NRC·CNRC

NRC Publications Archive
Archives des publications du CNRC

ADM1jl: a Julia implementation of the anaerobic digestion model 1

Allen, Courtney; Mazanko, Alexandra; Abdehagh, Niloofar; Eberl, Hermann

This publication could be one of several versions: author's original, accepted manuscript or the publisher's version. / La version de cette publication peut être l'une des suivantes : la version prépublication de l'auteur, la version acceptée du manuscrit ou la version de l'éditeur.
For the publisher's version, please access the DOI link below./ Pour consulter la version de l'éditeur, utilisez le lien DOI ci-dessous.

Publisher's version / Version de l'éditeur:
https://doi.org/10.1016/j.softx.2024.101682
SoftwareX, 26, pp. 1-6, 2024-03-25

NRC Publications Archive Record / Notice des Archives des publications du CNRC :
https://nrc-publications.canada.ca/eng/view/object/?id=ad080b8b-4e95-4bc8-9556-4ff1710caa55
https://publications-cnrc.canada.ca/fra/voir/objet/?id=ad080b8b-4e95-4bc8-9556-4ff1710caa55

Access and use of this website and the material on it are subject to the Terms and Conditions set forth at
https://nrc-publications.canada.ca/eng/copyright
READ THESE TERMS AND CONDITIONS CAREFULLY BEFORE USING THIS WEBSITE.

L'accès à ce site Web et l'utilisation de son contenu sont assujettis aux conditions présentées dans le site
https://publications-cnrc.canada.ca/fra/droits
LISEZ CES CONDITIONS ATTENTIVEMENT AVANT D'UTILISER CE SITE WEB.

Questions? Contact the NRC Publications Archive team at PublicationsArchive-ArchivesPublications@nrc-cnrc.gc.ca. If you wish to email the authors directly, please see the first page of the publication for their contact information.

Vous avez des questions? Nous pouvons vous aider. Pour communiquer directement avec un auteur, consultez la première page de la revue dans laquelle son article a été publié afin de trouver ses coordonnées. Si vous n'arrivez pas à les repérer, communiquez avec nous à PublicationsArchive-ArchivesPublications@nrc-cnrc.gc.ca.

National Research Council Canada | Conseil national de recherches Canada

Canada

SoftwareX 26 (2024) 101682

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

ADM1jl: A Julia implementation of the Anaerobic Digestion Model 1

Courtney Allen [a,*], Alexandra Mazanko [a], Niloofar Abdehagh [b], Hermann Eberl [a]

[a] University of Guelph, 50 Stone Road East, Guelph, Ontario, N1G 2W1, Canada
[b] National Research Council Canada, 1200 Montreal Road, Building M-58, Ottawa, Ontario, K1A 0R6, Canada

ARTICLE INFO

Keywords:
IWA Anaerobic Digestion Model 1
Anaerobic digestion
Simulation software
Julia Programming Language

ABSTRACT

The Anaerobic Digestion Model 1 is a system of differential equations that was developed by an International Water Association task group to describe the processes of anaerobic digestion. An implementation of the Anaerobic Digestion Model 1 (ADM1) must be as computationally fast and flexible as possible. The ADM1jl package was designed with those requirements in mind, exploiting the Julia Programming Language's computational speed to create a programme that is both fast (between 15 and 800 times faster than other implementations tested, depending on the implementation being compared to) and user friendly.

Code metadata

Current code version	1.0.6
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-23-00746
Permanent link to Reproducible Capsule	N/A
Legal Code License	MIT License
Code versioning system used	git
Software code languages, tools, and services used	Julia Programming Language
Compilation requirements, operating environments & dependencies	Julia 1.6.0 or higher
If available Link to developer documentation/manual	https://courta96.github.io/ADM1jl/
Support email for questions	callen15@uoguelph.ca

1. Motivation and significance

To curb the devastating effects of climate change it is necessary to shift away from fossil fuels towards renewable energy sources. One such renewable energy source is methane. Methane is a product of anaerobic digestion; so wastewater treatment plants that use anaerobic digestion can capture the methane produced to use as a biofuel [1,2]. Mathematical models, such as the Anaerobic Digestion Model 1 (ADM1) [3], have been designed to simulate the processes of anaerobic digestion to allow users to better understand the effects of different parameter choices on anaerobic digestion. For example, a user who wants to optimize the production of methane in a physical reactor can perform sensitivity analyses on ADM1 to better understand which reactor parameters need to be changed. However, sensitivity analyses can require thousands of model simulations. Users may also wish to use ADM1 for optimal control problems, or automatic parameter calibrations, both of which are also computationally intensive. Therefore, implementations of ADM1 must be both as computationally efficient as possible and flexible to user demand. The ADM1jl package was created to satisfy both of those requirements.

