Comparison of Simulation Tools for Dynamic Models

Sveinung M. Sund[1], Marianne Plouvier[2], and Bernt Lie[1]

[1]University of South-Eastern Norway, Bernt.Lie@usn.no, [2]IMT Mines Albi, France

Abstract

Macroscopic models are used extensively in process engineering, and can often be posed as DAE (Differential Algebraic Equation) models. Three generic tools for solving such DAEs are compared: OpenModelica, Julia, and MATLAB. To make the comparison concrete, a simple non-linear process model from the literature was extended by removing simplifying assumptions; the more complex model was posed as DAEs. Some implementation details of DAE models in OpenModelica, Julia, and MATLAB are given. Selected simulation results are given, with resulting execution time. The three tools gave identical simulation results. The tools are then compared wrt. cost, ease of use, documentation, numeric quality, Eco-system , and possibility for reuse of models/library. Overall, Julia appears may appear as the best choice. However, Modelica is found to be easier to use, so an ideal solution would probably be some tight integration of Modelica with Julia.
Keywords: process modeling, dynamics, DAEs, simulation tool.

1 Introduction

1.1 Background

Macroscopic models are used extensively in process engineering, and can in general be posed as a set of differential equations stemming from conservation/balance laws with the addition of algebraic relations describing thermodynamics, transport laws, reaction engineering, etc. The result is a set of Differential Algebraic Equations (DAE) with a relatively simple structure. Such DAE models can be transformed/reduced to Ordinary Differential Equations (ODEs) with some effort. The resulting ODEs are often simpler than the DAEs, thus more efficient solvers may be available. At the same time, reducing DAEs to ODEs also eliminates many variable which may be of interest to study.

In education, as well as in engineering practice, it is important to choose simulation tools for doing experiments on such models. Important aspects are, e.g.: cost, ease of use, how well the tool is documented, numeric quality, Eco-system (packages/functions for plotting, analysis, random numbers, control packages, access to time series database, etc.), possibility of reuse of the model (model library, extract information for control design, etc.).

Dynamic models are used for design and operation of systems, and it is therefore of interest to fit such models to experimental data, to reuse such models for process design, stability analysis, control design, estimator design, etc. The various simulation tools have different support for and capabilities wrt. analysis and synthesis. An important question is then: is it wise to aim at a one-language/tool solution, or is it better to use different tools for simulation and design/synthesis as long as the languages can interact?

1.2 Previous Work

Specialized process engineering tools are well developed, e.g., from AspenTech[1], Process Systems Enterprise[2], etc. More generic tools are also popular, such as Modelica based tools (OpenModelica[3], JModelica.org[4], Dymola[5], etc.), other high level tools (DAE Tools[6], APMonitor[7]), script based tools (MATLAB[8], Python[9], Julia[10], etc.), and computer algebra systems (Mathematica[11], Maple[12], etc.). The possibility to integrate OpenModelica with Python to enable more extended analysis was studied in (Lie et al., 2016).

A simple model of a nonlinear, open loop unstable reactor is given in (Seborg et al., 2011), and is used in various control studies (Henson and Seborg, 1997). In a student project/exam paper at University College of Southeast Norway[13], this model has been extended into a DAE model by removing some of the simplifying assumptions. Both of these models are suitable for testing basic capabilities of simulation tools for handling small-scale models.

1.3 Organization of Paper

In this paper, we compare the suitability of selected tools (OpenModelica, Julia, MATLAB) for solving macroscopic DAE models in process engineering. The paper

[1] www.aspentech.com
[2] www.psenterprise.com/products/gproms
[3] https://openmodelica.org/
[4] www.jmodelica.org
[5] www.3ds.com/products-services/catia/products/dymola
[6] http://daetools.com
[7] http://apmonitor.com
[8] https://mathworks.com
[9] www.anaconda.com
[10] https://julialang.org
[11] www.wolfram.com
[12] www.maplesoft.com
[13] Course FM1015 Modelling of Dynamic Systems, www.usn.no/academics/study-and-subjectplans/#/subjects/FM1015_2018H_1

https://doi.org/10.3384/ecp18153177

Proceedings of The 59th Conference on Simulation and Modelling (SIMS 59), 26-28 September 2018, Oslo Metropolitan University, Norway

is organized as follows. In Section 2, an overview of a suitable DAE model structure is given, with details of the case study model. In Section 3, some general characteristics and specific implementation details of the models in the chosen languages are given. In Section 4, selected simulation results are given, together with some resulting execution times. In Section 5, the findings are discussed, before some conclusions are drawn in Section 6. An appendix gives some model details.

2 Model Overview

2.1 Mechanistic Model Structure

Mechanistic models at the macroscopic level can typically be described using balance laws of form

$$\frac{dx}{dt} = f(x,z,u;\theta) \tag{1}$$

where differential variable x is the "balanced" property (amount, momentum, energy), while z is some auxiliary algebraic variable, u is some input variable and θ is some model constant/parameter. Such models need to be complemented with transport/thermodynamic/reaction laws (constitutive laws) of form

$$0 = g(x,z,u;\theta). \tag{2}$$

Together, Eqs. 1 and 2 form a set of differential-algebraic equations, DAE. The set of differential variables and algebraic variables is referred to as the model *descriptor*, (x,z). Such DAE models are conceptually simple to formulate, but may contain relatively many descriptor variables and many parameters θ.

