# OMJulia: An OpenModelica API for Julia-Modelica Interaction

Bernt Lie[1], Arunkumar Palanisamy[2], Alachew Mengist[2], Lena Buffoni[2], Martin Sjölund[2], Adeel Asghar[2], Adrian Pop[2], Peter Fritzson[2]

[1]University of South-Eastern Norway, Porsgrunn, Norway, Bernt.Lie@usn.no;
[2]Linköping University, Linköping, Sweden, Peter.Fritzson@liu.se

## Abstract

Modelica is an object oriented, acausal equation-based language for describing complex, hybrid dynamic models. About ten Modelica implementations exist, of which most are commercial and two are open source; the implementations have varying levels of tool functionality. Many Modelica implementations have limited support for model analysis. It is therefore of interest to integrate Modelica tools with a powerful scripting and programming language, such as Julia. Julia is a modern and free language for scientific computing. Such integration would facilitate the needed analysis possibilities and can speed up the development of effient simulation models. A number of design choices for interaction between Julia and Modelica tools are discussed. Next, Julia package OMJulia is introduced with an API for interaction between OpenModelica and Julia. Some discussion of the reasoning behind the OMJulia design is given. The API is based on a new class *ModelicaSystem* within package OMJulia, with systematic methods which operate on instantiated models. OMJulia supports handling of FMU and Modelica models, setting and getting model values, as well as some model operations. Results are available in Julia for further analysis. OMJulia is a further development of a previous OMPython package; a key advantage of Julia over Python is that Julia has better support for control engineering packages. OMJulia represents a first effort to interface a relatively complete Modelica tool to Julia, giving access to an open source set-up for modeling and analysis, including control synthesis, easily installable from a unified package manager. Some possibilities of OMJulia are illustrated by application to a few simple, yet industrially relevant problems within control design. *Keywords*: Modelica, FMI, FMU, OpenModelica, Julia, Julia API, OMJulia

## 1 Introduction

Julia is a modern, rich script language, (Bezanson et al., 2017), with excellent support for efficient and fast differential equation solvers (Rackauckas and Nie, 2017), including DAEs (Sund et al., 2018), as well as a number of other packages for plotting, control engineering, optimization, statistics, machine learning, etc., (JuliaLang, 2018).

*Modelica* is a modern, equation based, acausal language for encoding models of dynamic systems in the form of differential algebraic equations (DAEs), see, e.g., (Modelica Association, 2016), (Modelica Association, 2017), (Fritzson, 2015) on Modelica, and, e.g., (Brenan et al., 1989) on DAEs. The Functional Mock-up Interface (FMI) is a common standard format to support both model exchange and co-simulation of dynamic models in the form of Functional Mock-up Units (FMU) between many modelling and simulation environments, (FMI Consortium, 2018).

*OpenModelica*[1] (Fritzson et al., 2018) is a mature, freely available tool set that includes *OpenModelica Connection Editor* (flow sheeting, textual editor with debugging facilities, and simulation environment), the *OMShell* (command line/script based execution), and a number of extensions. OpenModelica Shell supports commands for simulation of Modelica models, for use of the Modelica extension Optimica, for carrying out analytic linearization via the Modelica package *Modelica_LinearSystem2*, and for converting Modelica models into Functional Mock-Up Units (FMUs) as well as for converting FMUs back to Modelica models. However, the OMShell is relatively limited wrt. other, advanced analysis possibilities such as availability of random number generator, control tools, etc.

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). Both Modelica (Baur et al., 2009) and Python[2] have limited support for control tools, and it is of interest to explore connecting OpenModelica to other scripting tools with richer eco-systems for control engineering — two possibilities are MATLAB and Julia. To ease the maintenance of interfacing Modelica with 3 different script languages, it is necessary to compromise on the specific style of each language. This paper discusses the API adapted to Julia, and illustrates how OMJulia can be used for analysis of Modelica models, exemplified by a simple water tank model, and then for more advanced analysis of a nonlinear reactor model[3]. The paper is organized as follows. In Section 2, an overview of the API is given. In Section 3, use of the API is applied to analysis of a simple, process oriented model. In Section 4, a somewhat more complex chemical engineering type process is used to illustrate possibilities

[1] `www.openmodelica.org`

[2] https://sourceforge.net/p/python-control/wiki/Home/

[3] The nonlinear reactor case will be added in the final paper.

with combining OpenModelica with Julia. In Section 5, some discussion of the API is provided with conclusions.

# 2 Overview of Julia API

## 2.1 Goal

Julia is a modern, rich script language, while Modelica, offers mature, equation based encoding of physically based models, with system (input-output), and library support. It is of interest to consider the use of Modelica with Julia for a wide range of engineering disciplines. The computer science threshold of using Modelica with Julia should be low. The OMJulia extension should be installed via the standard Julia packet manager (Git-based), and support the same platforms as Julia does. Results should be returned as standard Julia structures.

OMJulia can be installed as described at `https://github.com/OpenModelica/OMJulia.jl`.

## 2.2 Design Choices

Essentially, four paths to Modelica–Julia interaction are realistic[4].

1. Sending Modelica script commands as text strings from Julia to the Modelica tool via the ZMQ communication protocol[5] (Hintjens, 2013), and retrieving results. This is similar to the original idea of OMPython[6]. Advantage: simple solution. Disadvantage: requires detailed knowledge of Modelica tool script commands; possibly relatively slow if the interaction time is a large fraction of the computation time.

2. Julia API with commands native to Julia, which are translated to Modelica script commands "behind-the-scene", interacts with Modelica via ZMQ, and with results returned to Julia in Julia objects. Advantage: simple to use within Julia. Disadvantage: limited to existing possibilities in Modelica tool; possibly relatively slow.

3. Translate Modelica code to Julia code. Currently, OpenModelica code is translated to C code. It is possible to alternatively translate the code to Julia code. Advantage: utilize specialized syntax (Modelica) for describing models, and with full integration with Julia, fast. Disadvantage: the user must handle two languages.

4. Extend Julia with Modelica-like structures, such as the Modia initiative (Elmqvist et al., 2017). Advantage: the user operates in one language, fast. Disadvantage: limitations in Julia syntax and slightly different language semantics may make the extensions more complex for the user than Modelica is.

