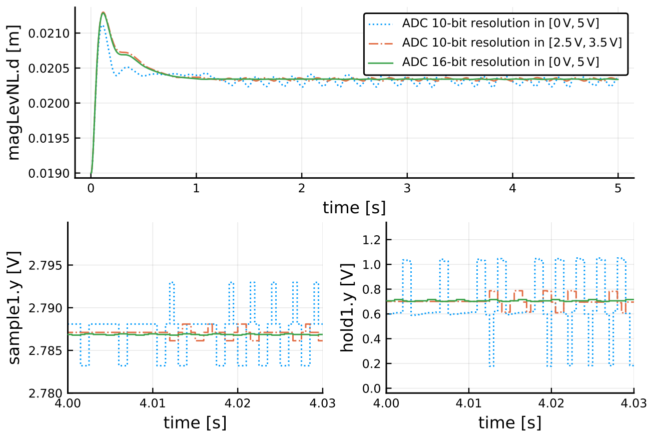

Figure 6. Simulation results of the sampled-data model for different ADC quantization settings and an initial distance $d_0 = 0.019\,\text{m}$.

a behaviour which is closer to the continuous-time controller. Besides increasing the ADC resolution (e.g., to 16-bit), the simulation results suggest that a 10-bit ADC resolution is fine, if it is available within the (smaller) relevant operating range of the sensor, e.g., $[2.5\,\text{V}, 3.5\,\text{V}]$. This can be achieved by using a suitable signal conditioning circuit for mapping the signal's operating range to the full-scale voltage range of the ADC.

6.4 Real-Time Target Code

In a first approach, the Modelica code for the demonstrator presented in Section 7 was hand translated to C in order to compile and upload it to Arduino. This is rather straightforward, since the control algorithm is short and the Arduino environment is easy to use.

However, particularly for more complex models, it would be beneficial to automatically generate the target code, instead of manually converting the controller models to compact C code. This is quicker and less error prone than manual translation. One big challenge is to produce target code that fits into very small foot-print platforms.

For these reasons we have developed an experimental version of an embedded target simple code generator[14] for OpenModelica aimed at very restricted platforms such as the Atmel AVR 8-bit microcontrollers. The regular C-code generator creates huge data structures and contains much debugging information while the run-time system contains many numerical solvers and is around 6 MB in size (of which 0.5 MB is textual strings for error messages). This regular C-code is intended to run on powerful desktop CPUs where the code size does not matter much and it proved difficult to try to strip out unnecessary code when targeting embedded systems. The largest of the 8-bit AVR processor MCUs (Micro Controller Units) have 16 kB SRAM. One of the smaller ones (ATmega328P; Arduino Uno) has 2 kB SRAM.

The embedded target code generator was designed to generate code for constructs that are easy to compile. For example, it does not support arrays, strongly connected components, or initialization, but still works fine for many models since the OpenModelica compiler will convert many complex constructs into simpler ones during the compilation process, e.g., make array equations into scalar equations. Instead of having a big run-time system that is linked in (as is the case for the regular code generator), the code generator will generate the needed C-functions corresponding to the Modelica and run-time functions called.

As can be seen from Table 2, the experiment was so far successful. The regular stripped-down code generator

Table 2. Code generator comparison. Regular vs Simple.

	Regular stripped-down source-code FMU targeting 8-bit AVR processor	*Simple code generator targeting 8-bit AVR processor*
Minimal model (0 equations)	43 kB flash memory, 23 kB variables (RAM)	130 B flash memory, 0 B variables (RAM)
Target system including controller	68 kB flash memory, 25 kB variables (RAM)	3350 B flash memory, 169 B variables (RAM)

with almost everything stripped out except the main simulation loop (it includes no solvers or numerical routines except the used ones) already reduces the code foot-print significantly compared to the standard desktop version. However, it is still too large for very small foot-print platforms like the Arduino Uno. The simple code generator allows a further reduction in size which makes it suitable for very small foot-print platforms.

