Fertilizer Granulation: Comparison of Modeling Languages

Amir Farzin, Ludmila Vesjolaja, and Bernt Lie

University of South-Eastern Norway, Porsgrunn, Norway, Bernt.Lie@usn.no

Abstract

Industrially produced fertilizers are of key importance to produce enough food for a growing global population. On-going work deals with models of the granulation loop in fertilizer production, based on a population balance that finds the particle size distribution of the product. The model is intended for control design in order to dampen or remove production oscillation for reduced energy consumption and improved product quality. In this paper, efficiency of model implementation is studied in addition to the possibility to automate the computation of a linear approximation of the model for control synthesis.

In the implementation study, the current tailor-made MATLAB solver for the model was cloned in computer language Julia. In addition, the implementations in both languages (MATLAB, Julia) were rewritten in a form that allows for use of the standard differential equation solvers of the respective languages. Results indicate that by changing from the tailor-made solvers to using the built-in solvers leads to a speed increase in the order of 6 times. Furthermore, results indicate that the Julia implementations are ca. 5 times faster than the MATLAB implementations. Overall, the fastest Julia implementation was 36 times faster than the current MATLAB implementation. The MATLAB execution can be sped up by using MATLAB Coder to convert the code to efficient C-code which is then used to generate a DLL. DLLs can be executed virtually without overhead from Julia. By measuring the execution time for the C-code/DLL vs. a similar implementation in pure Julia, the pure Julia code is ca. 12% *faster* than the compiled C code.

Next, the possibility of automatic linearization of the population balance model in Julia is studied. This is shown to be relatively straightforward. The linear approximation is very good for an input perturbation of 10%, and relatively good for an input perturbation of 50%. This indicates that it may be possible to use a linear model approximation for control design.

Keywords: linear regression, nonlinear regression, thermal model, machine learning, surrogate model, hybrid model.

1 Introduction

1.1 Background

Industrially produced fertilizers are of key importance in order to produce enough food for a growing global population. Fertilizers in the form of granules is allows for simple application and spreading of fertilizers. The quality of such fertilizers are determined by the average size and the size distribution of the fertilizer, as well as liquid content and porosity. Granulation of fertilizers at times lead to oscillatory operation, which widens the size distribution and increases the energy consumption. It is of interest to develop dynamic population balance models for fertilizer production which describes the size distribution, to some degree explains the product quality, and allows for understanding of what operating conditions lead to oscillatory behavior. Dynamic population balance models are partial differential equations in time as well as external and internal variables. The external variables are spatial position, while the internal variables are particle size, humidity, porosity, etc. The result is that dynamic population balance models are demanding to solve, both numerically and because of the model size.

To be used for on-line production planning/control, it may be necessary to solve the model much faster than real time, e.g., in some types of state estimators and in some types of optimization based control algorithms. It is therefore of interest to explore the possibility of optimizing the model formulation for fast execution within a given computer language, but also to explore whether different languages give different execution time. Some control algorithms may use a linear model approximation. Because of the complexity of the model, it is also of interest to study whether the linearization of the model can be automated in a given language. The ultimate goal of the population balance model is to see whether production with better quality and reduced energy consumption can be achieved.

1.2 Previous work

Population balances describe dynamic systems with both external and internal coordinates, leading to highly distributed models which are time consuming to solve (Wang and Cameron, 2007; Litster and Ennis, 2004; Iveson et al., 2001; Ramkrishna, 2000). (Vesjolaja et al., 2018) describe a population balance model for granulation of fertilizers, including both growth by layering and growth by agglomeration. As a population balance model, the model is relatively simple and homogeneous in the drum axial position as external coordinate, and particle size as internal coordinate. The two growth mechanisms require different types of discretization algorithms. In this simple implementation, the particle size is discretized into 80 different sizes. With 80 states in the model, the key output is the particle

size median. The resulting model is relatively complex as a dynamic model for control, but still simple as pertaining a population balance model. In a future stage, models will be extended with distribution in the external coordinate. It is of interest to compare different modeling languages wrt. solution efficiency.

