Implementing Ordinary Differential Equation Solvers in Rust Programming Language for Modeling Vehicle Powertrain Systems

Preprint

Robin Steuteville and Chad Baker

National Renewable Energy Laboratory

Presented at the Society of Automotive Engineers 2024 World Congress Experience
Detroit, Michigan
April 16-18, 2024

NREL is a national laboratory of the U.S. Department of Energy
Office of Energy Efficiency & Renewable Energy
Operated by the Alliance for Sustainable Energy, LLC

This report is available at no cost from the National Renewable Energy Laboratory (NREL) at www.nrel.gov/publications.

Contract No. DE-AC36-08GO28308

Conference Paper
NREL/CP-5400-91707
October 2024

Implementing Ordinary Differential Equation Solvers in Rust Programming Language for Modeling Vehicle Powertrain Systems

Preprint

Robin Steuteville and Chad Baker

National Renewable Energy Laboratory

Suggested Citation
Steuteville, Robin and Chad Baker. 2024. *Implementing Ordinary Differential Equation Solvers in Rust Programming Language for Modeling Vehicle Powertrain Systems: Preprint*. Golden, CO: National Renewable Energy Laboratory. NREL/CP-5400-91707. https://www.nrel.gov/docs/fy25osti/91707.pdf.

Conference Paper
NREL/CP-5400-91707
October 2024

National Renewable Energy Laboratory
15013 Denver West Parkway
Golden, CO 80401
303-275-3000 • www.nrel.gov

NOTICE

This work was authored by the National Renewable Energy Laboratory, operated by Alliance for Sustainable Energy, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. Funding provided by U.S. Department of Energy Office of Energy Efficiency and Renewable Energy Vehicle Technologies Office under the Energy Efficient Mobility Systems program. The views expressed herein do not necessarily represent the views of the DOE or the U.S. Government. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this work, or allow others to do so, for U.S. Government purposes.

This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Science Undergraduate Laboratory Internships Program (SULI).

This report is available at no cost from the National Renewable Energy Laboratory (NREL) at www.nrel.gov/publications.

U.S. Department of Energy (DOE) reports produced after 1991 and a growing number of pre-1991 documents are available free via www.OSTI.gov.

Cover Photos by Dennis Schroeder: (clockwise, left to right) NREL 51934, NREL 45897, NREL 42160, NREL 45891, NREL 48097, NREL 46526.

NREL prints on paper that contains recycled content.

Implementing Ordinary Differential Equation Solvers in Rust Programming Language for Modeling Vehicle Powertrain Systems

Abstract

Efficient and accurate ordinary differential equation (ODE) solvers are necessary for powertrain and vehicle dynamics modeling. However, current commercial ODE solvers can be financially prohibitive, leading to a need for accessible, effective, open-source ODE solvers designed for powertrain modeling. Rust is a compiled programming language that has the potential to be used for fast and easy-to-use powertrain models, given its exceptional computational performance, robust package ecosystem, and short time required for modelers to become proficient. However, of the three commonly used (>3,000 downloads) packages in Rust with ODE solver capabilities, only one has more than four numerical methods implemented, and none are designed specifically for modeling physical systems. Therefore, the goal of the Differential Equation System Solver (DESS) was to implement accurate ODE solvers in Rust designed for the component-based problems often seen in powertrain modeling. DESS is a text-based software package that provides a flexible framework for building and solving systems of ODEs. This allows DESS to be included as a dependency for automotive powertrain models that require a variety of solvers and solver configurations. Seven explicit ODE solver methods have been implemented in DESS: Euler's, Heun's, midpoint, Ralston's, classic Runge-Kutta, Bogacki-Shampine, and Cash-Karp. These represent five fixed-step methods and two adaptive-step methods. This paper shows that the solver implementations increase accuracy and computational efficiency compared to Euler's method when modeling a system of three thermal masses in Rust. DESS also includes features designed for modeling component-based physical systems. Users can define relationships between nodes in their system, which the package then translates into a system of equations, leading to simpler and more intuitive code. In the case of a three-thermal-mass system, the user can specify node thermal properties (e.g., thermal capacitance), how nodes are interconnected, and thermal conductance between nodes rather than providing a system of equations. The core contribution from this work is an open-source, text-based Rust package with ODE solvers for automotive powertrain modeling to support cost-free, fast, and accurate simulation.