Ideal integration for speed and use of Julia tools would be achieved by either design choices 3 or 4. `Sims.jl` represents an early exploration of choice 4, while `Modia.jl` represents a newer, more extensive work within choice 4.[7] Here, we describe the OMJulia API, which belongs to design choice 2. A longer term plan is to improve on the previous OMPython API (Lie et al., 2016), and offer a suite for Python, Julia, and MATLAB.

Based on experience with the OMPython API, the syntax of the OMJulia API is updated/improved for easier use. To be future proof, the tool developer should "own" the API. Ease of maintenance of such a suite is essential, which implies that the syntax should be similar across script languages. Thus, some compromises must be made wrt. syntax. As an example, the key paradigm in Python is objects, and applying method `simulate` to object `mod` would have the syntax `mod.simulate()`. The key paradigms in Julia are types and multiple dispatch ("function overloading"), and the natural syntax in Julia would be `simulate(mod)` where the type of `mod` decides which method/function implementation is used ("dispatching"). Still, Julia allows for the same syntax as Python, and the Pythonian syntax is therefore chosen — for ease of maintenance. Ease of maintenance also dictates that OMJulia should depend on as few packages as possible, and take advantage of existing packages in Julia for plotting, etc.

## 2.3 Description of the API

The API is described in the subsections below.

### 2.3.1 Julia Class and Constructor

The first step to using the OMJulia API is to introduce it in the Julia session using the `using` command:[8]

```
julia> using OMJulia
```

Next, an empty Julia model object is constructed which communicates with OpenModelica:[9]

```
julia> mod = OMJulia.OMCSession()
```

We are now ready to fill the model object with content. The OMJulia method which is used to populate the model object with a Modelica model is the model constructor `ModelicaSystem()`. This constructor requires two arguments, with an optional third argument:

1. The first argument is a string containing the name of the Modelica file which holds the model, if necessary with full directory path.

2. The second argument is a string containing the name of the main Modelica model within the file.

[4] The same paths are possible with other script languages such as Python and MATLAB

[5] http://zeromq.org/

[6] Originally, OMPython used CORBA technology instead of ZMQ

[7] See `www.julialang.org` under `Explore packages`.

[8] The Julia prompt `julia>` is not typed, and does not appear in script files, nor in IJulia/Jupyter notebooks.

[9] Any valid Julia identifier can be used as the model object name.


DOI 10.3384/ecp19157699

3. If the main Modelica model uses some libraries (e.g., the Modelica Standard Library), these are listed as strings in a Julia vector (= 1D array) in a third argument. If a single library is used, the vector of a single string can be replaced by the string.

**Example 1.** Use of Model Constructor

Suppose that we have estabilised a Julia object `mod` which communicates with OpenModelica, see above. Suppose next that we have a Modelica model with name CSTR, wrapped in a Modelica package *Reactors* — stored in file `Reactors.mo`:

```
package Reactors
  // ...
  model CSTR
    /// ...
  end CSTR;
  //
end Reactors;
```

Assuming that no external Modelica code is used, the following Julia code populates the Julia object `mod` with the Modelica model:

```
julia>  mod.ModelicaSystem("Reactors.mo", "
    Reactors.CSTR")
```

▲

### 2.3.2 Methods, Arguments, and Return Values

In the Julia language, it is in general recommended *not* to use class functions ("methods") in the way we have done in OMJulia. Instead of using get and set methods (as in Python), one could operate directly on the object attributes[10]. And instead of using methods that transform the object, e.g., simulate, linearize, etc., one could define general functions combined with type dispatching. However, because OMJulia is part of a family of script language interfaces for OpenModelica, some compromise has been made in order to simplify maintenance. To this end, in OMJulia, "methods" in the sense of object oriented languages a la Python are appended to the object after a dot.[11]

Methods in OMJulia have zero or one argument. In the case of one argument, this is either a Julia string or a vector (= 1D array) of strings.[12] The following Julia syntax is useful in this context:

1. String *concatenation* is achieved by symbol `*`, thus strings `"K"`, `"="`, and `"5"` can be concatenated by `"K"*"="*"5"` to become `"K=5"`.

2. String *substitution* (referred to as string *interpolation* in the Julia community) is achieved by the reserved symbol `$`, e.g., `"T=\$(25+273)"` is interpreted as `"T=298"`, while `T0=298` followed by `"T=\$T0"` or `"T=$(T0)"` gives the same result.[13]

Some methods return a single string `s` holding a numerical value, or a vector `v` holding strings each with a numerical value. Such a string `s` can be trivially converted to a floating point number by `parse(Float64,s)`; such a vector `v` can be converted to a vector of floating point numbers by `[parse(Float64,s) for s in v]`.

In the subsequent overview of methods, object name `mod` is used for illustration — in real use, any valid Julia identifier can be used as object name. Methods may or may not return results — if the methods do not return results, the results are stored within the object.

### 2.3.3 Utility Routines, Converting Modelica ↔ FMU

Two utility methods convert files between Modelica files with file extension `.mo` and Functional Mock-up Unit (FMU) files with file extension `.fmu`.

1. `mod.convertMo2Fmu()` — method for converting the *Modelica model* of the object into an FMU file.

   - Required arguments: *none*, operates on the Modelica file associated with the object.
   - Optional input arguments:
     - `version`: string with FMU version, `"1.0"` or `"2.0"`; the default is `"1.0"`.
     - `fmuType`: string with FMU type, `"me"` (model exchange) or `"cs"` (co-simulation); the default is `"me"`.
     - `fileNamePrefix`: string; the default is `"className"`.
   - Return argument:
     - `generatedFileName`: string, returns the full path + filename of the generated FMU (`.fmu`).

2. `mod.convertFmu2Mo(s)` — method for converting an FMU file into a Modelica file.

   - Required input arguments: string `s`, where `s` holds the name of the FMU file, including extension `.fmu`.

[10] In Julia, operating directly on the object attributes is safe because Julia is a strongly typed language, contrary to, e.g., Python. Safe, assuming that strong type definition has been used.

