Article

A New ODE-Based Julia Implementation of the Anaerobic Digestion Model No. 1 Greatly Outperforms Existing DAE-Based Java and Python Implementations

Courtney Allen [1], Alexandra Mazanko [1], Niloofar Abdehagh [2] and Hermann J. Eberl [1,*]

[1] Department Mathematics and Statistics, University of Guelph, Guelph, ON N1G 2W1, Canada; callen15@uoguelph.ca (C.A.); amazanko@uoguelph.ca (A.M.)
[2] CHFour Biogas, Manotick, ON K4M 1A4, Canada; nabdehagh@chfourbiogas.com
* Correspondence: heberl@uoguelph.ca

Abstract: The Anaerobic Digestion Model 1 is the quasi-industry standard for modelling anaerobic digestion, and it has seen several new implementations in recent years. It is assumed that these implementations would give the same results; however, a thorough comparison of these implementations has never been reported. This paper considers four different implementations of ADM1: one in Julia, one in Java, and two in Python. The Julia code is a de novo implementation of the ODE formulation of ADM1 that is reported here for the first time. The existing Java and Python codes implement the more common DAE formulation. Therefore, this paper also examines how DAE implementations compare to ODE implementations in terms of computational speed as well as solutions returned. As expected, the ODE and DAE forms both return comparable solutions. However, contrary to popular belief, the Julia ODE implementation is faster than the DAE implementations, namely by one to three orders of magnitude of compute time, depending on the simulation scenario and the reference implementation used for comparison.

Keywords: anaerobic digestion; ADM1; Julia programming language; Java programming language; Python programming language; numerical methods; performance comparison

check for updates

Citation: Allen, C.; Mazanko, A.; Abdehagh, N.; Eberl, H.J. 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*, 1899. https://doi.org/10.3390/pr11071899

Academic Editors: Pietro Bartocci, Qing Yang and Francesco Fantozzi

Received: 31 May 2023
Revised: 18 June 2023
Accepted: 20 June 2023
Published: 24 June 2023

1. Introduction

We present a de novo implementation of the Anaerobic Digestion Model 1 (ADM1) in the Julia programming language and compare our implementation against existing implementations in Python and Java. Our implementation is based on the ordinary differential equation form of ADM1 and uses the Petersen matrix formulation for better modularity and to more easily allow for future modifications. This is in contrast to existing implementations that hard-code the differential equations directly and are based on the Differential Algebraic Equation (DAE) form of ADM1, which is assumed to be faster than the Ordinary Differential Equation (ODE) implementation. When coding a tool for computer simulation, assumptions are made that are often taken for granted—assumptions about the computational speed of different numerical methods, for example. This conventional wisdom is often taken as fact, which limits experimentation. If one method is *known* to be faster than another, then why experiment with the other method? The issue with this is that these assumptions are often not rigorously tested but are treated as fact. Comparisons between different numerical methods, such as the one performed in this paper, are important because they challenge such assumptions and allow for deeper insights into the computational processes that underlie the simulation.

1.1. Anaerobic Digestion

Anaerobic digestion (AD) refers to the decomposition of organic waste material by anaerobic micororganisms. It is an old environmental engineering technology that received new attention in recent decades, because methane, which is an end product of the process, can be used as a renewable energy source [1,2]. Anaerobic digestion is a broadly applicable technology that allows a variety of feedstock to be processed, for example from agricultural, industrial, or municipal waste [3]. Among the advantages of anaerobic treatment processes are a low production of waste biological solids, low nutrient requirements, the potential to be a net producer of energy, and high organic loading; disadvantages include the low growth of microorgansims, odor production, high buffer requirement for pH control and poor removal efficiency with dilute wastes [4]. The main conversion processes taking place in anaerobic digestion are disintegration of the particulate material, hydrolysis, acidogenesis, acetogenesis, and methanogenesis [5,6]. Extensive mathematical modelling work has been conducted for each of these processes, with the ultimate goal being to assist in the design, analysis and operation and control of anaerobic digestion-based technologies. This culminated in the development of a generic, encompassing process model, the Anaerobic Digestion Model 1, in 2002.

1.2. The Anaerobic Digestion Model 1

The Anaerobic Digestion Model 1 (ADM1) is the quasi-industry standard for modelling the processes of anaerobic digestion that result in the production of methane from waste and wastewater. It was developed by the International Water Association (IWA) and is based on several simpler models of anaerobic digestion [6]. The model considers a continuously stirred tank reactor containing wastewater and 12 different bacterial groups that consume/produce 12 different substrates. This scenario is described by a system of 24 ODEs [6]. The model also considers physio-chemical reactions within the substrate itself, increasing the number of substrates to 23 and increasing the total number of state variables to 35. These physio-chemical reactions are classified into two categories: acid–base reactions and liquid–gas exchange, both of which can be modelled by ODEs, resulting in the total system being described by 35 ODEs.

The basic form of ADM1 is:

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

where $\vec{u} \in \mathbb{R}^{35}$ is the vector of state variables containing the concentration of each bacterial species and substrate; $\vec{r}(\vec{u}) \in \mathbb{R}^{29}$ is a vector of process reaction rates and depends on the concentrations of the components, $\vec{u}$; $\mathbf{P} \in \mathbb{R}^{35\times 29}$ is the transpose of the Petersen matrix, which is a sparse matrix that describes how the concentrations of the components affect the reaction rates; $\vec{u}_{\text{in}} \in \mathbb{R}^{35}$ is the vector of inflow concentrations; and $\mathbf{M_{in}} \in \mathbb{R}^{35\times 35}$ and $\mathbf{M_{out}} \in \mathbb{R}^{35\times 35}$ are diagonal matrices of the inflow and outflow rate, respectively. $\mathbf{P}$, $\vec{r}(\vec{u})$, $\mathbf{M_{in}}$, and $\mathbf{M_{out}}$ contain the 104 model parameters that describe the system. These parameters are listed in the Supplementary Material. Many implementations of ADM1 do not implement ADM1 directly in matrix form and instead opt to hard code each equation directly.

This system of equations is stiff, which typically results in increased computational time due to the need for implicit solvers. In order to reduce the computational time, the stiffness is often relaxed by rewriting the ordinary differential equations as differential algebraic equations [7], which allows for explicit solvers to be used. It appears to be generally accepted that the DAE form of ADM1 is faster than the ODE form, since that was the purpose of its development [7].