DAE models can be manipulated into sets of ordinary differential equations, ODE, of form

$$\frac{d\tilde{x}}{dt} = \tilde{f}(\tilde{x},u;\tilde{\theta}) \tag{3}$$

where $\tilde{x}$ is a state of the system. For DAE models of index 1, $\dim x \geq \dim \tilde{x}$, while for DAE models of index larger than 1, it is possible that $\dim x \leq \dim \tilde{x}$. Normally $\dim \theta \geq \dim \tilde{\theta}$.

Tools for analyzing ODE models (solvers, stability concepts, etc.) are better developed than tools for DAE models. ODE models are smaller and faster to solve, but manual model simplification may introduce (model) errors. DAE models, on the other hand, hold more information (the algebraic variables), and are simpler to formulate, so there are important advantages in keeping models in DAE form.

It is of interest to compare how easy it is to formulate DAE models and solve them in popular modeling/simulation tools. For such a comparison, it is convenient to introduce a case study.

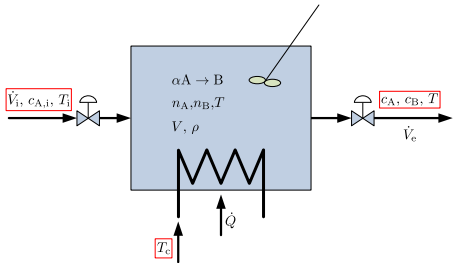

Figure 1. Cooled liquid reactor with reaction $a\mathrm{A} \rightarrow \mathrm{B}$.

2.2 Process Overview

We consider a liquid reactor of constant volume V, with influent volumetric flow rate $\dot{V}_\mathrm{i}$, influent concentration $c_\mathrm{A,i}$ of reactant A carried via an inert solvent S,[14] and influent temperature T_i, Fig. 1.

It is of interest to convert species A into species B through an exothermic reaction

$$a\mathrm{A} \rightarrow \mathrm{B}; \tag{4}$$

the products are carried out of the reactor via solvent S. To keep control of reactor temperature T, heat rate $\dot{Q}$ is added by flowing a liquid at temperature T_c through the tube side of a coil/heat exchanger. With a high flow rate of the cooling liquid, T_c is constant through the heat exchanger, and the heat rate can be described as

$$\dot{Q} = \mathscr{U}A(T_\mathrm{c} - T). \tag{5}$$

where $\mathscr{U}A$ is a parameter. If $T_\mathrm{c} < T$, $\dot{Q} < 0$ and the reactor is cooled.

The rate of reaction r is given as

$$r = kc_\mathrm{A}^a \tag{6}$$

where a is the reaction order and k is given by the Arrhenius expression

$$k = k_0 \exp\left(-\frac{E/R}{T}\right). \tag{7}$$

The operation of the reactor is influenced by inputs $\dot{V}_\mathrm{i}$, $c_\mathrm{A,i}$, T_i and T_c, and it is of interest to study how these influence the outputs c_A, c_B and T.

Although the case study has inputs $u = \left(\dot{V}_\mathrm{i}, c_\mathrm{A,i}, T_\mathrm{i}, T_\mathrm{c}\right)$ and outputs $y = (c_\mathrm{A}, c_\mathrm{B}, T)$, in a control problem one may choose to control the temperature $y = T$ by manipulating the input $u = T_\mathrm{c}$. In that case, the additional inputs $\left(\dot{V}_\mathrm{i}, c_\mathrm{A,i}, T_\mathrm{i}\right)$ may be considered disturbances.

For practical purposes, the cooling *liquid* temperature $T_\mathrm{c} \in [4,90]$ °C or so.[15] In reality, T_c is not directly controllable. Instead, an influent "cooling" temperature $T_\mathrm{c,i}$ could be varied while temperature T_c in the heat exchanger is varied via manipulating the cooling liquid flow rate.

[14] *Inert* implies non-reacting.

[15] If vaporized, higher temperatures may be achieved.

2.3 Balance Equations

For the given process, the following balance equations are relevant,

$$\frac{dm_{\mathrm{S}}}{dt} = \dot{m}_{\mathrm{S,i}} - \dot{m}_{\mathrm{S,e}} \tag{8}$$

$$\frac{dn_{\mathrm{A}}}{dt} = \dot{n}_{\mathrm{A,i}} - \dot{n}_{\mathrm{A,e}} + \dot{n}_{\mathrm{A,g}} \tag{9}$$

$$\frac{dn_{\mathrm{B}}}{dt} = \dot{n}_{\mathrm{B,i}} - \dot{n}_{\mathrm{B,e}} + \dot{n}_{\mathrm{B,g}} \tag{10}$$

$$\frac{dU}{dt} = \dot{H}_{\mathrm{i}} - \dot{H}_{\mathrm{e}} + \dot{Q} \tag{11}$$

where $\dot{x}$ indicates flow of variable x (in general, $\dot{x} \neq \frac{dx}{dt}$), m is mass, n is amount in mole, U is internal energy, and H is enthalpy; from thermodynamics, $U \triangleq H - pV$ and $\dot{U} \triangleq \dot{H} - p\dot{V}$. The differential variable is $x = (m_{\mathrm{S}}, n_{\mathrm{A}}, n_{\mathrm{B}}, U)$.

Indices i, e, and g indicate influent, effluent, and generation, respectively.