The clocked controller model from Figure 5 needs to be adapted in order to be suited as input to the experimental embedded target code generator. The embedded target code generator in the development branch for the upcoming OpenModelica v1.14 release does not yet support the synchronous clocked language elements, nor does it support when-equations for modeling sampled systems. As a workaround the clocked controller equations can be rewritten as an algorithm and placed into an algorithm section. In the generated code this algorithm section is called periodically using a base rate which can be specified during translation[15].

Figure 7 shows the input model for the code generator. The model uses blocks for interfacing to hardware facilities of the Arduino Uno like ADC or PWM units. These hardware interface blocks are available in the Modelica_DeviceDrivers library (Thiele et al., 2017). The model assumes that a signal conditioning circuit is used for mapping the Hall effect sensor voltage around the op-

[14]The embedded code generation target for OpenModelica can be activated by passing the option `--simCodeTarget=ExperimentalEmbeddedC` to the OMC.

[15]For example, when using OpenModelica's scripting interface the base period can be specified by providing the `stepSize` argument to the `translateModel(..)` function.

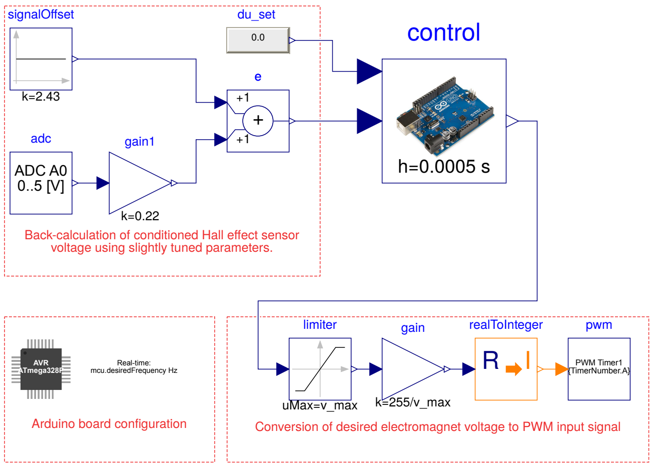
signalOffset
du_set
0.0
control
k=2.43
e
+1
+
+1
adc
gain1
ADC A0
0..5 [V]
k=0.22
h=0.0005 s
Back-calculation of conditioned Hall effect sensor voltage using slightly tuned parameters.
limiter
gain
realToInteger
pwm
AVR ATmega328P
Real-time: mcu.desiredFrequency Hz
R
I
PWM Timer1 (TimerNumber.A)
uMax=v_max
k=255/v_max
Arduino board configuration
Conversion of desired electromagnet voltage to PWM input signal

Figure 7. The input model for the code generator consisting of the control algorithm and hardware related blocks.

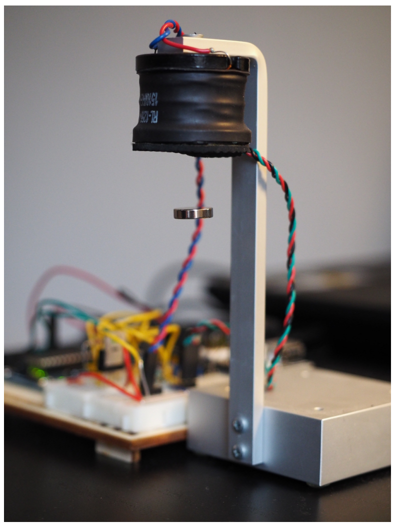

Figure 8. Arduino controlled electromagnetic levitation system.

erating point to the full-scale voltage range of the ADC. Notice that the parameters for back-calculation of the conditioned Hall effect sensor signal deviate slightly from the theoretical values ($k = 2.5$ for the signal offset and $k = 0.2$ for the signal gain). This is the result of tuning the parameters for the actual demonstrator with its (non-ideal) supporting electronics.