1.3. Benchmark Simulation Model 2

The Benchmark Simulation Model 2 (BSM2) is a modified version of ADM1 that was produced by the IWA [7]. The modifications that the BSM2 makes include, for example, modifications to the equations for inorganic carbon and inorganic nitrogen. The BSM2 also supplies standard parameter values and inflow concentration values to provide a benchmark for future research. All of the following implementations are based on the system of equations given by the BSM2, which means that they should give comparable results for the same inputs.

1.4. Purpose of Comparison

In recent years, there has been a call to examine the so-called reproducibility crisis in science [8,9]. This crisis generally refers to the "Three R's", which we will define as Repeatibility, Reproducibility, and Replicability. Although there is some variation in the words themselves, their meanings are generally fixed. Repeatability is often interchanged with Reliability, Robustness, or Rigour, but all three generally refer to the same basic idea: that a result should be consistent and strongly indicative of the real world. Repeatability encompasses Reproducibility and Replicability. A result that is reproducible can be obtained by another researcher using the same equipment and methods, and a result that is replicable can be obtained by a researcher re-creating the experiment from scratch using the same methods. The "Three R's" are an important component of scientific research, since research is always built on the work of others. However, they are often neglected in scientific computing, leading to programs that have only been minimally tested being used to develop software solutions, particularly in the case of modelling complex biological systems [8]. This problem is particularly common when modelling complex biological systems, with one 2019 study estimating with 95% confidence that between 0.68% and 6.8% of hydrology articles published in 2017 were reproducible [10]. This in turn leads to software that becomes less and less trustworthy over time. This paper aims to examine the replicability of the codes used to create the Anaerobic Digestion Model 1 and therefore to avoid that build-up of error.

It is generally accepted that the ODE and DAE forms of ADM1 both yield similar results. One IWA report lists the largest relative difference as 10^{-6} and the largest absolute difference as 10^{-5} [11]; however, these numbers are only for three sets of parameter values, and they only apply to the Matlab implementation of the BSM2 [7]. By comparing the Julia, Java, and Python implementations for a larger dataset, it is possible to make more general conclusions about how the ODE and DAE forms compare to each other. Including different DAE implementations in the comparison gives a better sense of the aforementioned "Replicability" of the results. If the difference between the ODE implementation and the DAE implementations is no more significant than the differences between the DAE implementations, then how significant are the differences really? Likewise, the purpose of comparing to two different Python implementations is to gauge the significance of any differences between the Python, Julia, and Java implementations. If there is a greater difference between the two Python implementations than there is between the other solver methods, what does that mean? After it has been established that two implementations give results that differ within acceptable tolerances, the question of computational efficiency, specifically compute time, arises. These questions will be further examined in Section 3, where we will see that our de novo implementation outperforms all three reference implementations.

A further question when comparing different implementations of a model is how easy it is for users to adapt and apply the simulation tool to scenarios beyond the test cases, say to include a new process, or to modify existing process descriptions. While this is difficult to measure quantitatively, this question entered the design of our new implementation of ADM1.

2. Materials and Methods

2.1. The Python Implementations

The two Python implementations are written by Peyman Sadrimajd et al. [12] and were validated in comparison with the MATLAB implementation of BSM2. One of the implementations is older and considers a fixed inflow rate; this will be called the "`SteadPy`" implementation, since it considers a steady inflow. The second Python implementation is newer and considers a variable inflow rate; this will be referred to as "`DynPy`", since this implementation models dynamic inflow. Both of the Python implementations are in the

DAE form and hard code all of the equations directly: that is, they do not code them in matrix form. The only real difference between `DynPy` and `SteadPy` is how they solve the DAE system.

To solve the ODEs, both versions of the Python code use `DOP853`, which is an "explicit Runge–Kutta method of order 8" [13]. `DOP853` is a Python implementation of the `DOP853` algorithm originally written in Fortran. It is an adaptive step-size method that is error controlled. To solve the algebraic equations, the time span is broken into time steps, and at each time step, the Newton–Raphson method is used to solve the algebraic equations, as specified by the BSM2. `SteadPy` recomputes the algebraic equations every 15 min of simulation time. Conversely, `DynPy` takes a vector of time steps and an array of inflow rates, where each row of the array correspond to a time step, and it recomputes the algebraic equations at each time step.

To make `DynPy` model a system with constant inflow values (such as `SteadPy` and the Julia implementation), it is only necessary to keep the row vector of inflow values constant for every time step. To keep the comparison consistent, the vector of time steps for `DynPy` was kept the same as it was originally coded. This vector is given in Table 1. Keeping the inflow vector constant means `DynPy` and `SteadPy` solve the same system with the same numerical methods; the only difference between them is then how often the algebraic equations are recomputed.

Table 1. Vector of time steps used for `DynPy`. After time $t = 10$ days, the time steps increment by 5 until $t = 205$ days.

Time Step (Days)
0
6.27×10^{-5}
0.000689664
0.00693381
0.014230415
0.031775873
0.051610754
0.088711557
0.135732573
0.212098681
0.311804414
0.474542448
0.722405853
1.163712693
2
4
5
10

2.2. The Java Implementation

The Java implementation was written by Liam Pettigrew et al. [14]. It is based on the Matlab code for the BSM2 and implements the DAE version of ADM1. Like the Python codes, it hard codes the equations instead of coding them in matrix form. This team has also produced a modified version of the code [15] that considers changes to the process rates. That version is not considered in this paper.

It uses the `AdamsBashforthIntegrator` class included in the `org.apache.commons.math3.ode.nonstiff` package to solve the ODEs. This class implements an explicit linear multistep solver known as an Adams method [16]. `AdamsBashforthIntegrator` also implements error control using an adaptive step size. Similarly to `SteadPy`, the Java implementation uses the Newton–Rhaphson method suggested by the BSM2 to recompute the algebraic equations every 15 min of simulation time.

2.3. The Julia Implementation

The Julia code was created to meet two main demands: the first to be flexible to model alterations, and the second to offer greater flexibility in the output. The Java and Python implementations both hard code the equations, whereas the Julia code implements the matrix form given by Equation (1). The matrix form is easier to make changes to, since it only requires editing discrete entries of the Petersen matrix and rate equations instead of going through and editing each equation individually. Additionally, the Java implementation and `SteadPy` only output the system solution at a specified final time. The `DynPy` outputs the solution at discrete times, but these times must be specified by the input, which can lead to a greater build-up of error if the step sizes are chosen to be too large, as can be seen in Section 3. Therefore, simulating a chain of reactors, where the outflow of one reactor becomes the inflow of another, requires additional changes to these programs, and it also introduces the possibility of a build-up of errors.

Since computational time is an important factor in the utility of an ADM1 implementation, the Julia implementation was developed to exploit the purported speed of Julia's DE solvers [17]. In contrast to the other implementations, the Julia implementation is not in DAE form but is instead in ODE form. This choice was made because Julia's ODE solvers are more versatile than its DAE solvers, offering more flexibility in the coding.