Introduction

Effective powertrain models are essential tools for increasing efficiency and transitioning to lower carbon vehicle designs. Ordinary differential equations (ODEs) are important for powertrain modeling. However, these equations are often difficult or impossible to solve exactly, meaning the solutions must be estimated. There are numerous ODE solvers that can be implemented for this purpose with various levels of accuracy, computational efficiency, and complexity, several of which will be discussed later in this paper. However, the current high-level ODE solver implementations on the market can be prohibitively expensive. MATLAB, Mathematica, and Maple are examples of commercial software with ODE solver capabilities, and annual individual commercial licenses for each of these programs cost ~$1,000–$2,000 [1-3]. Rust is an open-source programming language with the potential to be a powerful and useful modeling tool, but it currently lacks versatility and options for solving ODEs [4-10]. Therefore, the aim of this research was to create an open-source, text-based software in Rust implementing several ODE solvers, called the Differential Equation System Solver (DESS), tailored to the specific types of problems most often seen when modeling powertrain systems.

The efficiency and ease-of-coding that Rust offers makes it the ideal language for an open-source ODE solver software intended to be used for powertrain modeling. Rust is faster than Python [4] and comparable in speed to the compiled language C++ [5], and Rust code is simpler and more concise than other comparable languages [5, 6]. However, there are few ODE solver options implemented in Rust. Current Rust packages with ODE solver capabilities tend to have few solvers implemented and lack some capabilities that are helpful for powertrain modeling. The most popular packages in Rust (those with more than 3,000 downloads) with ODE solver capabilities are "ode_solvers," "mathru," and "peroxide" [7]. Of those, "ode_solvers" has three different solvers [8], "peroxide" has four [9], and "mathru" has 15 different solvers to choose from [10]. Given the limited options for solving ODEs in Rust, a primary goal of DESS was to provide additional capabilities within Rust to solve systems of ODEs. So far, DESS implements seven solver types. For powertrain problems, one important type of ODE solver is the adaptive-step solver, which is not implemented at all in the "peroxide" package [9]. DESS implements two adaptive solvers with different accuracy and complexity levels. Furthermore, "ode_solvers," "mathru," and "peroxide" all require users to write their own systems of equations [8-10]. While this may make sense for more generalized solvers, powertrain-specific problems benefit from conceptualizing the system in terms of the relationships between different components. Therefore, in addition to allowing users to directly input equations, DESS allows users to optionally specify and define relationships between nodes in their system, which the package then translates into a system of equations via the Rust macro system, leading to simpler and more intuitive code. DESS users can specify DESS as a dependency and follow provided examples of how to structure the system being solved to make use of DESS's solvers with the option of building a Python API, also provided in examples within the package. The resulting program is a user-friendly, accessible tool that can be applied to current and future models to improve efficiency and accuracy.

This work first describes the specific type of ODE problem to be solved as well as the methods implemented. It then discusses the implementation in Rust and how the accuracy of different solvers was measured. Finally, the paper discusses how the solver implementations affected accuracy and efficiency in two test cases, the limitations of DESS thus far, and capabilities and solvers that will be implemented in the future.

Methods

Initial Value Problem

The purpose of DESS is to solve a specific type of first-order ODE, called an initial value problem (IVP) [11]. An IVP includes a system of ODEs and a set of initial values, which identify a specific solution to the ODE system [11]. Although DESS cannot directly solve higher-order ODEs, it is often possible to write higher-order ODEs as a system of first-order ODEs in state-space representation [11], allowing them to be solvable by DESS.