[11] In Julia, the word *method* as a different meaning than in general object oriented languages. Here, the word "method" is used as in object oriented languages such as Python.

[12] In the OMPython initiative, (Lie et al., 2016), Python's keyword assignment syntax was used. Keyword assignment is, however, troublesome, since possible Modelica identifiers such as `mod.K` and `der(x)` are invalid as identifiers/keywords in Python, Julia, etc.

[13] With `$` being a reserved symbol in Julia, it is necessary to use the escape character `\`, i.e., `\$` to achieve the effect of character `$` in strings, e.g., to specify LaTeX typesetting. Alternatively, by using Julia package `LaTeXString`, syntax `L"..."` replaces `$` with `\$` in the string without user intervention.

- Optional input arguments: a number of optional input arguments, e.g., the possibility to change working directory for the imported FMU files.
- Return argument:
  - `generatedFileName`: string, returns the full path + filename of the generated Modelica file (`.mo`).

#### 2.3.4 Get and Set Information

Several methods are dedicated to getting and setting information about objects. With two exceptions — `getQuantities()` and `getSolutions()` — the get methods have identical use of arguments and results, while all the set methods have identical use of input arguments, with results stored in the object.

**Get Quantity Information. Show Quantity Information** Method `mod.getQuantities()` has no input arguments, and returns a vector[14] of dictionaries, one dictionary for each quantity. Each dictionary has the following keys (strings) — with values being strings, too.

- `"name"` — the name of the quantity, e.g., `"T"`, `"der(T)"`, `"n[1]"`, `"mod1.T"`, etc.,
- `"aliasvariable"` — typically `nothing`,
- `"variability"` — typically `"continuous"`, `"parameter"`, etc.,
- `"changeable"` — value `"true"` or `"false"`,
- `"causality"` — value `"internal"` or `"external"` (for inputs),
- `"value"` — string of number `"50"`, text string, or `"None"`,
- `"description"` — string copied from Modelica: description of the quantity, e.g. `"Mass in tank, kg"`, or `nothing`.
- `"alias"` — typically `"noAlias"`.

Modelica *constants* are not included in the returned vector of dictionaries.[15]

A Julia specific utility function `mod.showQuantities()` is included with the same syntax as `mod.getQuantities()`, taking advantage of Julia `DataFrames` to present the quantities in a table.[16]

**Get Solutions** We consider method `getSolutions()` — which assumes that the `simulate()` method has been applied (see below). Three calling possibilities are accepted.

- `mod.getSolutions()`, i.e., without input arguments, returns a *vector* of strings of *names* of quantities for which there is a solution.[17]
- `mod.getSolutions(s)`, where `s` is a *string* of a name, returns a single time series (= *vector* of floating point numbers) for the corresponding name.
- `mod.getSolutions(v)`, where `v` is a *vector* of strings of names, returns a *vector* of time series (= *vectors* of floating point numbers) for the corresponding names.

It follows that a vector of all time series can be returned by the construct `mod.getSolutions(mod.getSolutions())`.

**Standard Get Methods** We consider methods `getXXX()`, where `XXX` is either of `{Continuous, Parameters, Inputs, Outputs, SimulationOptions, LinearizationOptions}`. Thus, methods `mod.getContinuous()`, `mod.getParameters()`, etc. Three calling possibilities are accepted.

- `mod.getXXX()`, i.e., without input argument, returns a *dictionary* with names (strings) as keys and values given in strings.
- `mod.getXXX(s)`, where `s` is a *string* of a name, returns a single string with value of the corresponding name.
- `mod.getXXX(v)`, where `v` is a *vector* of strings of names, returns a *vector* of strings of values for the corresponding names.

**Set Methods** The information that can be set is a subset of the information that can be get. Thus, we consider methods `setXXX()`, where `XXX` is either of `{Parameters, Inputs, SimulationOptions, LinearizationOptions}`, thus methods `mod.setParameters()`, `mod.setInputs()`, etc. Two calling possibilities are accepted.

- `mod.setXXX(s)`, with `s` being a string of keyword assignments of type quantity `"name = value"`. Here, the quantity `name` could be a parameter name, an input name, etc.
  - For parameters and simulation/linearization options, the value should be a single value such as a numerical value or a name of a solver, etc., e.g., `s` is `"R=8.31"` or `"solver=dassl"`.

[14] In Julia, a *vector* is a 1D array.

[15] In Modelica, **`constant`** is used for values which require recompilation when changed. **`parameter`** values, on the other hand, can be changed without recompilation.

[16] In Python, `mod.showQuantities()` is redundant because the return object directly produces a table with Python `pandas`.

[17] The reason why a dictionary with every name as key and time series as value is not returned, is that the amount of data might be exhaustive.


DOI 10.3384/ecp19157699

  - For inputs, the value could be a numerical value if the input is constant in the time range of the simulation, e.g., `"u = 1.0"`, or
  - For inputs, the value could alternatively be a vector of tuples `(t_{j},u_{j})`, i.e., `[(t1,u1),(t2,u2),...,(tN,uN)]` where the input varies linearly between `(t_{j},u_{j})` and `(t_{j+1},u_{j+1})`, where `t_{j}<=t_{j+1}`, and where at most two subsequent time instances `t_{j},t_{j+1}` can have the same value. As an example, `"u=[..., (1,10), (1,20), ...]"` describes a perfect jump in input value from value `10` to value `20` at time instance `1`.

- `mod.setXXX(v)`, with `v` being a vector of strings as described for `mod.setXXX(s)`. An example could be could be `["R=8.31","cp=4.18"]`.

#### 2.3.5 Operating on Julia Object: Simulation

The following method operates on the object, and has no input arguments. The method has no return values; instead the results are stored within the object.

- `mod.simulate()` — simulates the system with the given simulation options

To retrieve the results, method `mod.getSolutions()` is used as described previously.