2.4 Model Complexity

Here, we will operate with two levels of model complexity.

1. At first, we will simply assume an *ideal solution*. The resulting model will occasionally be referred to with index is. For this model, the reactor composition influences both reactor density and energy properties.

2. In a simpler model, we will both assume constant overall density due to domination of solvent S, and that the composition does not influence the energy properties. If we also assume a first order reaction ($a = 1$), these are the assumptions behind the original model as presented in Example 2.5 of (Seborg et al., 2011). The *original* model is in state space form, with c_{A} and T as states. This model will occasionally be referred to with index org.

2.5 Ideal Solution Model

We assume a perfectly mixed tank volume V. We will also assume an *ideal solution*. Let superscript $\bullet$ indicate a quantity representing a hypothetically pure substance. Assuming an ideal solution implies:

$$V = V_{\mathrm{S}}^{\bullet} + V_{\mathrm{A}}^{\bullet} + V_{\mathrm{B}}^{\bullet} \tag{12}$$

$$H = H_{\mathrm{S}}^{\bullet} + H_{\mathrm{A}}^{\bullet} + H_{\mathrm{B}}^{\bullet} \tag{13}$$

where

$$V_j^{\bullet} = \frac{m_j}{\rho_j^{\bullet}} \tag{14}$$

$$m_j = n_j M_j \tag{15}$$

$$H_j^{\bullet} = m_j \hat{H}_j^{\bullet} = n_j \tilde{H}_j^{\bullet}. \tag{16}$$

Similarly, for the influent flow, we find

$$\dot{V}_{\mathrm{i}} = \dot{V}_{\mathrm{S,i}}^{\bullet} + \dot{V}_{\mathrm{A,i}}^{\bullet}; \tag{17}$$

$\dot{V}_{\mathrm{B,i}}^{\bullet} \equiv 0$ (no feed of species B). Equations 12, 13, and 17 are based on the assumption of ideal solution, and are not generally valid. Equations 14–16, on the other hand, are valid by definition.

In addition to the ideal solution assumption, we need a model of liquid specific enthalpy. For an ideal (incompressible) liquid with temperature independent specific heat capacity:

$$\hat{H}_{\mathrm{S}}^{\bullet} = \hat{H}_{\mathrm{S}}^{\circ} + \hat{c}_{p,\mathrm{S}}^{\bullet}\,(T - T^{\circ}) \tag{18}$$

and similar for species A and B.[16] Here, $\left(\hat{H}_{\mathrm{S}}^{\circ}, T^{\circ}\right)$ are specific enthalpy at standard state and a chosen reference temperature T°. Assuming ideal solution implies that

$$m_{\mathrm{S}} = \rho_{\mathrm{S}}^{\bullet}\left(V - \frac{n_{\mathrm{A}} M_{\mathrm{A}}}{\rho_{\mathrm{A}}^{\bullet}} - \frac{n_{\mathrm{B}} M_{\mathrm{B}}}{\rho_{\mathrm{B}}^{\bullet}}\right) \tag{19}$$

which gives an algebraic constraint between balanced variables $m_{\mathrm{S}}, n_{\mathrm{A}}, n_{\mathrm{B}}$, hence only two of the three amount balances in Section 2.3 are independent — and we really only have 3 states in the system. Similarly (no influent of species B),

$$\dot{m}_{\mathrm{S,i}} = \rho_{\mathrm{S}}^{\bullet}\left(1 - \frac{c_{\mathrm{A,i}} M_{\mathrm{A}}}{\rho_{\mathrm{A}}^{\bullet}}\right)\dot{V}_{\mathrm{i}}. \tag{20}$$

In total, the ideal solution DAE model has 18 constants/-parameters θ, 4 differential variables x, 41 algebraic variables z, and 4 input variables u. The number of state variables is 3; these are necessary to specify the system at initial time. Model parameters are given in Table 3, while operating conditions are given in Appendix A, Table 4.

2.6 Original ODE Model

The ideal solution DAE may be reduced to ODE form. Converting the species balances to differential equations for the concentrations, the result is rather complicated, and can be expressed in implicit form as

$$M \cdot \frac{d}{dt}\begin{pmatrix} c_{\mathrm{A}} \\ c_{\mathrm{B}} \end{pmatrix} = \begin{pmatrix} \frac{c_{\mathrm{A,i}}\dot{V}_{\mathrm{i}} - arV}{V} - \frac{c_{\mathrm{A}}\dot{m}_{\mathrm{S,i}}/\left(\rho_{\mathrm{S}}^{\bullet}V\right)}{1 - \frac{c_{\mathrm{A}}M_{\mathrm{A}}}{\rho_{\mathrm{A}}^{\bullet}} - \frac{c_{\mathrm{B}}M_{\mathrm{B}}}{\rho_{\mathrm{B}}^{\bullet}}} \\ r - \frac{c_{\mathrm{B}}\dot{m}_{\mathrm{S,i}}/\left(\rho_{\mathrm{S}}^{\bullet}V\right)}{1 - \frac{c_{\mathrm{A}}M_{\mathrm{A}}}{\rho_{\mathrm{A}}^{\bullet}} - \frac{c_{\mathrm{B}}M_{\mathrm{B}}}{\rho_{\mathrm{B}}^{\bullet}}} \end{pmatrix} \tag{21}$$

where M is the "mass matrix"