The model is designed for control synthesis. Standard controllers include proportional (P) and proportional + integral (PI) controllers, which often are tuned based on some tuning rule, (Åstrøm and Murray, 2008). Controllers of mid-level complexity are based on linear approximations of the model, e.g., root locus methods, synthesis based on Nyquist, Nichols, or Bode diagrams, as well as linear quadratic controllers (LQR) and linear Model Predictive Control (MPC), (Maciejowski, 2002). Thus, it is also of interest to consider modeling languages wrt. how they can aid in controller synthesis, e.g., by providing linearized model approximation. Examples of popular languages for solving such models include MATLAB and C++, but a recent, free language such as Julia (Bezanson et al., 2017) with an extensive package for solving differential equations (Rackauckas and Nie, 2017) is an interesting candidate. Julia uses Just-in-Time (JIT) compilation with strong typing, and thus provides a bridge between easy-to-use script languages and compiled languages. Julia also has other advantages with simple-to-use, free packages for Automatic Differentiation (AD) and linearization (Revels et al., 2016), simple-to-use, free packages for computing with distributions (Besanon et al., 2019) such as particle size distribution, etc.

1.3 Overview of the paper

The current model has been implemented in MATLAB with a fixed, user-developed step-length RK4 solver, and a user-developed routine for computing the median of the distribution. We consider to replace the user-developed routines in MATLAB with built-in routines from the ODE solver tools to see if this can make the MATLAB implementation more efficient. Next, we consider implementing the model in Julia and compare the execution speed in Julia vs. that of MATLAB. Both a direct translation of the user-developed RK4 solver and median computation is used, as well as the use of Julia packages such as DifferentialEquations and Statistics. We also compare the execution time of the model written in pure Julia, vs. conversion of the MATLAB model to C-code/DLL using MATLAB Coder from MathWorks. Finally, we study how Julia can be used for linearization of the model.

The paper is organized as follows. In Section 2, an overview of the granulation process is given. In Section 3, key elements of the MATLAB implementation are discussed, with a comparison of the current implementation vs. the use of the built-in ODE solvers. Next, implementation issues for Julia are discussed. Then, simulation results are provided, with a comparison of execution speed. In Section 4, the possibility of automatic linearization of the model using Julia is discussed, with some simple results. Finally, some conclusions are drawn in Section 5.

2 Overview of Industrial Granulation

2.1 Fertilizer granulation

Granulation processes are used in a wide range of industrial applications, such as pharmaceutical and fertilizer industries. The research reported here is focused on the last part of NPK (Nitrogen, Phosphorus, Potassium) fertilizer production. A granulation loop is used to produce different grades, i.e., various N:P:K ratios, of fertilizers. The NPK fertilizer is a high value type of fertilizer containing the three main elements essential for crop nutrition. Various NPK grades are specially developed for different crops growing in different climates and soils.

2.2 Granulation loop

A typical schematic of a granulation process with a recycle loop is shown in Figure 1.

The granulation loop consists of a granulator, a granule classifier (screens), and a double-roll crusher. During the granulation process, a slurry of liquid ammonium nitrate and partly dissolved minerals is solidified to form granules. Granules that are too small (under-sized particles) are recycled to the granulation unit, while granules that are too large (over-sized particles) are crushed and then recycled back to the granulator. Different granulation mechanisms are responsible for the granule formation in the granulator, depending on the granulator type and operational conditions. The granulator can be of different types, e.g., a spherodizer, a rotary drum granulator, a fluidized bed granulator, a pan granulator, etc. Some of the granulation mechanisms that are responsible for the granule formation are particle growth due to layering and particle agglomeration. Particle growth due to layering is a continuous process during which particle growth occurs due to a successive coating of a liquid phase onto a granule. Binary particle agglomeration is a particle growth mechanism that occurs due to successful collision of two particles, resulting in the formation of a larger, composite particle (Litster and Ennis, 2004; Vesjolaja et al., 2018).