1.1. Anaerobic digestion

Anaerobic digestion is the process by which anaerobic microorganisms decompose organic material. Many different bacterial groups contribute to anaerobic digestion, leading to a chain of conversion processes where one bacterial group consumes organic waste and produces a substrate which is consumed by another bacterial group, and so on [4]. The end product in this chain of processes is methane,

* Corresponding author.
E-mail addresses: callen15@uoguelph.ca (Courtney Allen), amazanko@uoguelph.ca (Alexandra Mazanko), niloofar.abdehagh@nrc-cnrc.gc.ca (Niloofar Abdehagh), heberl@uoguelph.ca (Hermann Eberl).

https://doi.org/10.1016/j.softx.2024.101682
Received 2 November 2023; Received in revised form 28 February 2024; Accepted 8 March 2024
Available online 25 March 2024
2352-7110/© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

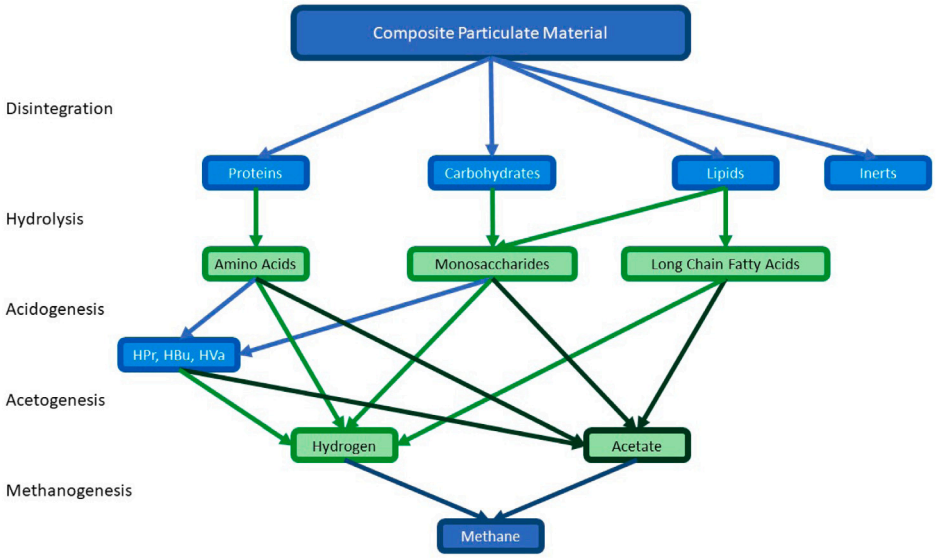

Fig. 1. A flowchart showing the processes of anaerobic digestion [3]. The abbreviations include HPr (propionic acid), HBu (butyric acid), and HVa (valeric acid).

a major component of biogas. Since methane can be used in biofuel, there has been a renewed interest in the use of anaerobic digestion in recent years [5,6].

In addition to the renewable energy aspect, there are many benefits to using anaerobic digestion as a method of organic waste treatment. The benefits include the ability to treat a wide range of organic waste (including agricultural, municipal, and industrial waste), the ability to efficiently treat large amounts of organic waste, low nutrient requirements, and low production of waste solids [7]. There are some drawbacks however, including low bacterial growth rates, inefficiencies when waste is dilute, and the production of odours [7]. However, the many benefits of anaerobic digestion, including the production of biofuel, are believed to outweigh the drawbacks, and the processes of anaerobic digestion remain an active area of research.

The conversion processes involved in anaerobic digestion are grouped into five categories: the disintegration of the initial particulate material, hydrolysis, acidogenesis, acetogenesis, and methanogenesis [3]. These processes are shown in Fig. 1. To promote the production of methane, users need to know how changes in conditions like feedstock composition and reactor temperature affect these conversion processes. This is what led to the development of mathematical models for each of these processes, and ultimately to the development of ADM1 itself.

1.2. The IWA anaerobic digestion Model no. 1

In 2002, an International Water Association task group published The Anaerobic Digestion Model 1 (ADM1) to model the processes of anaerobic digestion used in wastewater treatment [3]. It is a system of 35 ordinary differential equations with 104 model parameters. ADM1 was based on simpler models of anaerobic digestion that existed at the time, and was created to be a quasi industry standard that could be modified to include or alter processes depending on the requirements of different systems.