#### 2.3.6 Operating on Julia Object: Linearization

The following methods are used for linearization:

- `mod.linearize()` — with no input argument, returns a tuple of 2D arrays (matrices) $A$, $B$, $C$, $D$.
- `mod.getLinearInputs()` — with no input argument, returns a vector of strings of names of inputs used when forming matrices $B$ and $D$.
- `mod.getLinearOutputs()` — with no input argument, returns a vector of strings of names of outputs used when forming matrices $C$ and $D$.
- `mod.getLinearStates()` — with no input argument, returns a vector of strings of names of states used when forming matrices $A$, $B$, $C$, $D$.

Observe that linearization is carried out at the `stopTime` specified in `LinearizationOptions`. The reason why linearization is not carried out at initial time, is that to handle DAEs, OpenModelica needs to initialize the model at initial time — before linearization can be carried out. For normal use, `stopTime` should be given a small value if linearization at the current operating value is intended.

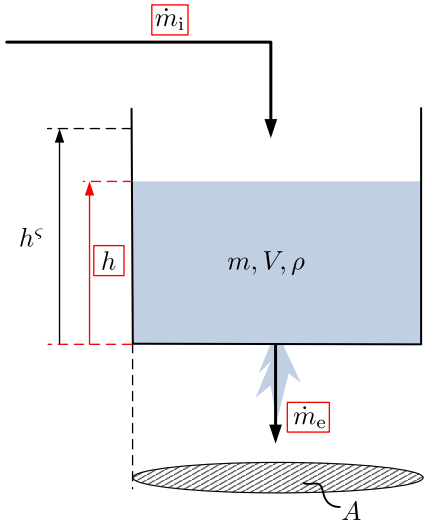


**Figure 1.** Driven water tank, with externally available quantities framed in red: initial mass is emptied through bottom at rate $\dot{m}_\mathrm{e}$, while at the same time water enters the tank at rate $\dot{m}_\mathrm{i}$.

#### 2.3.7 Operating on Julia Object: Sensitivity

Sensitivy is related to $\frac{\partial y(t)}{\partial \theta}$, i.e., how an infinitesimal change in a parameter $\theta$ leads to an infinitesimal change in the solution of variable $y$; both $\theta$ and $y$ can in principle be vectors. Sensitivity is very important in connection with model fitting and identifiability analysis. The following method is implemented on the Julia side, and provides *numeric* sensitivities. The method has 2 or 3 input arguments, and returns a tuple of 2 return arguments.

- `mod.sensitivity(a1,a2[,a3])` — computes sensitivity $\frac{\partial y(t)}{\partial \theta}$. Input arguments must be vectors: `a1` holds strings of the name of model parameters ($\theta$), `a2` holds strings of the name of system variables ($y$), while the optional third argument `a3` holds floating point values for fractional parameter perturbation. The return tuple holds two vectors, `r1` and `r2`. The first vector, `r1`, holds strings of the name of the sensitivities that have been computed, while vector `r2` holds the corresponding time series (vector of solution values) — computed at the time instances given by the simulation options.

# 3 Basic Use of API for Model Analysis

## 3.1 Case: Simple Tank Filled with Liquid

We consider the tank in Figure 1 filled with water.

Water with initial mass $m(0)$ is emptied by gravity through a hole in the bottom at effluent mass flow rate $\dot{m}_\mathrm{e}$, while at the same time water is filled into the tank at influent mass flow rate $\dot{m}_\mathrm{i}$. Our modeling objective is to find the liquid level $h$. Here, the input variable is the influent mass flow rate $\dot{m}_\mathrm{i}$, while the output variable is the quantity we are interested in, $h$.

**Table 1.** Parameters for driven tank with constant cross sectional area.

| *Parameter* | *Value* | *Comment* |
|---|---|---|
| $\rho$ | $1\,\mathrm{kg/L}$ | Density of liquid |
| $A$ | $5\,\mathrm{dm}^2$ | Constant cross sectional area |
| $K$ | $5\,\mathrm{kg/s}$ | Valve constant |
| $h^{\varsigma}$ | $3\,\mathrm{dm}$ | Level scaling |

**Table 2.** Operating condition for driven tank with constant cross sectional area.

| *Quantity* | *Value* | *Comment* |
|---|---|---|
| $h(0)$ | $1.5\,\mathrm{dm}$ | Initial level |
| $m(0)$ | $\rho h(0) A$ | Initial mass |
| $\dot{m}_\mathrm{i}(t)$ | $2\,\mathrm{kg/s}$ | Nominal influent mass flow rate; may be varied |

## 3.2 Model Summary

The model can be summarized in a form suitable for implementation in Modelica as

$$\frac{dm}{dt} = \dot{m}_\mathrm{i} - \dot{m}_\mathrm{e} \tag{1}$$

$$m = \rho V \tag{2}$$

$$V = Ah \tag{3}$$

$$\dot{m}_\mathrm{e} = K\sqrt{\frac{h}{h^{\varsigma}}} \tag{4}$$

To complete the model description, we need to specify model parameters and operating conditions. Model parameters (constants) are given in Table 1.

The operating conditions are given in Table 2.

## 3.3 Modelica Encoding of Model

The Modelica code describes the core model of the tank, `ModWaterTank`, and consists of a *first section* where constants and variables are specified, and a *second section* where the model equations are specified (compactified Modelica code is shown below).

```
model ModWaterTank
   constant Real rho = 1 "Density";
   parameter Real A=5, K=5, h_s=3;
   parameter Real h_0=1.5, m_0=rho*h_0*A;
   Real m(start=m_0, fixed=true);
   Real V, md_e;
   input Real md_i;
   output Real h;
equation
   der(m)=md_i-md_e;
   m=rho*V, V=A*h, md_e=K*sqrt(h/h_s);
end ModWaterTank;
```

As seen from the *first section* of model `ModWaterTank`, the model has 4 essential parameters (`rho`–`h_s`) of which one is a Modelica constant (`rho`) while other 3 are design parameters, compare this to Table 1. Furthermore, the model contains 2 "initial state" parameters, where 1 of them can be chosen at liberty, `h_0`, while the other one, `m_0`, is computed automatically from `h_0`, see Table 2. The purpose of the "free parameter" `h_0` is that it is easier for the user to specify level than mass. Also, free "initial state" parameters makes it possible for the user to change the initial states from outside of model `ModWaterTank`, e.g., from Julia.