2.3 Production challenges

Fertilizer manufacturing using the granulation process has received considerable research interest during the last few decades, due to (i) the increasing requirements for efficient production of high quality fertilizers for increased food production in a growing global population, (ii) difficult process control and operation. Process control of granulation loops is challenging since the particle size distribution (PSD) of the granules leaving the granulator is wider than the required PSD of the final product. A typical recycle ratio between the off-spec particles and the required product-sized particles is 4:1 respectively. In addition, granulation loops may show oscillatory behavior for certain operating points. Typical oscillations seen in an-industrial scale fertilizer granulation plants is depicted

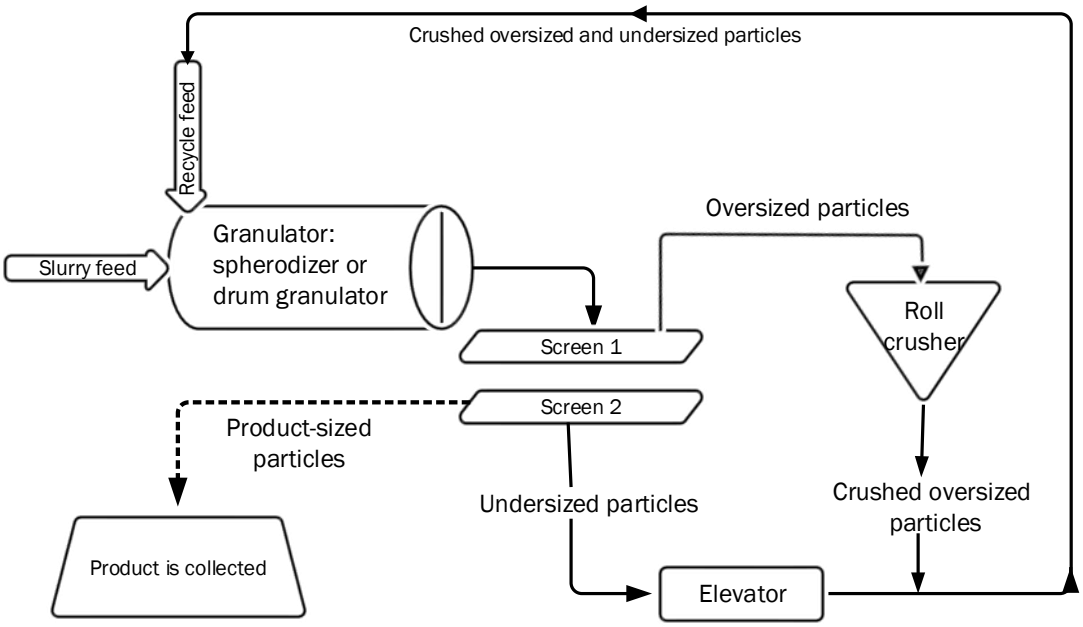

Figure 1. Granulation loop used in fertilizer industry.

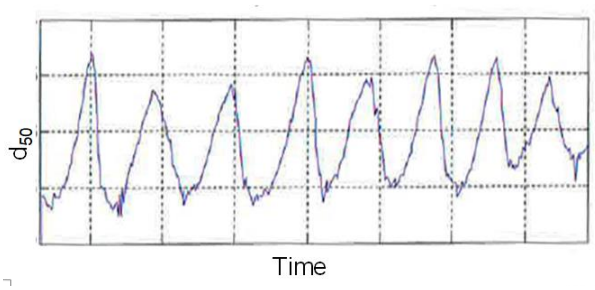

Figure 2. Oscillatory behavior observed in PSD (measured as d50) of the produced granules in fertilizer industry.

in Figure 2.

Control and operation of a granulation loop is still challenging. Some research papers with focus on process control of the granulation process are (Buck et al., 2016; Ramachandran and Chaudhury, 2012; Herce et al., 2017; Ramachandran et al., 2009; Valiulis and Simutis, 2009; Wang et al., 2006; Cameron et al., 2005).

2.4 Problem limitation

Here, we consider a model of the drum granulator with both layering and agglomeration, but we do not include the screening or the recycling. The main purpose here is to study the efficiency of model formulation and solution in MATLAB vs. Julia, and the possibility to use modern modeling tools for control analysis.