The most basic form of ADM1 considers 12 bacterial species and 12 substrates. It is a system of 24 ordinary differential equations (ODEs) that models how each bacterial species consumes and produces substrate. The bacterial species are inhibited by low pH, and the concentration of dissolved gasses in the substrate (carbon dioxide, methane, and hydrogen gas) depends on the pressure of the gasses above the liquid substrate. Both acid/base reactions and liquid/gas exchange, which determine the pH and concentration of dissolved gasses respectively, can be modelled by either ODEs or differential algebraic equations (DAEs). ADM1jl models these reactions as ODEs, meaning the total number of processes increases to 29 and the total number ODEs increases to 35. The ODE system can be represented by the following matrix equation:

$$\frac{d\vec{u}}{dt} = \mathbf{P}\,\vec{r}(\vec{u}) + \mathbf{M}_{\text{in}}\vec{u}_{\text{in}} - \mathbf{M}_{\mathbf{out}}\vec{u}, \tag{1}$$

where $\vec{u} \in \mathbb{R}^{35}$ is the vector of bacterial and substrate concentrations. The remaining terms describe the reactions and contain the 104 model parameters that describe the system. $\vec{r}(\vec{u}) \in \mathbb{R}^{29}$ is the vector of process rates and depends on the concentrations given by $\vec{u}$; $\mathbf{P} \in \mathbb{R}^{35,29}$ is the transpose of the Petersen matrix, which is a sparse matrix that contains the yields of each reaction, thereby describing how the reaction rates affect the bacterial and substrate concentrations; $\vec{u}_{\text{in}} \in \mathbb{R}^{35}$ is the vector of inflow concentrations; $\mathbf{M}_{\text{in}} \in \mathbb{R}^{35,35}$ and $\mathbf{M}_{\text{out}} \in \mathbb{R}^{35,35}$ are diagonal matrices that describe the inflow and outflow rate, respectively. This implementation also includes the minor updates to the carbon balances in the Petersen matrix and to the inhibition forms that are described in the Benchmark Simulation Model no. 2 (BSM2) [8].

1.3. Some other available implementations of ADM1

Due to the renewed interest in anaerobic digestion as a means of biofuel production, there have been several implementations of ADM1 in recent years. These include implementations in Java [9] and Python [6], as well as the original Matlab implementation of ADM1 [8], all of which use the DAE form of ADM1. While the Matlab implementation and one of the Python implementations give the option of using the ODE form of ADM1, the ODE form is significantly slower in both implementations, and is therefore not often used in practice.

This is because modelling the acid/base reactions as ODEs results in a stiff system and therefore requires the use of a stiff solver. Stiff solvers are conventionally believed to be computationally slower than using a non-stiff solver on a system where the stiffness has been relaxed [8]. The DAE form of ADM1 reformulates the acid/base reactions as algebraic equations, relaxing the stiffness of the system [3]. The DAE form of ADM1 does not require the use of stiff solvers to return an accurate result, and for that reason DAE implementations of ADM1 are widely believed to be compuationally faster than ODE implementations [8], so the DAE form is often the one chosen for implementations. However, the results of a comparison of ADM1jl with the Java and Python codes [10] show that ADM1jl is computationally faster. ADM1jl is 15 times faster than the Java implementation, and over 800 times faster

than the slowest Python implementation. This suggests that the choice of solver algorithm has a much greater impact on the speed of the implementation than the choice of form.

In addition to the potential drawback in computational speed, existing implementations of the DAE form also have a major drawback: instead of updating the algebraic equations every time step, they only recompute the algebraic equations every so many time steps [8]. This makes the solver faster, since it requires fewer computations, but it can also negate the benefit of the adaptive step size solver methods these implementations use. Conversely, ADM1jl is able to exploit the full benefit of adaptive step size methods due to implementing the ODE form of ADM1.

These implementations also hard-code the ODEs, instead of writing out the matrix form given in Eq. (1). This is likely due to a perceived loss in computational speed resulting from the Petersen matrix being sparse, leading to many multiplications by zero. However, ADM1jl employs the matrix formulation without any noticeable drop in computational speed [10]. The benefit of the matrix formulation is that it makes it easier to adapt the model to different situations. Adding/removing a process is as easy as adding/removing an element of the Petersen matrix, as opposed to going through each equation and adding it manually.