Next, one variable is given with initial value — the state `m` — is initialized with the "initial state" parameter `m_0`. Then, 2 variables are defined as auxiliary variables (algebraic variables), `V` and `md_e`.[18]

One input variable is defined — `md_i` — this is the influent mass flow rate $\dot{m}_\mathrm{i}$, see Table 2. Inputs are characterized by that their values are not specified in the core model — here `ModWaterTank`. Instead, their values must be given in an external model/code — we will specify this input in Julia. Finally, 1 output is given — `h`.

In the *second section* of model `ModWaterTank`, the Model equations exactly map the mathematical model given in Eqs. 1–4. For illustrative purposes, the core model `ModWaterTank` is wrapped within a package named `WaterTank` and stored in file `WaterTank.mo`,

```
package WaterTank
// Package for simulating
   // driven water tank
   model ModWaterTank
      // Main driven water tank model
      // ...
      ...
   end ModWaterTank;
// End package
end WaterTank;
```

## 3.4 Use of Julia API

First, the following Julia statements are executed — we did this in Jupyter notebook (IJulia).

```
using Plots; pyplot()
using LaTeXStrings
using DataFrames
using OMJulia
# Linewidth
LW1 = 1.5
LW2 = 1
# Colors - core
usn_red = colorant"#D64349"
usn_blue = colorant"#27B2DO"
usn_green = colorant"#3BAFA2"
usn_purple = colorant"#4646A5"
usn_gold = colorant"#FFD240"
```

Here, package `Plots` is the plotting meta package of Julia; we use `pyplot` as back-end. Package `LaTeXStrings` makes it possible to automate insertion of escape symbol `\` in LaTeX code to produce proper Julia strings. Package `DataFrames` is used to present `quantities` in Jupyter notebook tables. Two line widths

[18] `md` is notation for `m` with a dot, $\dot{m}$, i.e., a mass flow rate.


DOI 10.3384/ecp19157699

Out[57]:

| | causality | value | description | name | changeable | aliasvariable | variability | alias |
|---|---|---|---|---|---|---|---|---|
| 1 | internal | None | Mass in tank, kg | m | false | nothing | continuous | noAlias |
| 2 | internal | None | der(Mass in tank, kg) | der(m) | false | nothing | continuous | noAlias |
| 3 | output | None | Tank liquid level, dm | h | false | nothing | continuous | noAlias |
| 4 | internal | None | Effluent mass flow rate from tank, kg/s | md_e | false | nothing | continuous | noAlias |
| 5 | input | None | Influent mass flow rate to tank, kg/s | md_i | true | nothing | continuous | noAlias |
| 6 | internal | 5.0 | Cross sectional area of tank, dm2 | A | true | nothing | parameter | noAlias |
| 7 | internal | 5.0 | Valve constant, kg/s | K | true | nothing | parameter | noAlias |
| 8 | internal | 1.5 | Initial tank level, dm | h_0 | true | nothing | parameter | noAlias |
| 9 | internal | 3.0 | Scaling level, dm | h_s | true | nothing | parameter | noAlias |
| 10 | internal | None | Initial tank mass, kg | m_0 | false | nothing | parameter | noAlias |
| 11 | internal | None | Tank liquid volume, L | V | false | m | continuous | alias |

**Figure 2.** Typesetting of quantity vector of dictionaries as a table in a Jupyter notebook.

are assigned, to variables `LW1` and `LW2`, to obtain uniform line width.

Colors are taken from the graphical profile of the employer of first author are used to illustrate how one can define colors using `hex` code. Alternatively, the CSS color names are available[19] as case insensitive symbols, e.g., `:red`, `:cornflowerblue`, etc.

## 3.5 Basic Simulation of Model

We instantiate object tank with the following command:

```
tnk = OMJulia.OMCSession()
tnk.ModelicaSystem("WaterTank.mo","WaterTank.
    ModWaterTank")
```

In the sequel, Julia prompt `julia>` is used when Jupyter[20] notebook actually uses `In[*]` — where `*` is some number, while the response in Jupyter notebook is prepended with `Out[*]` .

```
julia> q = tnk.getQuantities()
julia> typeof(q)
Array{Any,1}
julia> length(q)
11
julia> q[1]
Dict{Any,Any} with 8 entries:
  "name"           => "m"
  "aliasvariable"  => nothing
  "variability"    => "continuous"
  "changeable"     => "false"
  "causality"      => "internal"
  "value"          => "None"
  "description"    => "Mass in tank, kg"
  "alias"          => "noAlias"
julia> tnk.showQuantities()
```

Method `tnk.showQuantities()` produces a table overview, Fig. 2.

The results in Figure 2 should be compared to the Modelica model in Section 3.2. Observe that Modelica constants are not included in the quantity list.

Next, we check the simulation options:

```
julia> tnk.getSimulationOptions()
Dict{Any,Any} with 5 entries:
  "startTime" => "0"
  "stopTime"  => "1"
  "solver"    => "dassl"
  "stepSize"  => "0.002"
  "tolerance" => "1e-006"
```

It should be observed that the `stepSize` is the frequency at which solutions are *stored*, and is *not* the step size of the solver. The number of data points stored, is thus (`stopTime`−`startTime`)/`stepSize` with due rounding. This means that if we increase the `stopTime` to a large number, we should also increase the `stepSize` to avoid storing large amounts of data.

Possible inputs are:

```
julia> tnk.getInputs()
Dict{Any,Any} with 1 entry:
  "md_i" => "None"
```

where value `None` implies that the available input, `md_i`, has yet not been set. The simulation will not work with value `None`; let us instead set $\dot{m}_\mathrm{i} = 3$, simulate for a long time, and then change "initial state" parameter `h(0)` to the steady state value of `h`:

```
julia> tnk.setInputs("md_i=3")
julia> tnk.setSimulationOptions(["stopTime=1
    e4", "stepSize=10"])
julia> tnk.simulate()
julia> h, = tnk.getSolutions("h")
julia> tnk.setParameters("h_0=$(h[end])")
```

Observe that the syntax `h,` is needed to unpack the time series for `h` when the vector of solutions has a single element.