The Julia implementation also returns the solution for a range of times t within the specified time range, as opposed to only returning the final solution. Additionally, the Julia implementation exclusively uses adaptive time-stepping methods without having to solve any algebraic equations, so the final solution does not suffer from the same inaccuracies as the dynamic Python implementation, which will be seen in Section 3. After testing several solver methods, `Rodas4P()` was chosen. It is a "4th order A-stable stiffly stable Rosenbrock method with a stiff-aware 3rd order interpolant [18]". How the sole use of an adaptive step size method will impact the accuracy of the solutions of systems with variable inflow, and therefore how it will affect the solutions of multi-reactor systems, will be the subject of another paper. Finally, some optimisations were made, such as using the `Memoize` package's `@memoize` macro on functions that repeatedly take the same inputs. `@memoize` stores the solutions of a function for given inputs in memory, so that the function does not have to be recomputed each time that those inputs are used. The `@profile` macro also found that performing the linear algebra calculations with sparse matrices was a causing bottleneck, so the matrices were all written in full matrix form.

2.4. Null-Hypothesis Significance Testing

To compare the solutions given by each of the implementations, we will use null-hypothesis significance testing. This type of statistical analysis returns a p-value that indicates whether a so-called "alternate hypothesis" can be accepted or rejected. To conduct null hypothesis significance testing, both an alternate hypothesis and a null hypothesis are required. In this case, our alternate hypothesis is that the mean values of our quantities of interest will differ depending on the implementation of ADM1 used to compute them. Our null hypothesis is therefore that the mean values will not differ depending on implementation. If the p-value is close to zero, then we reject the null hypothesis in favour of the alternate hypothesis: that quantities of interest differ depending on the implementation used to compute them.

Commonly used null-hypothesis tests are the Student's t-test and one-way analysis of variance (ANOVA). They compare the mean values of a quantity of interest with respect to their variances. The two tests differ based on the number of groups that are being considered; Student's t-test only considers two groups of data, whereas ANOVA considers more than two groups of data [19].

To use these two tests with large data sets, there needs to be an equality of variance [19]. That is, there must be an equal amount of variation around each of the mean values for both the Student's t-test and ANOVA to return accurate results. To determine if this condition is met, Levene's test [19] can be applied to the data. Levene's test returns a p-value that measures how different the variances of the data sets are from each other. If the p-value returned by Levene's test is significant, i.e., if the p-value is less than 0.05, the variances are not significatly different, and one can conclude there is an equality of variance.

If Levene's test finds an equality of variance, then one can proceed with Student's t-test/ANOVA. However, if that is not the case, a different null-hypothesis test must be used to compare the data. One such test is the Kruskal–Wallis test [20], which is used on non-parametric data, i.e., data that lack an equality of variance.

With all statistical tests, it is important to determine where resulting conclusions stem from. For this reason, post hoc tests are applied to examine the data more thoroughly. In this analysis, two post hoc tests were used to assess results: Student's t-test [21] and Dunn's test [22]. These tests were used to make pairwise comparisons between the means to determine which implementations were significantly different from the others. Student's t-test was performed on the data sets that passed Levine's test, and Dunn's test, a non-paramteric test that functions similarly to the Student's t-test, was performed on the remaining data sets.

2.5. Validating the Julia Code with the Python DAE Implementation

To first validate the code, the Julia implementation was compared against the latest implementation of the Python code, `DynPy`. The same initial conditions, inflow vectors, and model parameters were used in both cases. These parameters are given in the BSM2 and will be referred to as the "default" parameters. The solution was found on the time interval $(0.0, 200, 0)$ since at $t = 200.0$, the solution for these parameters will have had time to reach steady state. The maximum relative difference between the two solutions is defined as

$$D_{\text{rel}} = \max_{i \in [1,35]} \left(\left| \frac{(u_{\text{ju}})_i - (u_{\text{py}})_i}{(u_{\text{py}})_i} \right| \right) \tag{2}$$

where u_{ju} is defined to be the final solution using the Julia code and u_{py} is defined to be the final solution using the Python code. The index i refers to the component of the vector of state variables. Originally, the maximum relative difference was greater than 1000%. The difference decreased when typos were discovered in the Petersen matrix. When the Julia code returned a solution where the maximum relative difference between the solutions was less than 5%, the code was considered validated, and the following more rigorous tests were performed.

2.6. Comparison of Julia, Java, and Python Implementations

2.6.1. Datasets

To perform the comparisons, four different data sets were considered, each of which is based on a different set of model conditions: one where the inflow concentrations were varied within 50% of the BSM2 values, one where the inflow concentrations were varied within 15% of the BSM2 values, and two where the model parameters are varied around the BSM2 values. The two varied-parameter-values data sets are required because the Java implementation does not accept pressure as an input; it needs to be specified by manually. So, one data set varies the pressure, while the other keeps the pressure constant to allow the Java results to be compared to the others. In both data sets where parameter values were varied, some of the parameter values remain constant for all processes. Which parameters were varied, and by what amount, is given in the Supplementary Materials. Each set of model conditions contains 200 vectors each of which contains a set of inflow concentrations or parameter values, depending on the set of model conditions being considered. Each set of model conditions is then used to generate a data set of solutions at $t = 200.0$, which was chosen for the same reason as in Section 2.5, to ensure that the solution has time to reach steady state.

2.6.2. Comparison of Compute Times

To compare the compute times, we used an Acer Swift 3 laptop with an Intel Core i5-8250U CPU and 8 GB of RAM that was running Windows 11 Home. Java 13.0.2, Python 3.8.5, and Julia 1.7.3 were used to run the simulations. We considered three data sets, the BSM2 inflow +/− 50%, the BSM2 inflow +/− 15%, and the random parameter constant pressure dataset; the other random parameter set was omitted, since it could not be run on the Java code without modifying the Java code. Each implementation was run with the first 30 entries of each set of model conditions. The mean computation time was then computed, and the standard deviation was computed using the `STDEV.S` function in Excel, which computes the standard deviation of a sample using the formula

$$\sigma = \sqrt{\frac{\sum_i^n (x_i - \bar{x})}{n-1}}$$

where x_i are sample values, $\bar{x}$ is the sample mean, and n is the sample size [23].

2.7. Statistical Analysis

All statistical analysis was performed using the statistical software included in the R programming language. In order for these tests to be conducted using the R IDE, supplementary packages had to be installed and called when the analysis was conducted.