$$M = \begin{pmatrix} 1 + \frac{c_{\mathrm{A}}M_{\mathrm{A}}/\rho_{\mathrm{A}}^{\bullet}}{1 - \frac{c_{\mathrm{A}}M_{\mathrm{A}}}{\rho_{\mathrm{A}}^{\bullet}} - \frac{c_{\mathrm{B}}M_{\mathrm{B}}}{\rho_{\mathrm{B}}^{\bullet}}} & \frac{c_{\mathrm{A}}M_{\mathrm{B}}/\rho_{\mathrm{B}}^{\bullet}}{1 - \frac{c_{\mathrm{A}}M_{\mathrm{A}}}{\rho_{\mathrm{A}}^{\bullet}} - \frac{c_{\mathrm{B}}M_{\mathrm{B}}}{\rho_{\mathrm{B}}^{\bullet}}} \\ \frac{c_{\mathrm{B}}M_{\mathrm{A}}/\rho_{\mathrm{A}}^{\bullet}}{1 - \frac{c_{\mathrm{A}}M_{\mathrm{A}}}{\rho_{\mathrm{A}}^{\bullet}} - \frac{c_{\mathrm{B}}M_{\mathrm{B}}}{\rho_{\mathrm{B}}^{\bullet}}} & 1 + \frac{c_{\mathrm{B}}M_{\mathrm{B}}/\rho_{\mathrm{B}}^{\bullet}}{1 - \frac{c_{\mathrm{A}}M_{\mathrm{A}}}{\rho_{\mathrm{A}}^{\bullet}} - \frac{c_{\mathrm{B}}M_{\mathrm{B}}}{\rho_{\mathrm{B}}^{\bullet}}} \end{pmatrix}, \tag{22}$$

and the reaction rate is as in Eqs. 6–7. The internal energy balance can be simplified to

$$C_p\frac{dT}{dt} = \frac{\dot{V}_{\mathrm{i}}}{V}C_{p,\mathrm{i}}\,(T_{\mathrm{i}} - T) + \left(-\Delta_{\mathrm{r}}\tilde{H}\right)rV + \dot{Q} \tag{23}$$

[16] For species A and B, molar enthalpy is used.

where

$$C_p = c_{\mathrm{A}} V \tilde{c}^{\bullet}_{p,\mathrm{A}} + c_{\mathrm{B}} V \tilde{c}^{\bullet}_{p,\mathrm{B}} + m_{\mathrm{S}} \hat{c}^{\bullet}_{p,\mathrm{S}} \tag{24}$$

$$C_{p,\mathrm{i}} = V \rho^{\bullet}_{\mathrm{S}} \hat{c}_{p,\mathrm{S}} + V c_{\mathrm{A,i}} \tilde{c}_{p,\mathrm{A}} \tag{25}$$

with m_{S} as in Eq. 19, while the reaction enthalpy $\Delta_{\mathrm{r}}\tilde{H}$ is

$$\Delta_{\mathrm{r}}\tilde{H} = \tilde{H}^{\circ}_{\mathrm{B}} - a\tilde{H}^{\circ}_{\mathrm{A}} + \left(\tilde{c}^{\bullet}_{p,\mathrm{B}} - a\tilde{c}^{\bullet}_{p,\mathrm{A}}\right)(T - T^{\circ}) \tag{26}$$

and added heat rate is as in Eq. 5.

If we assume that the solvent totally dominates the mixture, the ideal solution model simplifies further to an ODE with identical structure as that of Example 2.5 in (Seborg et al., 2011). If, furthermore, the reaction order is set to $a = 1$ and specific heat capacities are chosen as $\tilde{c}^{\bullet}_{p,\mathrm{B}} \equiv \tilde{c}^{\bullet}_{p,\mathrm{A}}$, then the model becomes identical to that of (Seborg et al., 2011). The original, constant density model is presented in (Khalili and Lie, 2018).

3 Implementation Details

3.1 Basic Language Characteristics

This paper considers modeling languages Modelica, Julia, and MATLAB. For Modelica, tool OpenModelica is used.

Modelica is equation based and object oriented. Equality symbol = represents true mathematical equality, thus equations can appear in arbitrary order, and equations can be implicit. Constants and (default) parameters are named and given value within the model class, and input variables are supported. Modelica is object oriented, thus statement `Real T = 273.15` instantiates and gives value to a quantity named `T` in class `Real`. Similarly, if we create a model class `CSTR`, statement `CSTR mCSTR(R=8.31)` instantiates a model object `mCSTR` from class `CSTR` and sets class parameter `R = 8.31`.

When running Modelica models, they are first translated to assignments. This implies that a symbolic reordering and simplification of the model equations take place, before C code is generated and compiled to an executable file. If debugging is needed, this complicates matters because the actually executed order of the equations may differ from the order in the Modelica code.

Both Julia and MATLAB are assignment based. Thus, symbol = represents assignment. In assignments, the order of the statements is important. Typically, the current value of the model descriptor is passed as an input argument, possibly together with parameter values, and current time. There is no direct support for system inputs. The current descriptor and the parameter are passed with the abstract names (e.g., `x` and `par`), and may be given a physically recognizable name within the function body. Examples: if the first element in the descriptor is the mass of solvent m_{S}, we could name it as `m_S = x(1)`. Similarly, if the first element of the parameter is $\mathscr{U}A$, we could name it as `UA = par(1)`. This would be needed in order to use recognizable label names in the model formulation — for ease of understanding the code. Also, initial values of the full descriptor and parameter needs to be given outside of the model function.