3 Model implementation details

3.1 Overview of model

The population balance for combined layering and agglomeration is discussed in (Vesjolaja et al., 2018), and is for form

$$\frac{\partial n(L,t)}{\partial t} = -\frac{\partial}{\partial L}\left(G \cdot n(L,t)\right) + B(L,t) - D(L,t) + \dot{n}_{\mathrm{i}}\gamma_{\mathrm{i}} - \dot{n}_{\mathrm{e}}\gamma_{\mathrm{e}} \tag{1}$$

where $n(L,t)$ is the number density as a function of particle diameter L and time t, G is the growth rate relevant for layering, $B(L,t)$ is the birth rate relevant for agglomeration, while the death rate $D(L,t)$ describes particle disintegration. n_{i} is the influent number flow rate, γ_i is the influent size distribution, while n_{e} and γ_{e} are similar quantities for the effluent. Here, perfect (external) mixing has been assumed in the granulator drum. Alternatively to a number population balance as in Eq. 1, it is often more convenient to describe the mass population balance, where m is related to n via

$$n = \frac{6m}{\pi\rho L^3}. \tag{2}$$

When discretizing the particle size space in N_{p} particle sizes, the mass population balance has the following form:

$$\frac{\mathrm{d}M_{1:N_{\mathrm{p}}}}{\mathrm{d}t} = f_{\mathrm{agg}}\left(M_{1:N_{\mathrm{p}}};\theta\right) + f_{\mathrm{growth}}\left(M_{1:N_{\mathrm{p}}},\dot{M}_{\mathrm{sl}};\theta\right) + \dot{M}_{\mathrm{i}}\gamma_{\mathrm{i}} - \dot{M}_{\mathrm{e}}\gamma_{\mathrm{e}},$$

where $M_{1:N_{\mathrm{p}}}$ is the vector of masses within the N_{p}size ranges $L_1,\ldots,L_{N_{\mathrm{p}}}$, θ is some model parameter, $\dot{M}_{\mathrm{sl}}$ is the

feed rate of slurry fertilizer spray, and $\dot{M}_e$ is given by some expression. The quality of interest for control is the *median* of the particle size distribution (PSD) in the effluent, measured in particle diameter and denoted by $d_{50,e}$,

$$d_{50,e} = \text{median}\left(L, M_{1:N_p}\right).$$

Introducing $x = M_{1:N_p}, u = \left(\dot{M}_i, \dot{M}_{sl}\right), y = d_{50,e}$, we can express the model in the more abstract form

$$\frac{dx}{dt} = f(x,u,t;\theta)$$
$$y = g(x,u,t;\theta).$$

For the work reported in (Vesjolaja et al., 2018), the mass-based model was implemented in MATLAB using a tailor-made, fourth order Runge-Kutta (RK4) fixed step-length solver with integral computation of $d_{50,e}$ including conversion from mass to diameter, and storage of the output $d_{50,e}$.

3.2 MATLAB

It is of interest to rewrite the MATLAB code of (Vesjolaja et al., 2018) and use the built-in MATLAB ODE solvers, to see if the code can be made faster. Some advantages of MATLAB are a rich toolbox suite, easy debugging, and excellent documentation.

ODE solvers in MATLAB have the following call structure:

```
[t , x] = solver (odefun, tspan, x0, options)
```

where we considered the following `solver` alternatives: standard solvers **`ode45`**, **`ode23`**, and `ode113`, as well as stiff solvers `ode15s`, `ode23t`, and `ode23tb`. When solved as ordinary differential equations, these solvers only store the state, $M_{1:N_p}$, thus the output $d_{50,e}$ must be computed by post processing. It is possible to solve differential algebraic equations (DAEs) with MATLAB, e.g., introducing a singular mass matrix, but such a reformulation is somewhat clumsy for certain types of problems (e.g., with re-circulation), and DAE solver are slower than ODE solvers.

MATLAB has decent support for computing quantiles of distributions, e.g., $d_{50,e}$ of the particle size distribution (PSD). However, because of the conversion from mass distribution to diameter distribution indicated in Eq. 2, support for quantile computation with weighting is required; this is not supported in standard MATLAB. Because of this, the original function for computing $d_{50,e}$ from the distribution $M_{1:N_p}$ is preserved.

3.3 Julia

In order to compare how fast the model can be solved, it is of interest to compare the MATLAB implementation with an implementation in another language. Standard reference languages for speed are C/C++ and FORTRAN. However, the relatively new language Julia which is a dynamic language in style, while using Just-in-time compilation for execution, is also known to be fast — at least when properly implemented. Julia has excellent packages for differential equation solvers, and very good support for statistics.