For each of the four data sets described in Section 2.6.1, three quantities of interest were compared: the weighted average of the solution, the concentration of carbon dioxide gas, and the concentration of methane gas. The weighted average $\bar{u}$ of a solution at time t_{final} is given by

$$\bar{u}(t_{\text{final}}) = \frac{\sum_{i=1}^{35} \frac{u_i(t_{\text{final}})}{u_i(0)}}{35} \tag{3}$$

where $u_i(t_{\text{final}})$ is the solution for state variable i at time t_{final}, so the value of the state variable at time t_{final} is weighted by its initial value, making the weighted average unitless. In this case, as mentioned in Section 2.6.1, $t_{\text{final}} = 200$. The sum is from 1 to 35, since there are 35 state variables in ADM1. It was decided to take a weighted average to attempt to ensure that the value of each state variable affects the mean equally. Since the initial conditions given in the BSM2 were chosen to be close to the steady-state solutions, the initial conditions were also chosen to be the weights.

For each of the three quantities of interest, Levene's test was used to determine if there was an equality of variance between the mean of the values given by each implementation, and then, `ggbetweenstats` was used to perform the statistical analysis and plot the data. The `tibble` package [24] was used to convert the .csv files containing the data sets into `tibble` type data frames that could be interpreted by R. To determine whether the data were parametric or not, the function `leveneTest` from the `car` package [25] was used to perform Levene's test. The `ggbetweenstats` function in the `ggstatsplot` package [26] was then used to plot the data and perform the statistical tests. The `ggstatsplot` package uses functions from various packages to perform the statistical tests and functions from the `ggplot2` package [27] to plot.

To specify whether the `ggbetweenstats` function performs a parametric or non-parametric test, the optional argument `type` is set equal to either `parametric` or `non-parametric`. If `parametric` is specified, an ANOVA test is performed using the function `oneway.test` with the optional argument `var.equal = TRUE`. The Student's t-test is performed using the `pairwise.t.test` function from the `stats` package [28]. If the `non-parametric` argument is specified, the `kruskal.test` from the `stats` performs the Kruskal–Wallis test, and the Dunn test is performed using the `kwAllPairsDunnTest` function from the `PMCMRplus` package [29].

All results were then analysed using the generated p-values to determine if they were statistically significant or not in order to determine if the null hypothesis (that the solutions do not differ based on implementation) could be accepted. The greater the p-value, the less significant the differences between the implementations are. A level of significance, α, is generally chosen, below which the p-value is said to be significant. In this case, the level of significance was α = 0.05, meaning that if the p-value was less than 0.05, then the null hypothesis was rejected.

3. Results

3.1. Comparison of Solutions

We begin our assessment of the new ADM1 implementation by comparing the model solutions with those of the existing implementations. The results of the statistical tests of this comparison are given in Figures 1–4. Each figure shows the p-value for the ANOVA or Kruskal–Wallis test as well as the $p_{\text{holm-adj}}$-value given by the post hoc tests (Student's t-test or Dunn test, depending on whether ANOVA or Kruskal–Wallis was used) shows which pairs of data have a statistically significant difference. The DynPy implementations gave three negative solutions for the concentration of CO_2 gas. These negative CO_2 concentrations were significant with respect to their positive counterparts, and they had an average value of -9.2×10^{-4} KgCOD m^{-3} which is 6% of the average of the positive concentrations, 1.4×10^{-2} KgCOD m^{-3}. The negative values therefore cannot be considered small enough to be floating point approximations of zero, but they were instead indicative of some failure in the code. These data were excluded from our analysis.

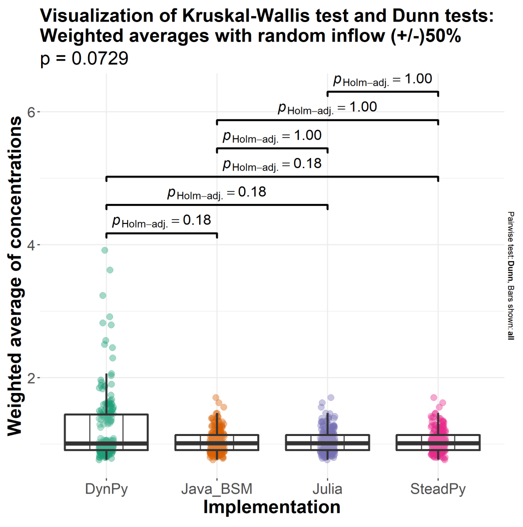
The number of such omitted data for each trial is given in Table 2, along with the mean concentrations of CO_2 gas for both the excluded cases and positive cases, and a comparison of both cases.

Looking at Figures 1–4, we can conclude that the Julia, Java, and `SteadPy` implementations all return results that show no statistically significant difference. This can be seen by looking at the $p_{\text{holm-adj}}$ value between each of them. However, the p-values for the ANOVA and Kruskal–Wallis tests are not always greater than 0.05. Looking at the $p_{\text{holm-adj}}$ values, this is due to the results given by the `DynPy` implementation, which have statistically significant differences with the other implementations. These differences occur for the concentration of CO_2 gas and concentration of CH_4 gas when the inflow is varied within 50% of BSM2 values (Figure 1), for the concentrations of CO_2 gas and CH_4 gas when the inflow is varied within 15% of BSM2 values (Figure 2b,c), and are not seen when the parameters are varied (Figures 3 and 4).

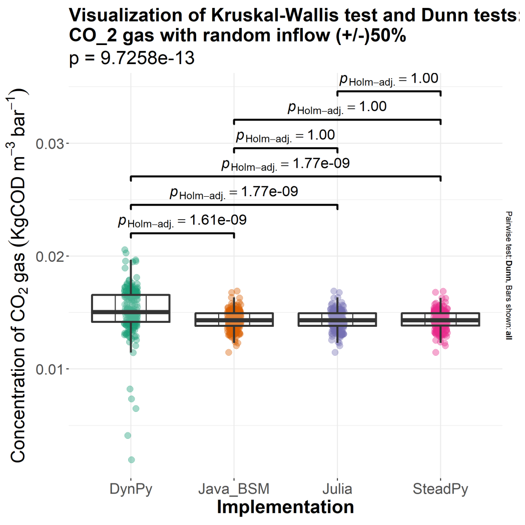
In both cases where the parameters are varied, all of the implementations give some solutions where the weighted averages are very high (>1000), as seen in Figures 3a and 4a, and some solutions where the methane concentrations are near zero, as seen in Figures 3c and 4c. A discussion of these phenomena can be found in the Supplementary Material.