ODE Solver Methods

Need for Various Solvers

DESS needed to easily handle the types of ODEs used in powertrain modeling and to effectively manage the trade-off between accuracy and computational efficiency. Therefore, different solvers were implemented in DESS for different scenarios. DESS needed basic methods to give rough estimates with very little computational cost. These were also easy to implement, so they could be used to troubleshoot more complex solvers. DESS also needed advanced solvers that could handle ODEs with high accuracy and high efficiency. With these goals in mind, two main solver types were implemented: explicit fixed-step solver methods and explicit adaptive-step solver methods. Explicit solver methods only need one initial value, whereas implicit methods require multiple initial values to solve a single ODE [11]. Given that the types of problems DESS aimed to solve only supplied one initial value per ODE, explicit solver methods were implemented in DESS.

Explicit Fixed-Step Solver Methods

Explicit fixed-step solver methods are the most basic type of ODE solver and serve as a useful demonstration for how ODE solvers work. Generally, one-step ODE solvers attempt to approximate the average derivative of the solution, $y(t)$, to the ODE over an interval of time h [11]. Given this approximation (which we call k), the solver then adds $k * h$ to the initial value $y(t_0) = x_0$ to get the approximate value at $t_0 + h$ [11]. For fixed-step solvers, the time step, h, is fixed. A depiction of a single step of an ODE solver can be seen in Figure 1.

The first fixed-step solver implemented in DESS was Euler's method [12], which is the most basic fixed-step method. This method approximates the average derivative for $y(t)$ over the time step h by taking the initial value $y(t_0) = x_0$ and initial time t_0 and calculating $y'(t_0) = f(t_0, y(t_0)) = f(t_0, x_0)$. The estimation for $y(t_0 + h)$ is then calculated by taking $x_0 + f(t_0, x_0) * h$.

Note that this method has the potential to be very inaccurate for large time steps but can be very accurate for many functions if a small enough time step is chosen (with the downside of added computational cost). As a result, when testing the accuracy of solvers and troubleshooting, an Euler's method approximation with a very small time step was used as a baseline. In general, the accuracy of an ODE solver increases with smaller time steps [11]. See Figure 2 for a comparison of the accuracy of Euler's method for three different time step sizes.

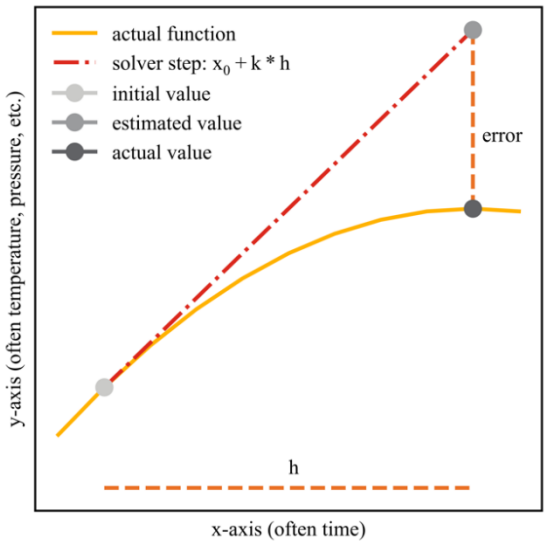

Figure 1. Depiction of a single step of an ODE solver.

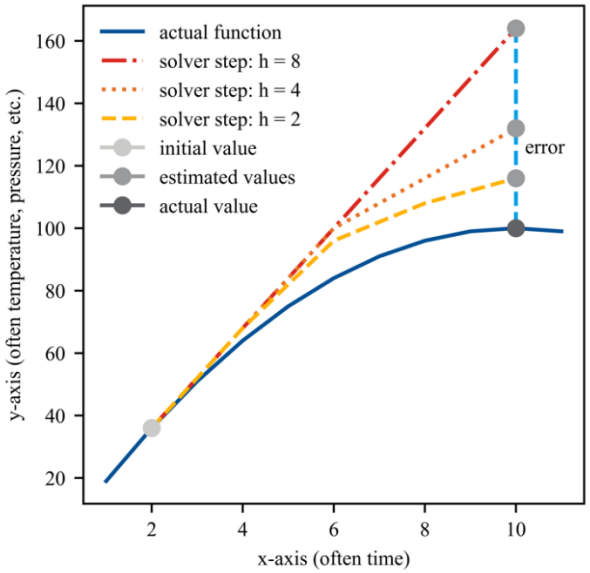

Figure 2. Accuracy of Euler's method for different step sizes, for the parabola $y = -(x - 10)^2 + 100$. Note that the accuracy increases as the time steps decrease in size.