In general, Julia has relatively rudimentary documentation for packages compared to MATLAB toolboxes. Some integrated development environments (IDE) with debugger are starting to appear, but they are still inferior to the MATLAB IDE. However, packages are, in general, of good–excellent quality.

For comparison with MATLAB, first the tailor-made RK4 solver used in (Vesjolaja et al., 2018) was translated more or less directly to Julia. Next, just like for MATLAB, a rewrite of the code was made to take advantage of the differential equation solvers in Julia's package `DifferentialEquations.jl`. Every ODE solver in MATLAB has an equivalent solver algorithm in Julia (https://docs.sciml.ai/stable/solvers/ode_solve). Specifically, Julia versions of the non-stiff MATLAB solvers were used: `DP5` (**`ode45`**), `BS3` (**`ode23`**), `VCABM` (`ode113`). Because experiments with MATLAB indicated that the system is non-stiff, stiff Julia solvers were not considered. Finally, the standard Julia solver `Tsit5` was used.

Julia has support for quantile computations with weights, and in the rewritten code, this function was used to compute $d_{50,e}$.

3.4 C-code/DLL

MATLAB has tools for automatic conversion of code to C-code and compilation into DLLs. Such DLLs can then be included in Simulink blocks. It is also possible to call DLLs from Julia virtually without overhead. This makes it possible to compare the execution time of C-code vs. Julia and MATLAB.

3.5 Comparison

In this section, two implementations of the granulation drum model in each programming language will be compared, i.e., 4 implementations (with some variation in solvers). The tailor-made RK4 solvers (MATLAB and Julia) will be referred to as `TM-RK4`. Next, we rewrite the tailor made code to be in the standard forms for use with built-in ODE solvers. For these, we refer to the code with language name and solver, e.g., `M-`**`ode45`** for solver **`ode45`** in MATLAB, and `J-DP5` for the similar `DP5` solver for Julia.

Table 1 shows a comparison of run-time (execution time) for MATLAB and Julia for simulation over 1.5 h. The results displayed are the best run out of 20 runs, with $N_p = 80$. For the tailor-made solvers, a step size of $h = 10$ s is used. Both for MATLAB and Julia, adaptive time stepping is used in the built-in ODE solvers. In general, Julia appears to use longer step-length and thus have fewer steps. This could be because most Julia solvers include interpolation of the solution for improved accuracy.

Finally, the model implementation with tailor-made RK4 solver (TM-RK4) was converted to C-code/a DLL using MATLAB Coder from MathWorks. This DLL re-

Table 1. Comparison of run time for the various implementations. TM: tailor-made, M: MATLAB, J: Julia. Columns for MATLAB and Julia show absolute simulation time in seconds, and in parenthesis: computation time relative to the fastest combination. Finally, the right-most column shows the relative run-time of MATLAB vs. Julia. In all solvers except the fixed step-length TM-RK4 algorithm, absolute tolerance 10^{-6} and relative tolerance 10^{-3} was used.

Algorithm	MATLAB	Julia	MATLAB/Julia
TM-RK4	171.2 s (36)	40.3 s (8.6)	4.25
M-ode45, J-DP5	29.4 s (6.3)	6.2 s (1.3)	4.74
M-ode23, J-BS3	40.7 s (8.7)	6.0 s (1.3)	6.78
M-ode113, J-VCABM	27.4 s (5.8)	4.7 s (1.0)	5.83
M-ode15s	59.3 s (13)	–	–
M-ode23t	63.9 s (14)	–	–
M-ode23tb	88.6 s (19)	–	–
J-Tsit5	–	6.9 s (1.5)	–

turned $f(x,t)$ in the ODE $\frac{\mathrm{d}x}{\mathrm{d}t} = f(x,t)$ and was then called virtually without overhead from Julia in an Euler integration loop, and the execution speed was compared to the TM-RK4 method implemented in Julia and called in an identical Euler integration loop. In this case, the pure Julia implementation was ca. 12% *faster* than the C-code/DLL. To have a 100% fair speed comparison, the model should have been implemented 100% in C and Julia by experts in the respective languages. However, the for loop in Julia for doing Euler integration is efficient, with most of the computational load taking place in computing $f(x,t)$. Because of this, we believe that execution speed in Julia is relatively similar to what can be achieved in C for this type of problem.