Being assignment based, debugging is relatively easy because the statements are executed in the order they appear in the model.

3.2 Modelica Formulation

In Modelica (Fritzson, 2015), models are classes; the Ideal Solution model is named `ModSeborgCSTRis` and is enclosed between statements

```
model ModSeborgCSTRis
...
end ModSeborgCSTRis;
```

The model body is composed of sections, given by a section statement which ends with line feed. Within sections, statements can be multi-line, and are ended by a semicolon `;`. The model quantities are usually defined in the first section. As an example, parameter $\rho^{\circ}_{\mathrm{S}}$, variable m_{S} with consistent initial value `mS0`, and input variable T_{c} are defined by

```
parameter Real rhoS_o = 1e3;
Real mS(start = mS0);
input Real Tc;
```

After the declaration of quantities, an equation section is declared by section statement **`equation`**. As an example, equations $\frac{dn_{\mathrm{A}}}{dt} = \dot{n}_{\mathrm{A,i}} - \dot{n}_{\mathrm{A,e}} + \dot{n}_{\mathrm{A,g}}$ and $V = V_{\mathrm{S}} + V_{\mathrm{A}} + V_{\mathrm{B}}$ can be stated as

```
der(nA) = ndAi - ndAe + ndAg;
V = VS + VA + VB;
```

For balanced models, all model constants/parameters should be specified within the model class, and the model should hold the same number of equations as variables — with the exception of input variables which are defined outside of the model class.

Models without input variables can be instantiated directly from the model class, and simulated. On the other hand, models with input variables must be instantiated in another model class, and be given an input value in this other model class. Several models can be wrapped within a *package* and put in a file with the same name as the package. Here, we use package name `SeborgCSTR`, and the package is enclosed between statements

```
package SeborgCSTR
...
end SeborgCSTR;
```

To define the inputs to model `ModSeborgCSTRis`, we thus create a second model `SimSeborgCSTR` which is enclosed statements as required for model classes. Within this second model, we instantiate object `srIS` in the declaration section by statement `ModSeborgCSTRis srIS;` — the class name followed by the object name, similar to the statement `Real mS(start = mS0);`. It is possible to let the instantiated object `srIS` over-rule a parameter specified in class `ModSeborgCSTRis` just as object

mS has over-ruled a (default) parameter in class Real. Finally, in the equation section of class SimSeborgCSTR, we can give an input value to the instantiated object srIS with a statement such as srIS.Tc = 300;.

To run the Modelica model, we can import the package into OpenModelica[17], select the main class that we want to solve (SimSeborgCSTR), check the model for errors, and set up simulation details (solver, simulation length, tolerance, etc.), and simulate the system. OpenModelica has simple facilities for plotting results and saving the plots to file.

3.3 Julia Formulation

To solve differential equations in Julia[18], it is necessary to add package DifferentialEquations (Rackauckas and Nie, 2017) from GitHub, which is straightforward. To activate this package within a Julia session, issue statement

```
julia> using DifferentialEquations
```

With this package, DAEs are posed as a DAEProblem. As an example, specifying the Ideal Solution model in function seborg_is with descriptor x[19] and initial value guesses for $\frac{dx}{dt}\big|_0$ and $x|_0$ given by dxdt0 and x0, simulation time span tspan, and parameter vector θ given in par_is, the DAE problem named probis is set up by command:

```
julia> probis = DAEProblem(seborg_is,dxdt0,x0
    ,tspan,par_is,differential_vars=diff_vars
    )
```

where keyword argument differential_vars is given value diff_vars; diff_vars is a Boolean vector (1D array) with true value for elements corresponding to differential variables in x, and false value for elements corresponding to algebraic variables.

The DAE problem is then solved by issuing command

```
julia> solis = solve(probis, IDA())
```

where IDA is the name of the chosen solver code (Hindmarsh et al., 2005). The solution is stored in type (object) solis.

The model has been implemented in function seborg_is, which is defined with arguments seborg_is(err,dxdt,x,par,t). Here, variable err contains the errors in the equations: if e is this error, then the DAE formulation is rephrased into $e_\mathrm{d} = -\frac{dx}{dt} + f(x,z,u;\theta)$ and $e_\mathrm{a} = g(x,z,u;\theta)$, respectively for the differential and algebraic equations; the solver then attempts to make the error err (= e_d, e_a) as close to zero as possible. In order to operate with variables within the Julia function which are physically descriptive, it is necessary to rename the abstract names like dxdt, x, and par into dmSdt = dxdt[1], etc. The model is then specified via statements for err, e.g.,

```
err[1] = -V + VS + VA + VB
...
err[42] = -dmSdt + mdSi - mdSe
```

etc. Input variables must be specified within the model function.

The Julia solver is more restrictive wrt. initial conditions than OpenModelica: for Julia, decent guesses for both dxdt0 and x0 must be supplied in order for the solution to be found, while for OpenModelica, initial conditions for the *state* suffice.