The fixed-step methods implemented were all Runge-Kutta (RK) methods (with Euler's method being the most basic RK method). Broadly, RK methods use derivative estimations at various points along the time step h to create an overall estimation for the average derivative that is more accurate than any one of the original estimations [11].Each additional estimation is obtained recursively through previous ones [11]. The general formula for any RK method is defined in equation (1). RK methods then calculate $y(t_0) + kh$ to be the estimation of $y(t_0 + h)$ [11].

$$k_1 = f(t_0, y(t_0))$$

$$k_2 = f(t_0 + p_1 h, y(t_0) + q_{11} k_1 h)$$

$$k_3 = f(t_0 + p_2 h, y(t_0) + q_{21} k_1 h + q_{22} k_2 h)$$

$$\vdots$$

$$k_n = f(t_0 + p_{n-1}h, y(t_0) + q_{n-1,1}k_1h + q_{n-1,2}k_2h + \cdots + q_{n-1,n-1}k_{n-1}h)$$

$$k = a_1k_1 + a_2k_2 + \cdots + a_nk_n \quad (1)$$

Where,
$k_1, \ldots, k_n$ = intermediate estimated average derivatives
k = final estimated average derivative
t_0 = initial time
h = time step
$y(t_0)$ = initial value at t_0
$a_1, \ldots, a_n, p_1, \ldots, p_{n-1}, q_{11}, \ldots, q_{n-1,n-1}$ = constants that depend on the specific Runge-Kutta method [11].

DESS includes five fixed-step RK methods in total: Euler's method [12], Heun's method [13], midpoint method [14], Ralston's method [15], and the classic Runge-Kutta method (RK4) [16]. For methods of greater order, a greater number of intermediate calculation stages is required [11]. Therefore, for the same constant time step across all methods, Euler's method is the least accurate as a first-order method with one stage; Heun's method, midpoint method, and Ralston's method are second-order methods with two stages and relatively higher accuracy; and the RK4 method, a fourth-order method with four stages, is the most accurate of the fixed-step solvers implemented in DESS [11]. With increased accuracy comes increased complexity, so for the same time step, Euler's method will be the most computationally efficient.

Explicit Adaptive-Step Solver Methods

Fixed-step methods can be effective for simple problems, for use as benchmarks, and for troubleshooting. However, powertrain models often need to handle fast transients, which might require a small time step to be accurate, and periods of slower change, where a larger time step would be sufficient and would save computational time and energy. For instance, in modeling a vehicle with engine stop/start behavior (e.g., any hybrid powertrain), larger time steps (e.g., 60 s) would be sufficiently accurate during the stops, and smaller time steps would be needed while the vehicle is in motion. It is for these types of cases that adaptive-step solver methods are used.

Adaptive-step solver methods use a varied time step based on a function's behavior at a given point in time, rather than a fixed time step. For this reason, they are often the better choice when approximating functions that have periods of fast change and periods of slow change. Two adaptive-step solvers were implemented in DESS: the Bogacki-Shampine (RK23) method and the Cash-Karp (RK45) method. The RK23 method was first introduced by Dr. Przemyslaw Bogacki and Dr. Lawrence Shampine in their 1989 paper titled "A 3(2) Pair of Runge-Kutta Formulas" [17]. In 1990, Dr. Jeff Cash and Dr. Alan Karp introduced the RK45 method in their paper titled "A Variable Order Runge-Kutta Method for Initial Value Problems with Rapidly Varying Right-Hand Sides" [18]. Both methods use as their basis two Runge-Kutta methods of different order [17, 18]. In the case of the RK23 method, the solver uses one Runge-Kutta method of order two (with four stages), and one of order three (with three stages) [17]. In the case of the RK45 method, one Runge-Kutta method of order four is used, and one of order five is used (each with six stages) [18].