4 Model linearization in Julia

For model-based control design, a linear approximation of a model is often sought in form

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

where matrices A,B, C, and D are found as the following Jacobians at the operating point $*$ given by (u^*, x^*):

$$\begin{aligned}A &= \left.\frac{\partial f(x,u,t;\theta)}{\partial x}\right|_* \\ B &= \left.\frac{\partial f(x,u,t;\theta)}{\partial u}\right|_* \\ C &= \left.\frac{\partial g(x,u,t;\theta)}{\partial x}\right|_* \\ D &= \left.\frac{\partial g(x,u,t;\theta)}{\partial u}\right|_*.\end{aligned}$$

The linear approximation is believed to give good approximation to the nonlinear model as long as the perturbations in the system are "small" relative to the operating point given by (u^*, x^*).

For complex models, it has traditionally been laborious to develop a linear approximation. However, some modern languages has support for Automatic Differentiation for exact linearization. Commercial language MATLAB has support for this, but since Julia is a free language, it is of interest to see how this can be done in Julia. Julia has several packages for carrying out automatic differentiation; here we use package `ForwardDiff.jl` . Assume that we have found a steady operating point (u^*, x^*). Next, we formulate specialized models for the vector fields $f(x,u)$ and $g(x,u)$:

$$\begin{aligned}f_x(x) &: x \to f(x,u^*) \\ f_u(u) &: u \to f(x^*,u) \\ g_x(x) &: x \to g(x,u^*) \\ g_u(u) &: u \to g(x^*,u).\end{aligned}$$

By associating `f_x`$= f_x$, `f_u`$= f_u$, `g_x`$= g_x$, `g_u`$= g_u$, as well as `x_ast`$= x^*$ and `u_ast`$= u^*$, we can compute matrices A, B, C, D as follows:

```
using ForwardDiff
A = ForwardDiff.jacobian(f_x, x_ast);
B = ForwardDiff.jacobian(f_u, u_ast);
C = ForwardDiff.jacobian(g_x, x_ast);
D = ForwardDiff.jacobian(g_u, u_ast);
```

We can now compare transients of the nonlinear model and the linear model from a chosen steady state, and compare, e.g., the final time mass distribution $M_{1:N_\mathrm{p}}$ in the effluent, and the output $d_{50,\mathrm{e}}$. Figure 3 shows the comparison with a 10% increase in the input u.

Figure 4 shows the comparison with a 50% increase in the input u.

Figure 3 shows only 0.08 % error in $d_{50,\mathrm{e}}$ in the linear approximation for 10% change in the input u, which is acceptable for many applications. Figure 4 shows 1.58 % error in $d_{50,\mathrm{e}}$ in the linear approximation for 50% change in the input u, which perhaps is a large error for prediction purposes, but even with this error, the indicated response is not too different from the nonlinear model, and this error can probably be compensated for by feedback control.

5 Conclusions

The outset of this study was an interest in comparing the run-time for differential equation solvers for different lan-

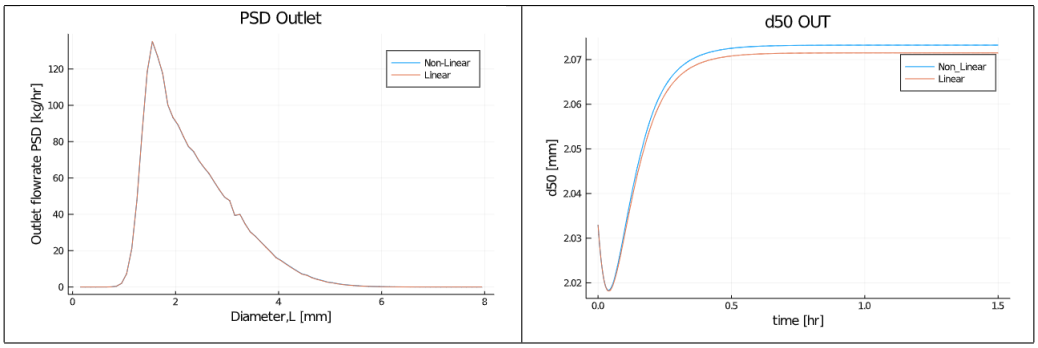

Figure 3. Comparison between linear and nonlinear models where both inputs increase 10%.