Next, we reset the stop time to 10, and specify an input sequence with a couple of jumps:

```
julia> tnk.setSimulationOptions(["stopTime=10
    ","stepSize=0.02"])
julia> tnk.setInputs("md_i = [(0,3),(2,3),
    (2,4),(6,4),(6,2),(10,2)]")
```

Finally, we simulate the model with the time varying input, and plot the result:[21]

```
julia> tnk.simulate()
julia> tm, h = tnk.getSolutions(["time","h"])
julia> plot(tm,h,linewidth=LW1, color=
    usn_blue, label=L"$h$")
julia> plot!(title="Water tank level")
julia> plot!(xlabel=L"time $t$ [s]")
julia> plot!(ylabel=L"$h$ [dm]")
```

The result is displayed in Figure 3.

## 3.6 Monte Carlo Simulation

It is of interest to study how the model behavior varies with varying uncertain parameter values, e.g., the effluent valve constant $K$. This can be done as follows:

[19] `https://www.w3schools.com/colors/colors_groups.asp`

[20] Jupyter is denoted `IJulia` in Julia.

[21] `plot()` plots a result, `plot!()` overlays information on an existing plot.

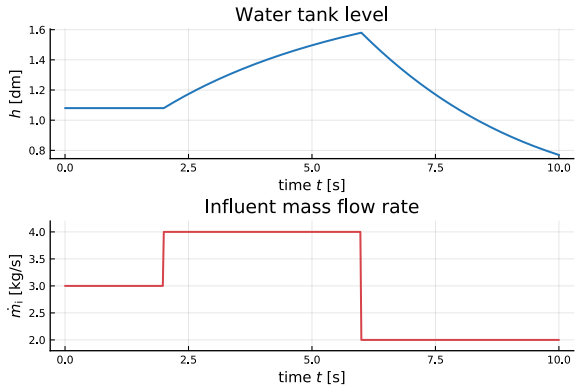


**Figure 3.** Tank level when starting from steady state, and $\dot{m}_\mathrm{i}(t)$ varies in a straight line between the points $(t_j, \dot{m}_\mathrm{i}(t_j))$ given by the list `[(0,3),(2,3),(2,4),(6,4),(6,2),(10,2)]`.

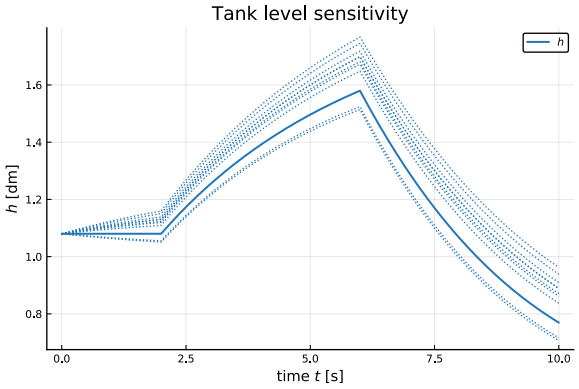


**Figure 4.** Uncertainty in tank level with a 5% uncertainty in valve constant $K$. The input is like in Figure 3.

```
julia> par = tnk.getParameters()
julia> K = parse(Float64,par["K"])
julia> Nmc = 10;
julia> KK = K + (randn(Nmc)-0.5)*K/20;
julia> tnk.simulate()
julia> tm, h = tnk.getSolutions(["time","h"])
julia> v_h = Vector{Vector}(Nmc)
julia> for (i,K) in enumerate(KK)
         tnk.setParameters("K=$(K)")
         tnk.simulate()
         v_h[i] = tnk.getSolutions("h")
       end
julia> plot(tm,h,lw=LW1,lc=usn_red,label=L"
    $h$")
julia> plot!(tm,v_h,lw=LW2,ls=:dot,lc=usn_red
    ,legend=false)
julia> plot!(title="Tank level sensitivity")
julia> plot!(xlabel=L"time $t$ [s]")
julia> plot!(ylabel=L"$h$ [dm]")
```

The result is as shown in Figure 4.

### 3.7 Linearizing Model

We can find a linearized approximation of the system. First we reset $K$ to 5, then set the stop time of the linearization to $10^{-6}$ before we linearize the system and extract matrices $A$, $B$, $C$, and $D$.

```
julia> tnk.setParameters("K=5")
julia> tnk.setLinearizationOptions("stopTime
    =1e-6")
julia> A,B,C,D = tnk.linearize();
```

If we use the Julia `ControlSystems` package,[22] we can define an LTI system and find the transfer function:

```
julia> using ControlSystems
julia> sys = ss(A,B,C,D)
julia> tf(sys)
ControlSystems.TransferFunction{
    ControlSystems.SisoRational{Float64}}
          0.2
--------------------------
1.0*s + 0.2587650960551352

Continuous-time transfer function model
```

We may also like to know the state which OpenModelica has chosen:

```
julia> tnk.getLinearStates()
1-element Array{Any,1}:
 "m"
```

## 4 Case study: PI control of reactor

### 4.1 Reactor

We consider an extension of a reactor described in (Seborg et al., 2011); see (Sund et al., 2018), (Khalili and Lie, 2018) for details of the model and linearization of the model. The reactor is exothermal with water cooling via a heat exchanger, and is unstable at the operating point. The original model (`org`) in (Seborg et al., 2011) has 2 states: reactor temperature $T$ and concentration $c_\mathrm{A}$ of species A. An extended model which only assumes ideal solution (`is`) has 3 states: the states of the `org` model as well as concentration $c_B$ of species B. Both models exhibit nonlinear oscillations when forced away from the equilibrium point. A possible control problem is to control the reactor temperature $T$ by means of the cooling water temperature $T_\mathrm{c}$ of the heat exchanger.

### 4.2 PI Controller

A linearized model can easily be found by using the `mod.linerize()` method of `OMJulia` — the linearized model is as in (Khalili and Lie, 2018), with cooling temperature $T_\mathrm{c}$ as control input. The closed loop matrix $A_\mathrm{cl}$ with a proportional controller (P controller) is

$$A_\mathrm{cl} = A - K_\mathrm{p} BC \tag{5}$$

where $B$ is the input matrix and $K_\mathrm{p}$ is the controller gain. Looping through $K_\mathrm{p} \in [-1, 8]$ leads to the closed loop eigenvalues as depicted in Figure 5.[23]

[22] A similar tool in Python is limited in scope, and rather complicated to install.