Table 2. Number of DynPy solutions that were omitted for each data set. The average values of CO_2 gas ($S_{\text{gas}_{co2}}$) at time $t = 200$ for both the negative and positive concentrations and the ratio between them are also shown, indicating the negatives were significant and not approximations of zero.

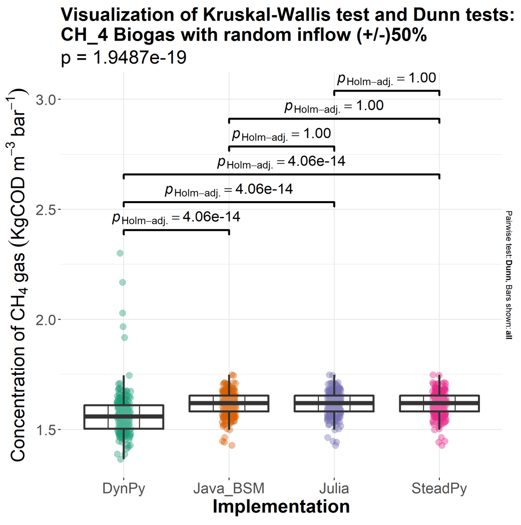
Dataset	**# Omitted**	**Mean $S_{\text{gas}_{\text{co2}}\text{neg}}$**	**Mean $S_{\text{gas}_{\text{co2}}\text{pos}}$**	**Ratio $\frac{\lvert\text{Mean Neg.}\rvert}{\text{Mean Pos.}}$**
inflow $\pm$ 50	3	−0.00092	0.014	0.063
inflow $\pm$ 15	none	N/A	N/A	N/A
rand params	7	−0.0076	0.015	0.50
rand params const. P	5	−0.010	0.015	0.68



(**a**)

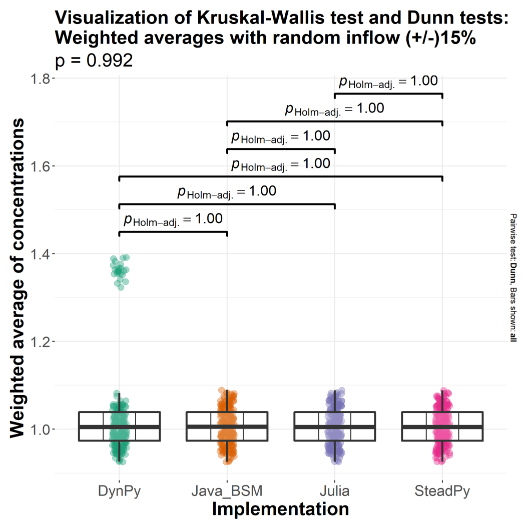


(**b**)

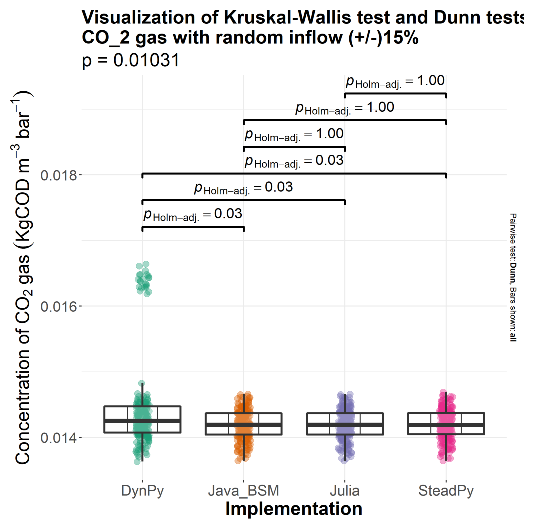


(**c**)

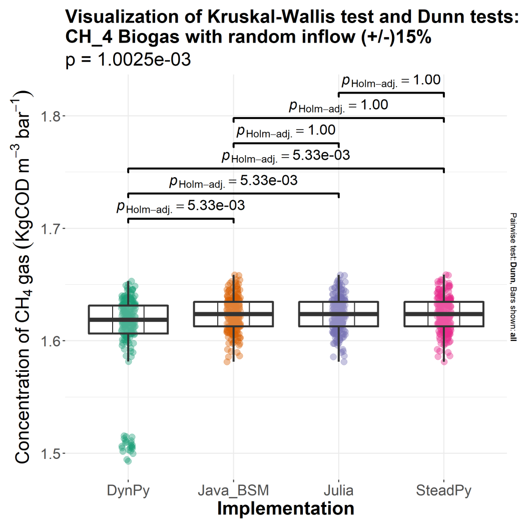
Figure 1. Statistical tests of data when inflow randomised within (+/−)%50 of BSM2 values. If the Levene test returned a *p*-value less than 0.05, a Kruskal–Wallis test was performed; otherwise, ANOVA was performed. The values of the Levene tests were: (**a**) 8.34×10^{-19}, (**b**) 1.37×10^{-21}, (**c**) 3.64×10^{-8}.



(**a**)

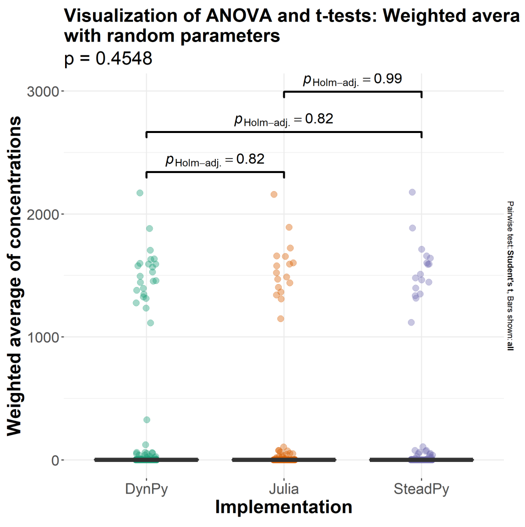


(**b**)

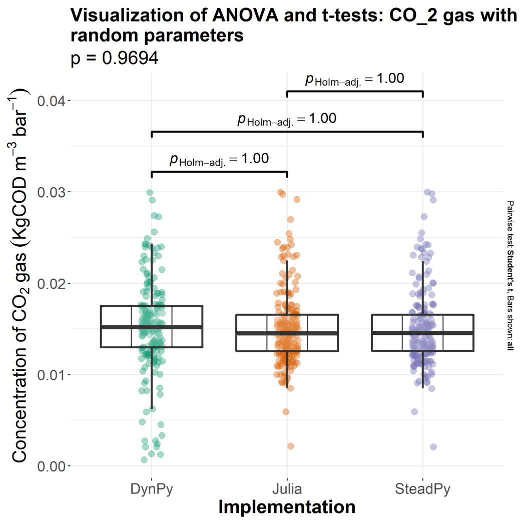


(**c**)