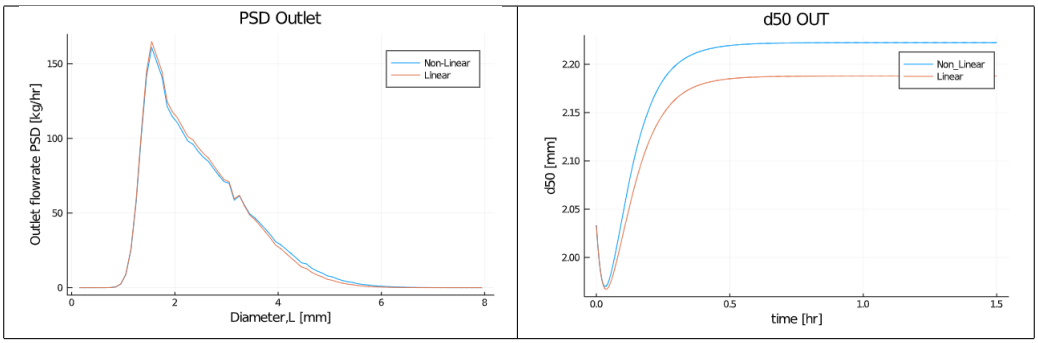

Figure 4. Comparison between linear and nonlinear models where both inputs increase 50%.

guages and different implementations. The test model is a population balance model for the granulation drum of fertilizer production, i.e., without product recycling. Such models are of high order, and are relatively complex to solve numerically. In order to use such a model in optimization based control algorithms, it must be possible to solve the model much faster than real time, so execution time is an issue.

The original implementation of the model under study was in MATLAB, and used a tailor-made implementation of a 4th order Runge-Kutta solver in order to store/present on-line the output, which is the median of the particle diameter distribution. This implementation was transferred to Julia for comparison.

In an extension of this rewrite to Julia, the MATLAB code was rewritten from the tailor-made solver implementation to a form which can use the standard MATLAB ODE solvers. Likewise, the Julia code was rewritten to take advantage of the Julia solvers in package DifferentialEquations.jl. In both cases, every attempt was made to make the code efficient, e.g., taking advantage of vectorization in MATLAB.

Speed comparisons indicate that Julia is typically ca. 5 times faster than MATLAB. Similarly, the results indicate that utilizing the built-in solvers in the languages is in the order of 6 times faster than using the tailor-made solvers. To this end, the fastest Julia implementation is approximately 36 times faster than the original, tailor-made MATLAB implementation.

Some experiments have been made with Julia DAE solvers instead of the ODE solvers. The advantage with DAE solvers is that one can compute outputs on-line instead of by post processing. Although a thorough comparison has not been carried out, initial attempts indicate that using DAE solvers approximately doubles the computation time compared to ODE solvers.

Some initial attempts have also been made with implementing the tailor-made solver in C. Specifically, the MATLAB code was converted to C-code/DLL using MATLAB Coder. The resulting DLL can be called from Julia virtually without overhead, and the execution speed was compared to that of the pure Julia code. The result of this comparison was that Julia was ca. 12% *faster* than the C-code implementation of the model.

Overall, the comparison between the three languages MATLAB, Julia, and C indicate that for solving ODEs, Julia and C have relatively similar execution speed, in spite of Julia being a scripted language — although with Just in Time Compilation. Execution in both Julia and C is considerably faster than in MATLAB. Using MATLAB Coder makes it possible to regain the speed advantage of Julia; however, Julia and the Julia eco-system is free.

In addition to speed comparisons, we have checked the possibility of automatic linearization of the Julia code. This can be done using available, free Julia packages, and appeared to be relatively straightforward, with the linear approximation of the granulation model being quite close to the nonlinear solution in realistic cases. This indicates that the linear approximation probably suffices for control design. It is also possible to do such automatic lineariza-

tion in MATLAB. However, because Julia is a free tool while MATLAB is an expensive, commercial tool, exploring that possible is perhaps less interesting.