3.4 MATLAB Formulation

The MATLAB[20] formulation is written in a script named seborg_is. There are several ways to formulate DAE models in MATLAB, e.g., via the Mass Matrix. Here, we have instead used the Symbolic Math Toolbox to convert the system to a form suitable for numeric solvers.[21]

Quantities are declared using the syms function, with a distinction between variables and constants. As an example, the dependent variable n_A is written as nA(t) and the parameter V as V. Further, the Symbolic Math Toolbox functionality allows for the algebraic relations to be specified as mathematical equalities. As an example, equations $\frac{dn_\mathrm{A}}{dt} = \dot{n}_\mathrm{A,i} - \dot{n}_\mathrm{A,e} + \dot{n}_\mathrm{A,g}$ and $n_\mathrm{A} = c_\mathrm{A} V$ can be written as

```
eqn1 = diff(nA(t),1) == ndAi - ndAe(t) + ndAg
    (t);
...
eqn5 = nA(t) == cA(t)*V;
```

These equations are stored in a row vector and the variables in a column vector.

In the case of ideal solution, it is necessary to reduce the differential index by invoking the reduceDifferentialOrder function. This function takes the equations and variables as input and creates new equations and variables to replace derivatives.

Next, input variables and parameters must be assigned a value and formulated as function handles suitable for MATLAB solvers. MATLAB is very sensitive regarding the order of assignment of variables and initial values.

It is also necessary to set initial values for every variable, including the differential variables created by reduceDifferentialOrder, which can be complicated to initialize. The decic function can be called to find consistent initial values that satisfy the equations from initial guesses. This function is sensitive to the input tolerances, and may produce false solutions if they are set too low.

Finally, the system is solved by integrating over a specified time span with the implicit solver ode15i.

[17] https://openmodelica.org/

[18] https://julialang.org/

[19] Julia doesn't distinguish between differential and algebraic variables.

[20] https://mathworks.com/

[21] https://se.mathworks.com/help/symbolic/solve-differential-algebraic-equations.html?s_tid=gn_loc_drop

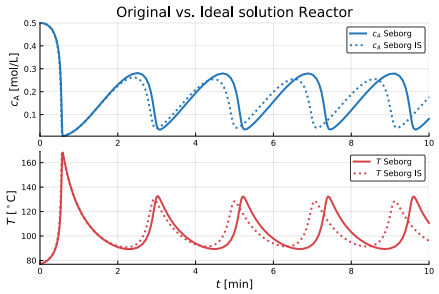

Figure 2. Temporal evolution of concentration c_A and temperature T for original model (solid) and ideal solution model (dotted).

Table 1. Average Scaled Execution Times over 1000 runs; basis is 5.3 ms for solving Ideal Solution (IS) model in Julia.

	OM*	Julia	MATLAB
ORG	11.5	0.36	0.3 (6.0)
IS	12.4	1	11.1

4 Simulation Results

4.1 Simulation Results

The simulation results are the same, whether OpenModelica, Julia, or MATLAB are used. Figure 2 illustrates the temporal evolution for the original model and the ideal solution model, as computed and plotted by Julia running in IJulia notebook[22] when the cooling temperature jumps from $T_c = 300\,\mathrm{K}$ to $T_c = 305\,\mathrm{K}$ at initial time.

4.2 Execution Time

It is very difficult to compare various languages wrt. the execution time for solving models. Important considerations are:

- Are the same algorithms used?
- Is the same solution tolerance used?
- Is the execution time problem dependent?
- Do all languages have the same initial information, e.g., the same initial guess if iterations are needed?
- Do all languages have the same overhead in finding the solution?

Table 1 reports some execution times for OpenModelica as run from Python, for Julia, and for MATLAB — for the original model of (Seborg et al., 2011) and the DAE formulation of the ideal solution model. The results are indicative for assessing the languages, at best.

A few comments on the results in Table 1.

1. In all cases, different (but comparable?) solvers are used (OpenModelica: DASSL for ORG/IS; Julia: default for ORG, IDA for DAE; MATLAB: ode45(ode15s) for ORG, ode15i for IS). For DAE problems, the tolerances have been tuned to find a solution for Julia and MATLAB.

2. For the DAE model, OpenModelica and Julia need to iterate to find initial value for the entire descriptor, while this has been precomputed for MATLAB.

3. For the ODE model, MATLAB solves the problem quickly when using the `ode45` solver, but 20 times slower when using the `ode15s` solver. This must be due to considerable overhead when using a stiff solver on a non-stiff problem.

4. OpenModelica code has been optimized, translated to C and compiled into an executable file. Thus, OpenModelica code should execute fast. The reason why OpenModelica is relatively slow must therefore lie in some inefficiency in the Python API.

[22] = Jupyter notebook

Table 2. Comparison of Simulation Tools for solving DAE models. OM = (Open)Modelica, grading A–F where A is best.

	OM	Julia	MATLAB
Cost	A	A	E
Ease of use	B	C	D
Documentation	C	D	A
Numeric quality	A	A	A
Debugging	B	C	A
Eco-system	D	B	A
Library facility	A	C	D

5 Discussion

Table 2 gives a *subjective* comparison of computer languages/tools MATLAB, Julia, and OpenModelica.

With excellent support for libraries, similarity between theoretic model and implementation, support for input variables, and inclusion of named quantities within classes, Modelica is made for DAEs and is easy to use. It suffices to specify initial states; this adds to the ease of use. Julia package `DifferentialEquations.jl` gives good support for DAEs. MATLAB has good built-in support for ODEs; for solving DAEs, some complexity is involved in combining additional toolboxes such as the Symbolic Toolbox. Julia and MATLAB require good initial guesses for the entire descriptor and its derivative, which adds to their complexity.