Most of the parameters in the hardware interface blocks are at their default values. However, several interesting parameter settings are not visible on the diagram layer. The microcontroller block is set to the ATmega328P platform and its internal parameter `desiredPeriod` is set to 0.0005 s. The real-time block is configured to use `Timer0` for the real-time synchronization. The PWM block is configured to use `Timer1` with a prescaler value of “1/8”.

7 Demonstrator

Figure 8 shows a setup in which Zeltom’s controller has been replaced by an Arduino Uno and supporting electronics.

It was possible to stabilize the levitating mass for several minutes at a time using the presented controller and the experimental hardware setup with a 10-bit ADC resolution in the range of $[0\,\mathrm{V}, 5\,\mathrm{V}]$, but the magnet showed clearly visible oscillations around the equilibrium position and was very sensitive to disturbances, e.g, a tiny push against the table would destabilize the mass instantly. Motivated from the simulation results in Figure 6, a signal conditioning circuit based on Texas Instrument’s INA333 instrumentation amplifier was developed to map the operating range $[2.5\,\mathrm{V}, 3.5\,\mathrm{V}]$ of the Hall effect sensor signal to the full-scale voltage range of the ADC. As suggested by the simulation results, this attenuated the oscillation and lead to a greatly improved robustness in maintaining the equilibrium position.

8 Conclusion

The paper shows how computer aided control system design based on Modelica models can benefit from the new OpenModelica OMJulia API which allows joint interaction between the Modelica and Julia ecosystems. For practical illustration a complete magnetic levitation application is presented with sufficient details so that the example can be readily reproduced, e.g., in the context of a lab session in control education.

While Modelica excels in modeling and simulation of complex technical systems, Julia can provide the numerical analysis, optimization and advanced visualization capabilities, including specialized packages for control engineering. Simple web technology based GUIs can be created in Julia in just a few lines of code, which allows interactive experimentation with Modelica simulation models giving immediate feedback to the user, e.g., by updating key performance plots. The magnetic levitation application aims at illustrating how a carefully designed API has the potential to leverage attractive synergies between the two languages.

Acknowledgements

This work has been supported by Vinnova in the ITEA OPENCPS and EMPHYSIS projects, and in the Vinnova RTISIM project. Support from the Swedish Government has been received from the ELLIIT project. The OpenModelica development is supported by the Open Source Modelica Consortium.

References

Marcus Baur, Martin Otter, and Bernhard Thiele. Modelica Libraries for Linear Control Systems. In Francesco Casella, editor, 7th *Int. Modelica Conference*, Como, Italy, September 2009. doi:10.3384/ecp09430068.

Kevin Craig, Thomas Kurfess, and Mark Nagurka. Magenetic levitation testbed for controls eduction. In *Proceedings of the ASME Dynamic Systems and Control Division*, volume 64, 1988.

Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley-IEEE Press, Piscataway, NJ, second edition, 2015. ISBN 978-1-118-85912-4.

Anand Ganeson. Design and Implementation of a User Friendly OpenModelica - Python interface. Master's thesis, Linköping University, 2012.

Anand Ganeson, Peter Fritzson, Olena Rogovchenko, Adeel. Asghar, Martin Sjölund, and Andreas Pfeiffer. An OpenModelica Python Interface and its Use in PySimulator. In Martin Otter and Dirk Zimmer, editors, 9th *Int. Modelica Conference*, Munich, Germany, September 2012. doi:10.3384/ecp12076537.

Bernt Lie, Sudeep Bajracharya, Alachew Mengist, Lena Buffoni, Arunkumar Palanisamy, Martin Sjölund, Adeel Asghar, Adrian Pop, and Peter Fritzson. API for Accessing OpenModelica Models from Python. In *Proceedings of EuroSim 2016*, Oulu, Finland, September 2016.