[23] Here, Julia's `ControlSystems` package has been used, together with a user-modified `rlocus()` function.


DOI 10.3384/ecp19157699

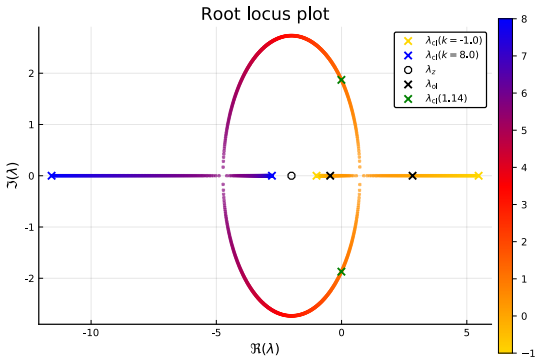


**Figure 5.** Root locus plot $\lambda\left(A_{\mathrm{cl}}; K_{\mathrm{p}}\right)$ for $K_{\mathrm{p}} \in [-1, 8]$.

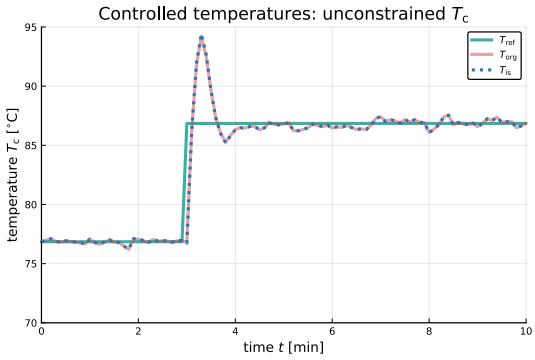


**Figure 6.** Output $T$ as controlled with PI controller tuned for `org` model, and applied to `org` and `is` model.

The P-controller stabilizes the system for $K_{\mathrm{p}} \gtrapprox 1.14$; $K_{\mathrm{p}} = 5.7$ gives two real, closed loop eigenvalues/poles at approximately $\lambda \approx -5$, which implies closed loop time constants $\tau_j \approx \frac{1}{5} = 0.2$.

For a proportional + integral controller, it is reasonable to let the reset time (= integral time) be, say, 10 times larger than the closed loop time constants of a P controller. Thus, the PI controller

$$T_{\mathrm{c}}(s) = T_{\mathrm{c}}^{*} + K_{\mathrm{p}} \frac{1 + T_{\mathrm{int}} s}{T_{\mathrm{int}} s} \cdot e(s) \tag{6}$$

$$e(s) = T_{\mathrm{ref}}(s) - T(s) \tag{7}$$

with $K_{\mathrm{p}} = 5.7$ and $T_{\mathrm{int}} = 2$ may be an acceptable choice.[24] Nominal input $T_{\mathrm{c}}^{*}$ is not needed with integral action, but is useful to avoid an initial "kick" in the control action. $T_{\mathrm{ref}}$ is the reference temperature. If we let $T_{\mathrm{int}} \to \infty$, the controller becomes a P controller.

In the time domain, we can express the PI controller as

$$T_{\mathrm{c}} - T_{\mathrm{c}}^{*} = K_{\mathrm{p}} e + \tilde{T}_{\mathrm{c}} \tag{8}$$

$$\frac{d\tilde{T}_{\mathrm{c}}}{dt} = \frac{K_{\mathrm{p}}}{T_{\mathrm{int}}} e. \tag{9}$$

To handle constraints for $T_{\mathrm{c}} \in [4, 96]\,^{\circ}\mathrm{C}$, if $T_{\mathrm{c}} = K_{\mathrm{p}} e + T_{\mathrm{c}}^{*} + \tilde{T}_{\mathrm{c}}$ breaks this constraint, we set $T_{\mathrm{c}}$ equal to the constraint and $\frac{d\tilde{T}_{\mathrm{c}}}{dt} = 0$ to avoid controller wind-up. The controller is implemented in Modelica, but controller parameters and constraints in $T_{\mathrm{c}}$ are set from Julia using OMJulia.

### 4.3 Proportional + Integral Control

Figure 6 shows the use of a PI controller to keep reactor temperature $T$ close to a reference $T_{\mathrm{ref}}$. The PI controller tuned for the `org` model, is also applied to the `is` model.

The result indicates that the controller easily handles the model difference between the two models. Figure 7 shows the applied control input $T_{\mathrm{c}}$ as well as the integral state $\bar{T}_{\mathrm{c}}$ in the controller for the two model cases.

Figure 7 clearly shows a problem for the controller: the cooling water can not take on negative temperatures $T_{\mathrm{c}}$

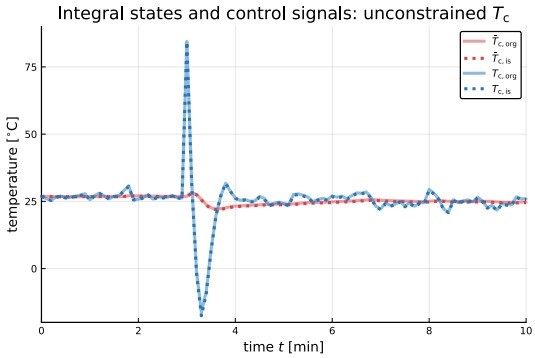


**Figure 7.** PI control signal $T_{\mathrm{c}}$ and integrator state $\bar{T}_{\mathrm{c}}$ for `org` and `is` models.

[°C]. We therefore add the constraint that $T_{\mathrm{c}} \in [4, 96]\,^{\circ}\mathrm{C}$, which together with anti-windup leads to the results in Figures 8 and 9 for output $T$ and controller $T_{\mathrm{c}}$, respectively.

## 5 Discussion and Conclusions

This paper presents OMJulia, a first effort to interface a relatively complete Modelica tool, OpenModelica, to Julia, giving access to an open source set-up for modeling and analysis, including control synthesis, easily installable from a unified package manager.