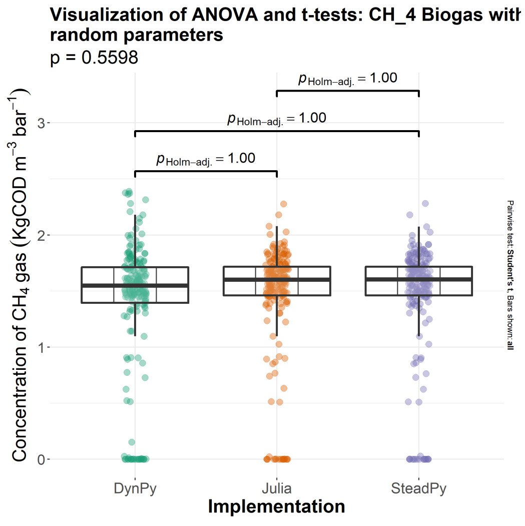
Figure 2. Statistical tests of data when inflow randomised within (+/−)%15 of BSM2 values. If the Levene test returned a *p*-value less than 0.05, a Kruskal–Wallis test was performed; otherwise, ANOVA was performed. The values of the Levene tests were: (**a**) 1.37×10^{-21}, (**b**) 7.55×10^{-14}, (**c**) 4.34×10^{-11}.



(**a**)

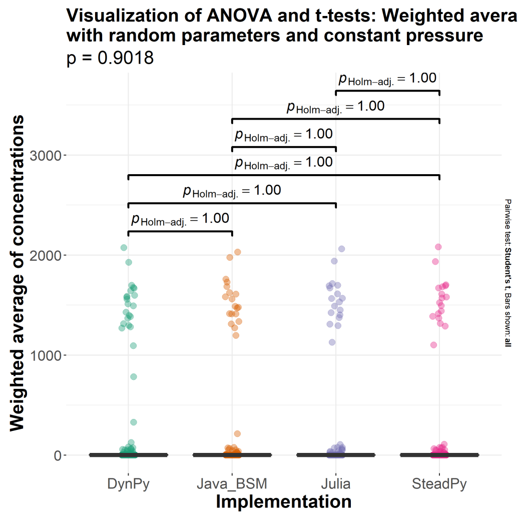


(**b**)

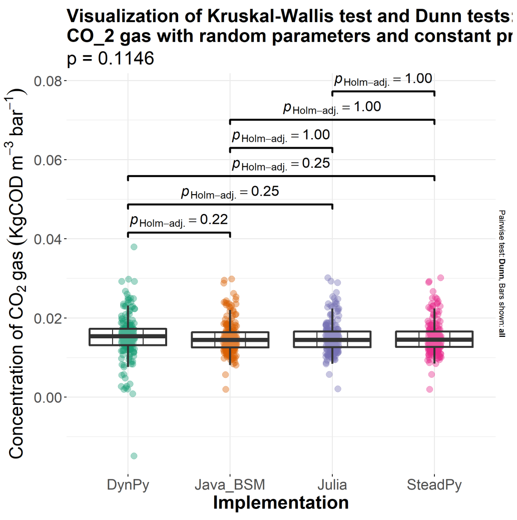


(**c**)

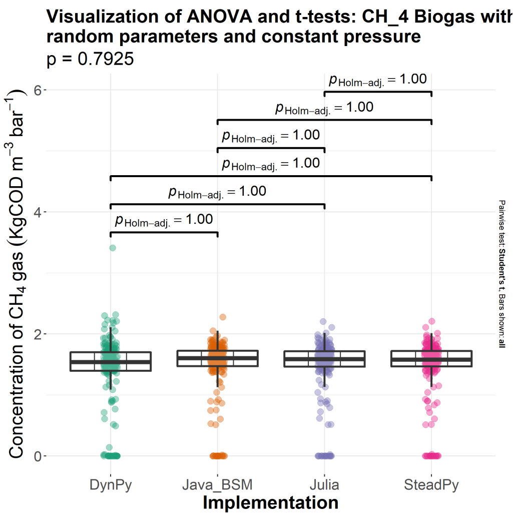
Figure 3. Statistical tests of data when parameter values randomised around BSM2 values. If the Levene test returned a *p*-value less than 0.05, a Kruskal–Wallis test was performed; otherwise, ANOVA was performed. The values of the Levene tests were: (**a**) 0.455, (**b**) 0.0546, (**c**) 0.178.



(**a**)



(**b**)



(**c**)

Figure 4. Statistical tests of data when parameter values randomised around BSM2 values and pressure is kept constant. If the Levene test returned a *p*-value less than 0.05, a Kruskal–Wallis test was performed; otherwise, ANOVA was performed. The values of the Levene tests were: (**a**) 0.902, (**b**) 0.0466, (**c**) 0.394.

3.2. Computational Time

Tables 3–5 show the average computation times and their standard deviations for the simulation runs reported in the previous section. From these tables, it is clear that Julia outperforms the other implementations, with computational times that are around 1/10th the speed of the next fastest implementation, Java. The Python implementations are the slowest, with `DynPy` taking around 20 times longer than the Java code and `SteadPy` taking around three times as long as `DynPy`. It is also worth noting that changing the parameter values increases the computation time by a factor of about two for all of the implementations, with the exception of `SteadPy`, which sees a time decrease of around 15% when the parameter values are varied.

Table 3. Average computation times for the Julia and Python implementations when the inflow vector is varied within 50% of BSM2 values.

Implementation	Average Time (s)	STDEV
Julia	0.22	0.15
`SteadPy`	194	2
`DynPy`	59	4
Java	3.51	0.01

Table 4. Average computation times for the Julia and Python implementations when the inflow vector is varied within 15% of BSM2 values.

Implementation	Average Time (s)	STDEV
Julia	0.24	0.12
`SteadPy`	194	2
`DynPy`	56.5	0.9
Java	3.51	0.02

Table 5. Average computation times for the Julia and Python implementations when the parameter values are varied around the BSM2 values.