2. Software description

2.1. Software architecture

ADM1jl is a Julia package. In order to make predictions quickly and accurately, an implementation of ADM1 needs to be computationally fast, and flexible to changes made to the base model. ADM1jl outperforms existing implementations of ADM1 in both of these areas by exploiting Julia features and by programming the model to be as adaptable as possible to both casual and experienced users, as has already been demonstrated in [10]. Expanding upon that work, this paper makes the code that was used in [10] publicly available, and goes into greater detail on its use.

The Julia programming language was chosen for its computational speed. Julia was designed for scientific computing and puts an emphasis on optimizing linear algebra operations [11]. The speed of linear algebra operations in Julia, along with other Julia optimizations, are harnessed by Julia's DifferentialEquations package [12], which has been shown to be computationally faster than differential equation solvers in other languages [13] in a comparison performed by the author of the DifferentialEquations package. The DifferentialEquations package was therefore used to solve the ODEs in ADM1jl, and a comparison of our implementation of ADM1 with implementations in other languages [10] corroborates the claims about the speed of Julia's solvers.

ADM1jl was designed to be flexible in its basic functionality. Using the built-in functions and external `.csv` files containing parameters for each reactor, the user can model one reactor or multiple reactors in series, as well as specify the initial conditions, inflow concentrations, solver algorithms, etc. With this information, ADM1jl builds the Petersen matrix, rate vector, and inflow/outflow matrices. ADM1jl then uses the DifferentialEquations package to create and solve the ODE. See Fig. 2 for more information on the architecture of the code.

While many ADM1 implementations only output the solution at a final time, ADM1jl returns the solution as type `ODESolution`, which has two fields. If the solution is called `sol`, then the fields are `sol.t` and `sol.u`. The `sol.t` field is a vector of floats and contains the output time steps. The `sol.u` field is a vector that contains the solution vector at each time step given in `sol.t`. The solution vector is length 35, corresponding to the 35 state variables in ADM1. The documentation contains a table that specifies which vector index corresponds to which state variable. Additionally, a Jupyter notebook containing examples of all of the above is available at: https://github.com/CourtA96/ADM1jl.

2.2. Basic functionality

The function `ADM1sol(tspan,u0,IV)` builds the matrix form of ADM1 and solves the system. `tspan` is the timespan of the desired solution, `u0` is the initial concentration vector, and `IV` is the inflow concentration vector. The inflow vector does not need to be static, and can also vary with time, which is accomplished by defining `IV` as a vector of inflow vectors, where each inflow vector corresponds to a time in `tspan`. The 104 model parameters can be modified by editing the file `model_parameters.csv` and calling the `ADM1sol` function again. By default, the `Rodas4P()` algorithm, a 4th order A-stable stiffly stable solver method, is used to solve the system. However, other algorithms in the DifferentialEquations package can be specified by the user as optional function arguments, allowing the user to experiment with how solver algorithms affect solution time. The output is a two element vector that contains the `ODESolution` and the time in seconds it took to solve the system.

To solve a system with multiple reactors in series, where the outflow of the first reactor is the inflow of the second, and so on, the function `MultiChamberSolution` is used. Similarly to `ADM1sol`, it takes as input: the timespan, the initial conditions for each reactor, and the inflow to the first reactor as inputs, along with the number of reactors in series. `MultiChamberSolution` has all of the same flexibility as `ADM1sol`, allowing the user to vary the inflow concentration to the first reactor, to specify the solver algorithm used, the parameters for each reactor, and so on.

ADM1jl also includes a simple plotting function. The solution can be plotted using the `plotSols(sol)` function, which takes an 'ODESolution' as input and returns a plot showing how the concentration of each of the 35 variables changes with time. For example, see Fig. 3.

2.3. Extended functionality

A user who is familiar with both the structure of ADM1 and the Julia Programming Language may want to make modifications to the system beyond what a basic user would need. Since ADM1jl was written in matrix form, editing the `transportmatrix_definition` ($\mathbf{M_{out}}$ in Eq. (1)), `reactionrates` ($\vec{r}(\vec{u})$ in Eq. (1)), and `petersenmatrixtranspose_definition` ($\mathbf{P}$ in Eq. (1)) functions is relatively straight-forward.[1] For example, the vector of reaction rates is defined as `rrates` in the definition of the `reactionrates` function. The `rrates` vector has the same elements as the reaction rates vector given in tables 3.1 and 3.2 of the Anaerobic Digestion Model 1. So a user that wants see what will be the effect of making decay rates proportional to the squares of the bacterial concentrations can open the `matrix_definitions.jl` file and edit the `reactionrates` function at lines 233–239, squaring the `sx[i]` terms in `rrates[13]` through `rrates[19]`. See Listing 1 for reference.