Future work will focus on implementing Julia and MATLAB code for the extended system with product screening and re-circulation, as well as automatic computation of a linear approximation for control design. In a longer horizon, it is also of interest to consider possibilities for solving the model over a distributed computer network, or on GPUs.

Acknowledgment The economic support from The Research Council of Norway and Yara Technology Centre through project no. 269507/O20 "Exploiting multi-scale simulation and control in developing next generation high efficiency fertilizer technologies (HEFTY)" is gratefully acknowledged

References

Karl Johan Åstrøm and Richard M. Murray. *Feedback Systems. An Introduction for Scientists and Engineers*. Princeton University Press, Princeton, NJ, 2008. ISBN 978-0-691-13576-2.

Mathieu Besanon, David Anthoff, Alex Arslan, Simon Byrne, Dahua Lin, Theodore Papamarkou, and John Pearson. Distributions.jl: Definition and modeling of probability distributions in the juliastats ecosystem. *arXiv*, 2019.

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:10.1137/141000671.

A. Buck, R. Durr, M. Schmidt, and E. Tsotsas. Model predictive control of continuous layering granulation in fluidized beds with internal product classification. *Journal of Process Control*, 45:66–75, 2016.

I.T. Cameron, F.Y. Wang, C.D. Immanuel, and F. Stepanek. Process systems modelling and applications in granulation: A review. *Chemical Engineering Science*, 60:3723–3750, April 2005.

C. Herce, A. Gil, M. Gil, and C. Cortés. A cape-taguchi combined method to optimize a npk fertilizer plant including population balance modeling of granulation-drying rotary drum reactor. *Computer Aided Chemical Engineering*, 40:49–54, 2017.

S.M. Iveson, J.D. Litster, K. Hapgood, and B.J. Ennis. Nucleation, growth and breakage phenomena in agitated wet granulation processes: a review. *Powder Technology*, 117(1–2): 3–39, 2001.

Jim Litster and Bryan Ennis. *The Science and Engineering of Granulation Processes*, volume 15 of *Particle Technology Series*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004. ISBN 1-4020-1877-0.

Jan M. Maciejowski. *Predictive Control with Constraints*. Prentice Hall, Harlow, England, 2002.

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:10.5334/jors.151.

R. Ramachandran and A. Chaudhury. Model-based design and control of a continuous drum granulationprocess. *Chemical Engineering Research and Design*, 90(8):1063–1073, 2012.

R. Ramachandran, C.D. Immanuel, F. Stepanek, J.D. Litster, and F.J. Doyle, III. A mechanistic model for breakage in population balances of granulation: Theoretical kernel development and experimental validation. *Chemical Engineering Research and Design*, 87(4):598–614, 2009.

D. Ramkrishna. *Population Balances. Theory and Applications to Particulate Systems in Engineering*. Academic Press, London, 2000.

J. Revels, M. Lubin, and T. Papamarkou. Forward-Mode Automatic Differentiation in Julia. *arXiv:1607.07892 [cs.MS]*, 2016. URL `https://arxiv.org/abs/1607.07892`.

G. Valiulis and R. Simutis. Particle growth modelling and simulation in drum granulator-dryer. *Information Technology and Control*, 28(2), 2009.

Ludmila Vesjolaja, Björn Glemmestad, and Bernt Lie. Population balance modelling for fertilizer granulation process. In Lars Erik Øi, Tiina Komulainen, Robin T. Bye, and Lars O. Nord, editors, *Proceedings of the 59th Conference on Simulation and Modelling*, pages 95–102, Oslo Metropolitan University, Oslo, Norway, September 2018. SIMS, Linköping University Electronic Press. doi:http://doi.org/10.3384/ecp181531.

F.Y. Wang and Ian T. Cameron. A multi-form modelling approach to the dynamics and control of drum granulation processes. *Powder Technology*, 179(1–2):2–11, 2007.

F.Y. Wang, X.Y. Ge, N. Balliu, and I.T. Cameron. Optimal control and operation of drum granulation processes. *Chemical Engineering Science*, 61(1):257–267, 2006.