Bernt Lie, Arunkumar Palanisamy, Alachew Mengist, Lena Buffoni, Martin Sjölund, Adeel Asghar, Adrian Pop, and Peter Fritzson. OMJulia: An OpenModelica API for Julia-Modelica Interaction. In Anton Haumer, editor, 13th *Int. Modelica Conference*, Regensburg, Germany, March 2019.

Katie A. Lilienkamp and Kent Lundberg. Low-cost magnetic levitation project kits for teaching feedback system design. In *Proceedings of the 2004 American Control Conference*, volume 2, pages 1308–1313 vol.2, June 2004. doi:10.23919/ACC.2004.1386755.

Modelica Association. Modelica - A Unified Object-Oriented Language for Systems Modeling - Version 3.4. Standard Specification, April 2017. URL http://www.modelica.org/.

Martin Otter, Bernhard Thiele, and Hilding Elmqvist. A Library for Synchronous Control Systems in Modelica. In Martin Otter and Dirk Zimmer, editors, 9th *Int. Modelica Conference*, Munich, Germany, September 2012. doi:10.3384/ecp1207627.

Bernhard Thiele, Thomas Beutlich, Volker Waurich, Martin Sjölund, and Tobias Bellmann. Towards a Standard-Conform, Platform-Generic and Feature-Rich Modelica Device Drivers Library. In Jiří Kofránek and Francesco Casella, editors, 12th *Int. Modelica Conference*, Prague, Czech Republic, May 2017. doi:10.3384/ecp17132713.

T. H. Wong. Design of a Magnetic Levitation Control System - An Undergraduate Project. *IEEE Transactions on Education*, E-29(4):196–200, Nov 1986. ISSN 0018-9359. doi:10.1109/TE.1986.5570565.

Myung-Gon Yoon and Jung-Ho Moon. A Simple Analog Controller for a Magnetic Levitation Kit. *International Journal of Engineering Research & Technology (IJERT)*, 5(3):94–97, March 2016.

Zeltom LLC. Electromagnetic Levitation System - Mathematical Model, June 2009. URL http://zeltom.com/documents/emls_md.pdf.

A Listing of the Nonlinear Closed-Loop MagLev Model

The complete listing of the nonlinear closed-loop Modelica model used in Section 5.

```
model MagLevNLPD
  // Parameters MagLev
  parameter Real R=2.41, L=15.03e-3,
      m=3.02e-3, k=17.31e-9, alpha=2.44,
      beta=1.12e-4, gamma=0.26;
  // Equilibrium point (values actually
      depend on parameters above!)
  parameter Real v_e=0.659957,
      e_e=2.791198;
  // Setting initial conditions to values
      at equilibrium point
  parameter Real d0=0.02, d_der0=0,
      i0=0.273841;
  // Variables MagLev
  Real d(start=d0, fixed=true),
      d_der(start=d_der0, fixed=true),
      i(start=i0, fixed=true), v, f, e;
  constant Real g=9.81;
  // Parameters PD
  parameter Real Kp=15, Td=0.05;
  parameter Real du_set=0 "Desired
      setpoint OP delta voltage of PD
      controller";
  // Variables PD
  Real u,y;
equation
  u = du_set - (e - e_e) "Input to the PD
      controller (negative feedback loop)";
  y = Kp*(u + Td*der(u)) "Ideal PD
      controller";
  v = y + v_e "Controller output to the
      plant";
  // Nonlinear MagLev plant equations
  f = k*i/d^4 "(1) force applied by the
      electromagnet on the levitating
      magnet";
  e = alpha + beta*1/d^2 + gamma*i "(2)
      voltage across the Hall effect
      sensor";
  der(d) = d_der;
  m*der(d_der) = m*g - f "(3) Newton's
      second law that";
  v = R*i + L*der(i) "(4) Kirchhoff's
      voltage law";
end MagLevNLPD;
```