To alter the `transportmatrix_definition` and `petersenmatrixtranspose_definition` functions, the process is much the same as the above. Both functions are defined in the `matrix_definitions.jl` file. The `transportmatrix_definition` function returns a diagonal matrix, and the flow rates are defined in a 35 element vector, `TMtemp`. The `petersenmatrixtranspose_definition` function returns a sparse matrix where the elements are described by the vectors `P1`, `P2`, and `Q`, which specify the column, row, and value of each element in the Petersen matrix, respectively. Altering the matrices is therefore as simple as adding an element to the desired vector(s).

[1] Note that altering the code is only suggested for users that want to make fundamental changes to the ADM1 system itself. Minor changes to parameter values, for example setting decay rates of bacterial species to zero, can be easily achieved by changing parameter values in the `model_parameters.csv` file. In the case of setting decay rates to zero, this would be done by setting the parameters `k_m_i` to zero, where `i` indicates the bacterial species.

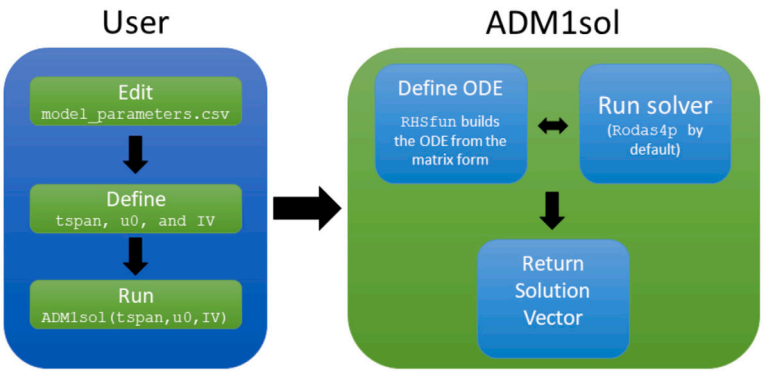

Fig. 2. A flowchart showing the architecture of the `ADM1sol` function. The file `model_parameters.csv` must be in the working directory in order for the function to run.

Listing 1: Snippet of the code for the `reactionrates` function.

```
    "Final inhibition factors:"

    I = I_pH*I_IN # I_1 in tables 3.1-3.2 of ADM1
    II = I[1]*I_h2 # I_2 in tables 3.1-3.2 of ADM1
    III = I[2]*I_nh3 # I_3 in tables 3.1-3.2 of ADM1

    "Reaction Rates"

    rrates=Array{Real}(undef,NREAC)
    rrates[1]= bp[1]*sx[13] # k_dis * X_xc
    rrates[2]= bp[2]*sx[14] # k_hyd_ch * X_ch
    rrates[3]= bp[3]*sx[15] # k_hyd_pr * X_pr
    rrates[4]= bp[4]*sx[16] # k_hyd_li * X_li
    rrates[5]= (bp[10]*(monod(sx[1],bp[11]))*sx[17])*I[1] # k_m_su *(S_su/(K_S_su+S_su)*X_su*I_1
    rrates[6]= (bp[13]*(monod(sx[2],bp[14]))*sx[18])*I[1] # k_m_aa *(S_aa/(K_S_aa+S_aa)*X_aa*I_1
    rrates[7]= (bp[16]*(monod(sx[3],bp[17]))*sx[19])*II[1] # k_m_fa *(S_fa/(K_S_fa+S_fa)*X_fa*I_2
    rrates[8]= (bp[20]*(monod(sx[4],bp[21]))*sx[20]*(sx[4]/(sx[4] + sx[5] + 10.0^(-6.0))))*II[2] # k_m_c4
        *(S_va/(K_S_va+S_va)*X_c4*(1/(1+(S_bu/S_va)))*I_2
    rrates[9]= (bp[20]*(monod(sx[5],bp[21]))*sx[20]*(sx[5]/(sx[5] + sx[4] + 10.0^(-6.0))))*II[2] # k_m_c4
        *(S_bu/(K_S_bu+S_bu)*X_c4*(1/(1+(S_va/S_bu)))*I_2
    rrates[10]= (bp[24]*(monod(sx[6],bp[25]))*sx[21])*II[3] # k_m_pr *(S_pro/(K_S_pro+S_pro)*X_pro*I_2
    rrates[11]= (bp[28]*(monod(sx[7],bp[29]))*sx[22])*III # k_m_ac *(S_ac/(K_S_ac+S_ac)*X_ac*I_2
    rrates[12]= (bp[34]*(monod(sx[8],bp[35]))*sx[23])*I[3] # k_m_h2 *(S_h2/(K_S_h2+S_h2)*X_h2*I_2

    # The decay rates:

    rrates[13] = bp[39]*sx[17] # k_dec_su * X_su
    rrates[14] = bp[40]*sx[18] # k_dec_aa * X_aa
    rrates[15] = bp[41]*sx[19] # k_dec_fa * X_fa
    rrates[16] = bp[42]*sx[20] # k_dec_c4 * X_c4
    rrates[17] = bp[43]*sx[21] # k_dec_pro * X_pro
    rrates[18] = bp[44]*sx[22] # k_dec_ac * X_ac
    rrates[19] = bp[45]*sx[23] # k_dec_h2 * X_h2
```