MATLAB documentation is excellent, Modelica has a handful of introductory books, while the Julia language is still in a flux prior to v. 1.0[23] with partially outdated and simple books. All three languages have excellent solvers, still Julia has by far the largest set of available solvers.

MATLAB has a useful visual debugger with breakpoints. Being a young language, Julia currently has somewhat poor support for debugging. Because both languages

[23] Julia v.1.0 is scheduled for August 6, 2018.

are assignment based, debugging is relatively easy. Modelica is equation based, thus equations are reordered and modified during translation to executable code without user intervention. It is therefore difficult for the user to keep track of which assignment comes first — and therefore which equation causes the problem. Modelica does have decent syntax checking and the user can keep track of undeclared/unused quantities, and whether the number of equations matches the number of unknowns. Also, there is a debugger which keeps track of the re-ordering of equations, but some expertise is needed to use this debugger efficiently.

MATLAB comes with good plotting capabilities, random number generators, etc., and has a rich Eco-system. Julia has a large set of available packages of very good quality, including support for Automatic Differentiation, which conceptually can be used for automatic linearization of models. Modelica has a more limited Eco-system, and lacks good support for random number generation, good plotting capabilities, etc. However, OpenModelica has support for automatic linearization, optimal control, etc., which can be integrated with Python via a Python API.

Although MATLAB and Julia have tools for solving DAEs, they lack good support for libraries and flowsheeting. Modelica, on the other hand, excel in these areas.

6 Conclusions

When it comes to ease of use, direct translation from mathematical model to computer code, and support for libraries, Modelica is the clear winner. Coding DAE models is simple in Julia, too, but the notation is more abstract, and more importantly: a decent initial value for the entire descriptor is required. MATLAB is somewhat more complex in use than Julia again, and is also demanding wrt. initial value for the descriptor.

Regarding documentation and debugging, MATLAB is the clear winner. Julia is a very new language; v. 1.0 is still in the pipeline, and it is natural that the documentation is lacking. However, it is expected that documentation for Julia, as well as debugging facilities, will improve rapidly. Modelica has good documentation and decent debugging facilities.

The MATLAB Eco-system is extensive; Julia's is smaller, but surpasses that of MATLAB in some areas. The Modelica/OpenModelica Eco-system is more limited, with only rudimentary tools for plotting and analysis.

Regarding numeric quality, the languages are comparable although Julia is richest wrt. solvers. For execution speed, OpenModelica (compiled C code) and Julia (low level JIT compiler) should have an edge over MATLAB. Limited results indicate that, indeed, Julia is very fast. However, OpenModelica is comparable to MATLAB. The reason for this may lie in overhead in the Python API.

The ideal tool would be a one-language solution; this would enable application of the entire Eco-system on the model, e.g., automatic differentiation, mixing model and optimization, structural analysis of the model, etc. At the moment, Julia is perhaps the best/most promising one-language tool.

However, recent work enables operation of OpenModelica within Python; support for integration of OpenModelica in both Julia and MATLAB are in the works. Although these are two-language solutions, the combinations reduce OpenModelica's disadvantage wrt. Eco-system. It would be better with an even tighter integration to eliminate interface overhead; this could, e.g., be achieved by compilation of Modelica into a script language instead of into C.

A Parameters and Operating Conditions

For the DAE model, choose the following parameters for the model, Table 3. The operating conditions (initial states, inputs) are defined in Table 4.

References

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.

Michael A. Henson and Dale E. Seborg. *Nonlinear Process Control*. Prentice Hall, Upper Saddle River, New Jersey, 1997.

A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*, 31(3):363–396, 2005. Also available as LLNL technical report UCRL-JP-200037.

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, Arun Kumar, 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.

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.

Table 3. Parameters for ideal solution liquid reactor.

Quantity	Symbol	Value
Density of pure species S	$\rho_{\mathrm{S}}^{\bullet}$	$1000\,\mathrm{g/L}$
Density of pure species A	$\rho_{\mathrm{A}}^{\bullet}$	$1500\,\mathrm{g/L}$
Density of pure species B	$\rho_{\mathrm{B}}^{\bullet}$	$2500\,\mathrm{g/L}$
Molar mass of species A	M_{A}	$50\,\mathrm{g/mol}$
Molar mass of species B	M_{B}	$a \cdot M_{\mathrm{A}}$
Reactor volume	V	$100\,\mathrm{L}$
Stoichiometric constant	a	1
Activation energy per gas constant	E/R	$8750\,\mathrm{K}$
Reaction constant	k_0	$\exp\left(\frac{E/R}{350}\right) \approx 7.2 \cdot 10^{10}\,\mathrm{min}^{-1}$
Standard state temperature	T°	$293.15\,\mathrm{K}$
Standard state pressure	p°	$1.01 \times 10^5\,\mathrm{Pa}$
Specific enthalpy of species S at (T, p°)	$\hat{H}_{\mathrm{S}}^{\circ}$	$0\,\mathrm{J/g}$
Molar enthalpy of species A at (T, p°)	$\tilde{H}_{\mathrm{A}}^{\circ}$	$5 \times 10^4\,\mathrm{J/mol}$
Molar enthalpy of species B at (T, p°)	$\tilde{H}_{\mathrm{B}}^{\circ}$	$0\,\mathrm{J/mol}$
Specific heat capacity of species S	$\hat{c}_{p,\mathrm{S}}^{\bullet}$	$0.239\,\mathrm{J/(g\,K)}$
Molar heat capacity of species A	$\tilde{c}_{p,\mathrm{A}}^{\bullet}$	$5\,\frac{\mathrm{J}}{\mathrm{mol\,K}}$
Molar heat capacity of species B	$\tilde{c}_{p,\mathrm{B}}^{\bullet}$	$5\,\frac{\mathrm{J}}{\mathrm{mol\,K}}$
Heat transfer parameter	$\mathscr{U}A$	$5 \times 10^4\,\mathrm{J/(min\,K)}$