Implementation	Average Time (s)	STDEV
Julia	0.54	0.50
`SteadPy`	166	22
`DynPy`	96	11
Java	4.4	1.4

4. Discussion

4.1. Reproducibility and Validation

Validating an implementation of a mathematical model is one of the biggest challenges of coding computer simulations—in some cases because no data are available, and in others because data are limited, and translating known physical parameters to the model parameters is difficult. Implementations of ADM1 suffer from both issues. ADM1 has 104 model parameters, including reaction rates and yield coefficients, the exact values of which can only be estimated. This is why the BSM2 only examines three different physical cases. Independent implementations offer another way of evaluating the code. Every additional implementation that produces the same results serves to validate the previous implementations. Additionally, if any discrepancies are detected between implementations, finding their source can reveal more information about the strengths and weaknesses of the implementations. For example, the greater number of discrepancies between the `DynPy` implementation and the others can be explained by the time step size between recomputations of the algebraic equations. While a constant step size of $dt = 15$ min appears sufficiently accurate when compared to the variable step size method used by the

Julia implementation, the gradual increase in step size from $dt \approx 5$ s to $dt = 5$ days used in `DynPy` can result in a loss of accuracy.

4.2. Compute Time: Why It Matters

Naïvely, it may seem like the speed increase from the Java implementation to the Julia implementation is not a significant result, since a compute time of 3 s is already pretty quick. However, for many computational problems, the 10-fold speed increase makes a significant difference. For example, for a model of the size of ADM1, optimal control problems, automatic parameter calibrations, and sensitivity analyses can easily require the generation of thousands of model simulations to determine the effect that the different state variables and parameters have on the system. The amount of computations needed for these problems increases with the number of state variables and parameters. So, to perform a sensitivity analysis on ADM1, each of the 104 model parameters needs to be varied several times to determine the model's sensitivity to that parameter. If each parameter is varied only ten times, the total computation time is approximately an hour for the Java code and only around four minutes for the Julia code. The computation time difference only grows as more and more data are required, so it is best to reduce the computation time as much as possible in order to perform more detailed analyses of the model.

Additionally, if a model has a fast computation time, the model can be connected to a physical reactor to make real-time predictions based on data as it is measured, for example in digital twin applications. These predictions allow engineers to make informed operational decisions. However, the more parameters the model has, the more computationally expensive it is to calibrate the model to match the physical reactor. So, increasing the speed of the compute time increases the feasibility of linking the model to the physical reactor. In this regard, we argue that our de novo implementation offers a significant improvement on existing implementations, which might allow a model-supported design and operation of anaerobic digestion processes that are currently not in reach.

4.3. DAE vs. ODE Formulations of ADM1

The DAE form was developed to increase the computational speed of ADM1 [7]; however, the results of this study show that the DAE form is not necessarily faster than the ODE form. In fact, it seems that which numerical methods are used makes a bigger difference on computational time than the form of the DAE. When designing the Julia implementation, four different solver methods were tested, `Rosenbrock23()`, `Rodas5()`, `Rodas4P()` and `radau()`. Each solver method was tested for absolute and relative tolerancess of 10^{-1}, 10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}, and 10^{-10}. These tests showed that the choice of both the solver method and the tolerances had a significant impact on the computation time, with times varying from 0.6 s to over five minutes for a single solver method depending on the tolerances used. Additionally, some solver methods were more reliably fast, which is why the `Rodas4P()` method with absolute and relative tolerances of 10^{-4} was ultimately chosen. Although this appears to be a rather large tolerance, the simulation results obtained agree very well with those given by other, slower implementations. This indicates that the commonly held beliefs about the computation speed of the DAE form may be steering people away from examining the root causes of computational slowdown.

4.4. Use of the Petersen Matrix Formulation of ADM1

Other than the choice of DAE form vs. ODE form, the most significant difference between the Julia implementation and the other implementations examined in this paper is the choice to implement ADM1 directly in the matrix form given by Equation (1) as opposed to typing out all of the differential equations. Although ADM1 itself was developed as a generic and widely applicable model of anaerobic digestion processes, many applications might require adaptation to the specifics of the particular problem being studied. This can take the form of modifying process rates, refining process descriptions or including processes that are not included in the generic ADM1. The new Julia implementation can be adapted by providing an updated Petersen matrix and a corresponding vector of reaction rates; the user is not required to alter the affected and modified differential equations individually. This design choice improves the usability of the code compared to the reference implementations that went the more traditional route of implementing each differential equation directly. However, implementing the matrix form might seem like it would negatively impact the computation time, since the Petersen matrix is sparse and therefore results in a lot of multiplications by zero. One possible work around could be declaring the Petersen matrix as `Sparse` and using sparse linear algebra techniques. However, when comparing both versions of the Julia code, keeping the Petersen matrix as a full array actually results in a faster computation time. If we again look back to our results on computation time, the Julia implementation is actually significantly faster than the others despite performing all of the multiplications by zero. So, implementing the matrix form offers flexibility without any noticeable impact on computation time, contrary to what might be expected.

5. Conclusions

It may seem like implementing new software to solve a problem that has existing software solutions available is merely retreading established work or at most serves as an independent validation of existing results. However, not only do new implementations validate existing implementations, they also offer deeper insight into the more complex issues of how to implement a model, and they can serve to challenge naïve assumptions. Without these observations, innovation and improvement is not possible. We have shown that, contrary to popular belief, the ODE form of ADM1 is not necessarily slower than the DAE form, and that the numerical methods used play a much bigger role. We have also shown that implementing the matrix form of ADM1 does not have a noticeable effect on compute time. Both of these combined allow for a more efficient and flexible implementation that would not exist if the conventional wisdom regarding computational speed was not tested.

Our conclusions about the performance of the new implementation of ADM1 relative to earlier implementations are based on a large number of simulations and on the strong statistical significance of our results. Therefore, we think they will carry over generally to situations where ADM1 is an appropriate model.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/pr11071899/s1.

Author Contributions: Conceptualisation, C.A., N.A. and H.J.E.; methodology, C.A and H.J.E.; software, C.A. and A.M.; validation, C.A. and A.M.; formal analysis, C.A. and A.M.; investigation, C.A., N.A. and H.J.E.; resources, H.J.E. and N.A.; data curation, C.A.; writing—original draft preparation, C.A. and A.M.; writing—review and editing, C.A., N.A. and H.J.E.; visualisation, C.A.; supervision, H.J.E.; project administration, H.J.E. and N.A.; funding acquisition, H.J.E. and N.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Natural Sciences and Engineering and Research Council of Canada (NSERC) [grant number ALLRP 560564] and by the Ontario Centre for Innovation (OCI) [grant number #34231].

Data Availability Statement: The supplementary material includes a link to the database containing the Julia code used to generate the data.