3. Illustrative examples

3.1. Basic example

The code in Listing 2 shows a basic use case with a single reactor where the timespan is 200 days and the default inflow vector and initial conditions are called. The solution is solved using the `Rosenbrock23()` algorithm defined in the DifferentialEquations package. The solution is then plotted with the title "Example Plots" and the plots are saved as `.png` files to the current directory with the file names `Example Plots (1 of 2).png` and `Example Plots (2 of 2).png`. The resulting plots are shown in Fig. 3.

Listing 2: Basic use case. The DifferentialEquations package is called so that the user can specify the solver algorithm. If no solver algorithm is specified, the default algorithm is `Rodas4P()`.

```
julia> using ADM1jl

julia> using DifferentialEquations

julia> IV = inflowvector_definition(); # assigns the
    default inflow vector to IV

julia> u0 = initialConditions(); # assigns the default
    initial conditions to u0

julia> tspan = (0.0,200.0); # the solution will be computed
    from t=0.0 to t=200.0
```

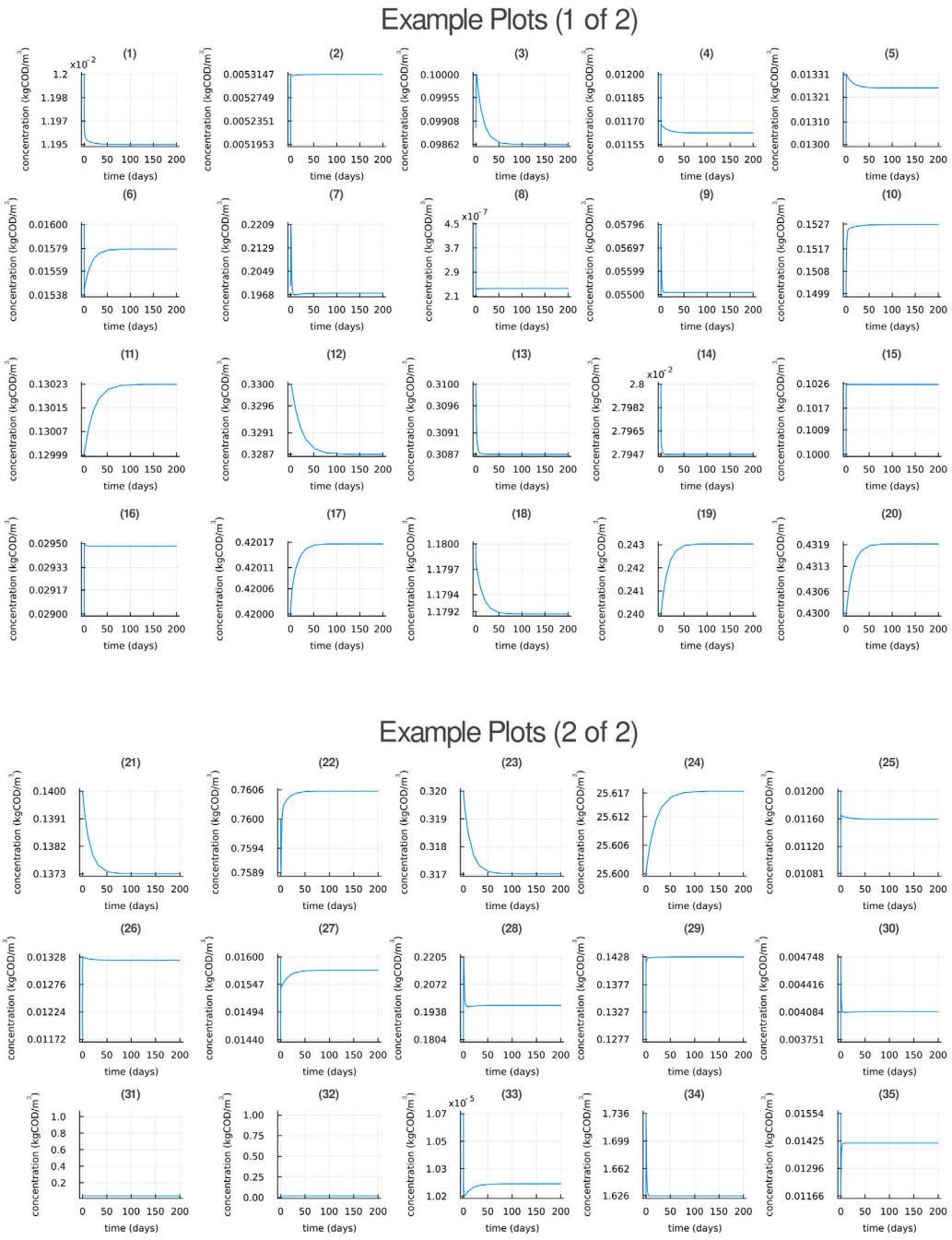

Fig. 3. Example plots of the code given in Section 3 demonstrating the output of ADM1jl's `plotSols` function, which is intended to give a general sense of the dynamics of the system. Each plot has an index corresponding to a state variable and shows how the concentration of the variable changes with time. To see which index corresponds to which state variable, consult the online documentation. See the Supplementary Material for a higher resolution version of this plot.

```
julia> sol, tSol = ADM1sol(tspan,u0,IV,alg=Rosenbrock23
    ()); # compute the solution with the Rosenbrock23()
    algorithm and save it to sol, the time to solve is
    saved to tSol

julia> plotSols(sol, titleText="Example Plots",savePNG=
    true) # plot the solution and save to
    .png
```

3.2. Advanced example

The code in Listing 3 solves a three chamber system for 50 days where the inflow to the first chamber is read in from a `.csv` file named `inflow.csv`. The CSV and DataFrames packages are needed to read in the data. The `inflow.csv` file breaks the 50 day timespan into 0.1 day increments and specifies the inflow vector at each time step. Each reactor in the series has the same default initial conditions. As opposed to the previous example, no solver algorithm is specified, so the default solver algorithm (`Rodas4P()`) is used to solve the system. The plots of the solution to the first chamber are then displayed.