Table 4. Operating conditions for ideal solution liquid reactor. Superscript $*$ for inputs indicates nominal inputs.

Quantity	Symbol	Value
Initial value, concentration of A	$c_{\mathrm{A}}\vert_{t=0}$	$0.5\,\mathrm{mol/L}$
Initial mole number of species B	$n_{\mathrm{B}}\vert_{t=0}$	$0\,\mathrm{mol}$
Initial temperature	$T\vert_{t=0}$	$350\,\mathrm{K}$
Initial mole number, A	$n_{\mathrm{A}}\vert_{t=0}$	$= c_{\mathrm{A}}\vert_{t=0} \cdot V$ in mol
Initial mass, species A	$m_{\mathrm{A}}\vert_{t=0}$	$n_{\mathrm{A}}\vert_{t=0} \cdot M_{\mathrm{A}}$ in g
Initial mass, species B	$m_{\mathrm{B}}\vert_{t=0}$	$n_{\mathrm{B}}\vert_{t=0} \cdot M_{\mathrm{B}}$ in g
Initial volume, pure species A	$V_{\mathrm{A}}\vert_{t=0}$	$\frac{m_{\mathrm{A}}\vert_{t=0}}{\rho_{\mathrm{A}}^{\bullet}}$ in L
Initial volume, pure species B	$V_{\mathrm{B}}\vert_{t=0}$	$\frac{m_{\mathrm{B}}\vert_{t=0}}{\rho_{\mathrm{B}}^{\bullet}}$ in L
Initial volume, pure species S	$V_{\mathrm{S}}\vert_{t=0}$	$V - V_{\mathrm{A}}\vert_{t=0} - V_{\mathrm{B}}\vert_{t=0}$ in L
Initial mass, species S	$m_{\mathrm{S}}\vert_{t=0}$	$V_{\mathrm{S}}\vert_{t=0} \cdot \rho_{\mathrm{S}}^{\bullet}$ in g
Initial specific enthalpy of species S	$\hat{H}_{\mathrm{S}}^{\bullet}\vert_{t=0}$	$\hat{H}_{\mathrm{S}}^{\circ} + \hat{c}_{p,\mathrm{S}}^{\bullet}\,(T\vert_{t=0} - T^{\circ})$ in J/g
Initial molar enthalpy of species A	$\tilde{H}_{\mathrm{A}}^{\bullet}\vert_{t=0}$	$\tilde{H}_{\mathrm{A}}^{\circ} + \tilde{c}_{p,\mathrm{A}}^{\bullet}\,(T\vert_{t=0} - T^{\circ})$ in J/mol
Initial molar enthalpy of species B	$\tilde{H}_{\mathrm{B}}^{\bullet}\vert_{t=0}$	$\tilde{H}_{\mathrm{B}}^{\circ} + \tilde{c}_{p,\mathrm{B}}^{\bullet}\,(T\vert_{t=0} - T^{\circ})$ in J/mol
Initial enthalpy of species S	$H_{\mathrm{S}}^{\bullet}\vert_{t=0}$	$m_{\mathrm{S}}\vert_{t=0}\,\hat{H}_{\mathrm{S}}^{\bullet}\vert_{t=0}$ in J
Initial enthalpy of species A	$H_{\mathrm{A}}^{\bullet}\vert_{t=0}$	$n_{\mathrm{A}}\vert_{t=0}\,\tilde{H}_{\mathrm{A}}^{\bullet}\vert_{t=0}$ in J
Initial enthalpy of species B	$H_{\mathrm{B}}^{\bullet}\vert_{t=0}$	$n_{\mathrm{B}}\vert_{t=0}\,\tilde{H}_{\mathrm{B}}^{\bullet}\vert_{t=0}$ in J
Initial total enthalpy of ideal mixture	$H\vert_{t=0}$	$H_{\mathrm{A}}^{\bullet}\vert_{t=0} + H_{\mathrm{B}}^{\bullet}\vert_{t=0} + H_{\mathrm{S}}^{\bullet}\vert_{t=0}$ in J
Initial internal energy	$U\vert_{t=0}$	$H\vert_{t=0} - p^{\circ}V \times 10^{-3}$ in J
Influent volumetric flow rate	$\dot{V}_{\mathrm{i}}^{*}$	$100\,\mathrm{L/min}$
Influent concentration of species A	$c_{\mathrm{A,i}}^{*}$	$1\,\mathrm{mol/L}$
Influent temperature	T_{i}^{*}	$350\,\mathrm{K}$
Cooling liquid temperature	T_{c}^{*}	$300\,\mathrm{K}$