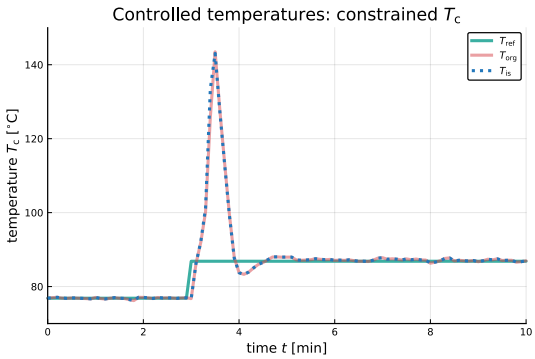


**Figure 8.** Output $T$ as controlled with PI controller tuned for `org` model, and applied to `org` and `is` model: control input $T_{\mathrm{c}}$ is constrained to $[4, 96]\,^{\circ}\mathrm{C}$ and anti-windup is applied.

[24] The integral time is denoted $T_{\mathrm{int}}$ in order to make a distinction between integral time and influent temperature, $T_{\mathrm{i}}$.

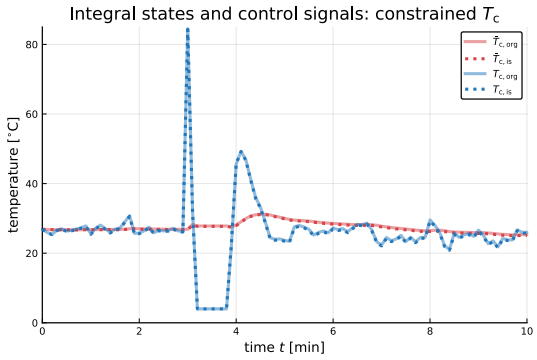


**Figure 9.** PI control signal $T_c$ and integrator state $\bar{T}_c$ for `org` and `is` models: control input $T_c$ is constrained to $[4,96]$ °C and anti-windup is applied.

Some design choices of the Julia API are briefly described, and the syntax and possibilities of OMJulia are then detailed. The use of the API is illustrated with a simple example of a water tank model, then some possibilities for control analysis of a chemical reactor are detailed. The API has also been tested on more complex models not shown here.

The key contribution of the OMJulia package is not within Modelica as a language, but rather to increase the usefulness of Modelica into new fields such as control engineering. Future work will include a package OMMatlab, updating the syntax of OMPython, and possibly extension of the API to the optimization and symbolic sensitivity analysis routines in OpenModelica. Another possibility is to consider a translator from OpenModelica to Julia (design choice 3).

# References


Marcus Baur, Martin Otter, and Bernhard Thiele. Modelica Libraries for Linear Control Systems. In *Proceedings, the 7th International Modelica Conference*, Como, Italy, 2009.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Sha. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 49(1):65–98, 2017. doi:10.1137/14100067.

K. E. Brenan, S. L. Campbell, and Linda R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. North-Holland, New York, 1989.

Hilding Elmqvist, Toivo Henningsson, and Martin Otter. Innovations for Future Modelica. In *Proceedings of the 12th International Modelica Conference*, Prague, Czech Republic, May 2017. doi:10.3384/ecp17132693. May 15-17, 2017, Prague, Czech Republic.

FMI Consortium. Functional Mock-up Interface for Model Exchange, version 2.0, 2018. URL `https://fmi-standard.org/`.

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.

Peter Fritzson, Adrian Pop, Adeel Asghar, Bernhard Bachmann, Willi Braun, Robert Braun, Lena Buffoni, Francesco Casella, Rodrigo Castro, Alejandro Danós, Rüdiger Franke, Mahder Gebremedhin, Bernt Lie, Alachew Mengist, Kannan Moudgalya, Lennart Ochel, Arunkumar Palanisamy, Wladimir Schamai, Martin Sjölund, Bernhard Thiele, Volker Waurich, and Per Östlund. The OpenModelica Integrated Modeling, Simulation and Optimization Environment. In *Proceedings of the 1st American Modelica Conference*, Cambridge, MA, USA, October 2018. LIU Electronic Press, www.ep.liu.se. October, 8-10, 2018.

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

Anand Kalaiarasi Ganeson, Peter Fritzson, Olena Rogovchenko, Adeel Asghar, Martin Sjölund, and Andreas Pfeiffer. An OpenModelica Python Interface and its Use in PySimulator. In *Proceedings of the 9th International Modelica Conference*, September 2012. doi:10.3384/ecp12076537. September 3-5 2012.

Pieter Hintjens. *ZeroMQ. Messaging for Many Applications*. O'Reilly Media, March 2013.

JuliaLang. The Julia Programming Language, 2018. URL `https://julialang.org/`.

Mohammad Khalili and Bernt Lie. Comparison of Linear Controllers for Nonlinear, Open-loop Unstable Reactor. In *Proceedings, SIMS 2018*, Oslo Metropolitan University, September 2018. SIMS, Linköping University Press.

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, 2016, September 2016.

Modelica Association. The Modelica Standard Library, v. 3.2.2, 2016. URL `https://github.com/modelica/ModelicaStandardLibrary/`.

Modelica Association. Modelica® — a Unified Object Oriented Language for System Modeling Language Specification, version 3.4, 2017. URL `https://modelica.org/documents/ModelicaSpec34.pdf`.

Christopher Rackauckas and Qing Nie. DifferentialEquations.jl — A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *Journal of Open Research Software*, 5(15), 2017. DOI: http://doi.org/10.5334/jors.151.

Dale E. Seborg, Thomas F. Edgar, Duncan A. Mellichamp, and III Doyle, Frank J. *Process Dynamics and Control*. John Wiley & Sons, Hoboken, NJ, third edition edition, 2011. ISBN 978-0-470-12867-1. ISBN 978-0-470-12867-1.

Sveinung M. Sund, Marianne Plouvier, and Bernt Lie. Comparison of Simulation Tools for Dynamic Models. In *Proceedings, SIMS 2018*, Oslo Metropolitan University, September 2018. SIMS, Linköping University Press.