Listing 3: Multi-chamber use case where the inflow to the first chamber in the series is variable and read in from a `.csv` file.

```
julia> using ADM1jl

julia> using CSV, DataFrames
```

```
julia> inflow = CSV.read("inflow.csv",DataFrame); # read
    in the inflow data from a .csv file

julia> M = Matrix(inflow); # convert data to matrix form

julia> IV = [M[1:35,i] for i in 1:size(M)[2]]; # define
    vector of inflow vectors

julia> t = [M[36,i] for i in 1:size(M)[2]]; # define vector
    of timesteps

julia> u0 = initialConditions(); # default initial
    conditions

julia> sols = MultiChamberSolution((t[1],t[end]),(u0,u0,
    u0),IV,t,3); # compute the solution for three
    reactors

Finished Chamber 1
Finished Chamber 2
Finished Chamber 3

julia> plotSols(sols[1],titleText="Plot of Variable
    Inflow") # plot Chamber 1 solution with title "Plot of
    Variable Inflow"
```

4. Impact and conclusions

ADM1jl is a new implementation of ADM1 that seeks to improve on the computational time and flexibility of previous ADM1 implementations. It was designed to be accessible for users and to be used in real-world applications. It is also flexible to user demand, both through function calls and the easy adaptability afforded by the implementation of the matrix formulation. ADM1jl is able to achieve this flexibility without compromising on computation time. In comparison with other implementations in Java and Python, ADM1jl has comparable accuracy but is computationally faster. For more details on this comparison, including details on the benchmarking process, see [10].

The improved compute time given by ADM1jl, will be beneficial when using the model in situations where a large amount of simulation data needs to be generated. Potential such applications include optimal control problems, automatic parameter calibrations, sensitivity analyses, optimizing treatment efficiency, and other situations where many simulations need to be carried out. The computation speed could also allow the model to be connected to a physical reactor to make predictions and recalibrations in real time based on the physical conditions of the reactor and its feedstock. The ability to perform these analyses and make predictions allows users to optimize physical reactors for the production of methane, enabling the efficient production of renewable energy.