Acknowledgments: The authors thank Mohammad Zaker for assistance with the java codes.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AD	Anaerobic Digestion
ADM1	Anaerobic Digestion Model Number 1
ANOVA	Analysis of Variances
BSM2	Benchmark Simulation Model 2
DAE	Differential–Algebraic Equation
ODE	Ordinary Differential Equation
STDEV	Standard Deviation

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. [CrossRef] [PubMed]
2. Uddin, M.N.; Siddiki, S.Y.A.; Mofijur, M.; Djavanroodi, F.; Hazrat, M.A.; Show, P.L.; Ahmed, S.F.; CHu, Y.M. Prospects of Bioenergy Production From Organic Waste Using Anaerobic Digestion Technology: A Mini Review. *Front. Energy Res.* **2021**, *9*, 627093 . [CrossRef]
3. Uddin, M.M.; Wright, M.M. Anaerobic Digestion Fundamentals, Challenges, and Technological Advances. *Phys. Sci. Rev.* **2022**. [CrossRef]
4. Rittmann, B.E.; McCarty, P.L. *Environmental Biotechnology: Principles and Applications*; McGraw-Hill: New York, NY, USA, 2001.
5. Meegoda, J.N.; Li, B.; Patel, K.; Wang, L.B. A Review of the Processes, Parameters, and Optimization of Anaerobic Digestion. *Int. J. Environ. Res. Public Health* **2018**, *15*, 2224. [CrossRef] [PubMed]
6. Batstone, D.; Keller, J.; Angelidaki, I.; Kalyuzhnyi, S.; Pavlostathis, S.; Rozzi, A.; Sanders, W.; Siegrist, H.; Vavilin, V. *Anaerobic Digestion Model No. 1 (ADM1)*; Scientific and Technical Report, no. 13; IWA Publisher: London, UK, 2002.
7. Alex, J.; Benedetti, L.; Copp, J.; Gernaey, K.; Jeppsson, U.; Nopens, I.; Pons, M.; Rosen, C.; Steyer, J.; Vanrolleghem, P. *Benchmark Simulation Model No. 2 (BSM2)*; International Water Association: London, UK, 2019.
8. Gavaghan, D. Problems with the Current Approach to the Dissemination of Computational Science Research and Its Implications for Research Integrity. *Bull. Math. Biol.* **2018**, *80*, 3088–3094. [CrossRef] [PubMed]
9. Schnell, S. "Reproducible" Research in Mathematical Sciences Requires Changes in our Peer Review Culture and Modernization of our Current Publication Approach. *Bull. Math. Biol.* **2018**, *80*, 3095–3105. [CrossRef] [PubMed]
10. Stagge, J.H.; Rosenberg, D.E.; Abdallah, A.M.; Akbar, H.; Attallah, N.A.; James, R. Assessing Data Availability and Research Reproducibility in Hydrology and Water Resources. *Sci. Data* **2019**, *6*, 190030. [CrossRef] [PubMed]
11. Rosén, C.; Jeppsson, U. *Aspects on ADM1 Implementation within the BSM2 Framework*; Department of Industrial Electrical Engineering and Automation, Lund University: Lund, Sweden, 2006; pp. 1–35.
12. Sadrimajd, P.; Mannion, P.; Howley, E.; Lens, P.N.L. PyADM1: A Python Implementation of Anaerobic Digestion Model No. 1. *bioRxiv* **2021**. [CrossRef]
13. Hairer, E.; Nørsett, S.; Wanner, G. Explicit Runge-Kutta Methods of Higher Order. In *Solving Ordinary Differential Equations I: Nonstiff Problems*; Springer Series in Computational Mathematics; Springer: Berlin/Heidelberg, Germany, 2008; Chapter II, pp. 181–184.
14. Pettigrew, L.; Hubert, S.; Groß, F.; Delgado, A. Implementation of Dynamic Biological Process Models into a Reference Net Simulation Environment. In Proceedings of the ASIM Dedicated Conference Simulation in Production and Logistics, Dortmund, Germany, 24 September 2015.
15. Pettigrew, L.; Gutbrod, A.; Domes, H.; Groß, F.; Méndez-Contreras, J.M.; Delgado, A. Modified ADM1 for high-rate anaerobic co-digestion of thermally pre-treated brewery surplus yeast wastewater. *Water Sci. Technol.* **2017**, *76*, 542–554. [CrossRef]
16. Hairer, E.; Nørsett, S.; Wanner, G. Chapter III.1 Classical Linear Multistep Formulas. In *Solving Ordinary Differential Equations I: Nonstiff Problems*; Springer Series in Computational Mathematics; Springer: Berlin/Heidelberg, Germany, 1993; pp. 356–361.
17. Rackauckas, C. A Comparison Between Differential Equation Solver Suites in MATLAB, R, Julia, Python, C, Mathematica, Maple, and Fortran. *Winnower* **2018**. [CrossRef]
18. Rackauckas, C.; Nie, Q. Differentialequations.jl–a Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *J. Open Res. Softw.* **2017**, *5*. [CrossRef]
19. Fox, J. *Applied Regression Analysis and Generalized Linear Models*, 3rd ed.; Sage: Newcastle upon Tyne, UK, 2008.
20. Kruskal, W.H.; Wallis, W.A. Use of Ranks in One-Criterion Variance Analysis. *J. Am. Stat. Assoc.* **1952**, *47*, 583–621. [CrossRef]
21. Kalpić, D.; Hlupić, M.L.N. *International Encyclopedia of Statistical Science*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 1559–1563.
22. Dunn, O.J. Multiple Comparisons Using Rank Sums. *Technometrics* **1964**, *6*, 241–252. [CrossRef]
23. Microsoft Support: STDEV.S Function. 2023. Available online: https://support.microsoft.com/en-us/office/stdev-s-function-7d69cf97-0c1f-4acf-be27-f3e83904cc23 (accessed on1 May 2023).
24. Müller, K.; Wickham, H. tibble: Simple Data Frames. 2022. R Package Version 3.1.7. Available online: https://CRAN.R-project.org/package=tibble (accessed on1 May 2023).

25. Fox, J.; Weisberg, S. *An R Companion to Applied Regression*, 3rd ed.; Sage: Thousand Oaks, CA, USA, 2019.
26. Patil, I. Visualizations with Statistical Details: The ‘ggstatsplot’ approach. *J. Open Source Softw.* **2021**, *6*, 3167. [CrossRef]
27. Wickham, H. *ggplot2: Elegant Graphics for Data Analysis*; Springer: New York, NY, USA, 2016.
28. R Core Team. *R: A Language and Environment for Statistical Computing*; R Foundation for Statistical Computing: Vienna, Austria, 2022. Available online: https://www.R-project.org/ (accessed on 1 May 2023).
29. Pohlert, T. PMCMRplus: Calculate Pairwise Multiple Comparisons of Mean Rank Sums Extended. 2022. R Package Version 1.9.5. Available online: https://CRAN.R-project.org/package=PMCMRplus (accessed on 1 May 2023).