Abbreviations

The following abbreviations are used in this manuscript:

ADM1	Anaerobic Digestion Model 1
ODE	Ordinary Differential Equation
DAE	Differential Algebraic Equation
HPr	Propionic Acid
HBu	Butyric Acid
HVa	Valeric Acid

Funding sources

This work was supported by the Natural Sciences and Engineering and Research Council of Canada (NSERC) [grant number ALLRP 560564], and the Ontario Centre of Innovation (OCI) [grant number #34231].

CRediT authorship contribution statement

Courtney Allen: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Alexandra Mazanko:** Writing – review & editing, Validation, Software, Formal analysis. **Niloofar Abdehagh:** Writing – review & editing, Resources, Project administration, Investigation, Funding acquisition, Conceptualization. **Hermann Eberl:** Writing – review & editing, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A link to the GitHub repository containing the code and all other data is given in the "code metadata" table in the article.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.softx.2024.101682.

References

[1] Kunatsa T, Xia X. A review on anaerobic digestion with focus on the role of biomass co-digestion, modelling and optimisation on biogas production and enhancement. Bioresour Technol 2022;344:126311. http://dx.doi.org/10.1016/j.biortech.2021.126311.

[2] Uddin MM, Wright MM. Anaerobic digestion fundamentals, challenges, and technological advances. Phys Sci Rev 2022. http://dx.doi.org/10.1515/psr-2021-0068.

[3] Batstone D, Keller J, Angelidaki I, Kalyuzhnyi S, Pavlostathis S, Rozzi A, et al. Anaerobic digestion model no. 1 (ADM1). In: Scientific and technical report; no. 13, London: IWA Task Group for Mathematical Modelling of Anaerobic Digestion Processes; 2002.

[4] Meegoda JN, Li B, Patel K, Wang LB. A review of the processes, parameters, and optimization of anaerobic digestion. Int J Environ Res Public Health 2018. http://dx.doi.org/10.3390/ijerph15102224.

[5] Pettigrew L, Gutbrod A, Domes H, Groß F, Méndez-Contreras JM, Delgado A. Modified ADM1 for high-rate anaerobic co-digestion of thermally pre-treated brewery surplus yeast wastewater. Water Sci Technol 2017;76(3–4):542–54. http://dx.doi.org/10.2166/wst.2017.227.

[6] Sadrimajd P, Mannion P, Howley E, Lens PNL. PyADM1: a Python implementation of Anaerobic Digestion Model No. 1. 2021, http://dx.doi.org/10.1101/2021.03.03.433746, bioRxiv, URL https://www.biorxiv.org/content/10.1101/2021.03.03.433746v1.

[7] Rittmann BE, McCarty PL. Environmental Biotechnology: Principles and Applications. New York: McGraw-Hill; 2020.

[8] Alex J, Benedetti L, Copp J, Gernaey K, Jeppsson U, Nopens I, et al. Benchmark Simulation Model no. 2 (BSM2). International Water Association, Modelling and Integrated Assesment; 2008, URL http://iwa-mia.org/benchmarking/#BSM2.

[9] Pettigrew L, Hubert S, Groß F, Delgado A. Implementation of dynamic biological process models into a reference net simulation environment. In: Rabe M, Clausen U, editors. ASIM dedicated conference - simulation in production and logistics. Germany: Fraunhofer IRB Verlag; 2015, p. 651–60, URL https://www.asim-gi.org/fileadmin/user_upload_asim/ASIM_Publikationen_OA/AM157_OA/AM_157_ASIM_SPL_Dortmund_2015_Dortmund_Tagungsband_OA.pdf.

[10] Allen C, Mazanko A, Abdehagh N, Eberl HJ. A new ODE-based Julia implementation of the Anaerobic Digestion Model No. 1 greatly outperforms existing DAE-based Java and Python implementations. Processes 2023;11(7). http://dx.doi.org/10.3390/pr11071899.

[11] Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: A fresh approach to numerical computing. SIAM Rev 2017;59(1):65–98. http://dx.doi.org/10.1137/141000671.

[12] Rackauckas C, Nie Q. DifferentialEquations.jl–A performant and feature-rich ecosystem for solving differential equations in Julia. J Open Res Softw 2017;5(1). http://dx.doi.org/10.5334/jors.151.

[13] Rackauckas C. A comparison between differential equation solver suites in MATLAB, R, Julia, Python, C, Mathematica, Maple, and Fortran. The Winnower 2018. http://dx.doi.org/10.15200/winn.153459.98975.