These adaptive solvers work by calculating two different Runge-Kutta solvers (of different order) for each step and comparing the results to get a rough estimate of the error [17, 18]. If the difference between the two solvers is less than a previously specified error tolerance (in relative or absolute error, depending on which is smaller), then the step is successful [17, 18]. The solver then steps forward using the found time step with the higher-order solver estimation [17, 18]. If the difference between the two solvers does not meet the specified error tolerance, then the step is repeated with a smaller time step [17, 18]. See Figure 3 for a representation of a single step of an adaptive-step ODE solver.

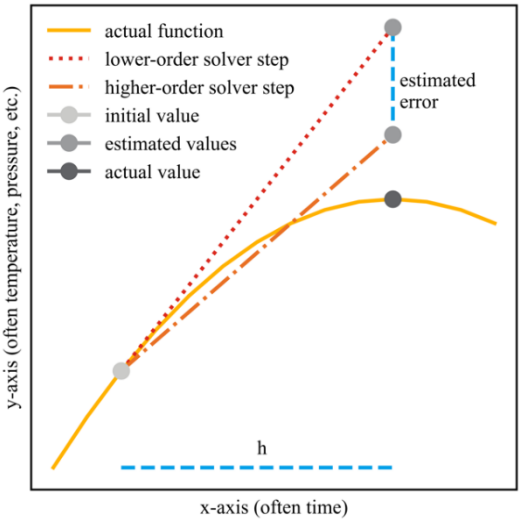

Figure 3. Depiction of a single step of an adaptive-step solver.

The RK23 method and the RK45 method each have ways of keeping the computational cost low throughout this process. See equation (2) for the RK23 algorithm. Note that the second- and third-order methods used in RK23 reuse three of the same stages, meaning that doing both methods requires very little additional computational cost. Furthermore, the calculation for the third-order method is reused to make the second-order method more robust while adding little to no computational cost. In a similar manner, the RK45 method also reuses stages to keep the computational cost low [18]. See the appendix for the full algorithm of the RK45 method.

$$k_1 = f(t_0, y(t_0))$$

$$k_2 = f\left(t_0 + \frac{1}{2}h, y(t_0) + \frac{1}{2}k_1h\right)$$

$$k_3 = f\left(t_0 + \frac{3}{4}h, y(t_0) + \frac{3}{4}k_2h\right)$$

$$O_3 = y(t_0) + \frac{2}{9}k_1h + \frac{1}{3}k_2h + \frac{4}{9}k_3h$$

$$k_4 = f(t_0 + h, O_3)$$

$$O_2 = y(t_0) + \frac{7}{24}k_1h + \frac{1}{4}k_2h + \frac{1}{3}k_3h + \frac{1}{8}k_4h \quad (2)$$

Where,
$k_1, \ldots, k_n$ = intermediate estimated average derivatives
t_0 = initial time
$y(t_0)$ = initial value
h = current time step

O_2 = estimated value for the second-order Runge-Kutta method
O_3 = estimated value for the third-order Runge-Kutta method [17].

Solver Implementation in Rust

As the simplest method, Euler's method was implemented first as a benchmark and to troubleshoot later solvers. For testing purposes, two systems of interacting thermal masses were chosen. Given that the systems were simple and their behavior known, these systems were able to flag any inaccurate or incorrect behavior by the solvers to ensure the solvers were producing accurate and realistic results.

The solver environment in DESS includes various parameters that can be updated by the user, depending on the type of solver used. For the fixed-step solvers, the user inputs the time step. For the adaptive-step solvers, there are several parameters to give the user control over the process, including the initial time step, maximum time step (the upper bound on the time step chosen by the solver), maximum number of iterations before the solver moves on without reaching error tolerance, acceptable relative error, acceptable actual error, and whether to save the solver history. All the implemented solvers work with vectors rather than single numbers and therefore can solve a system of ODEs simultaneously.

The user must also add the derivative of the equation(s) for which a solution is required. This is done through a Rust macro that updates the derivatives of the system when given a set of user-defined values and times. The macro updating the derivative is the only required user-defined step, with all other required macros and functions already defined within the program. Allowing the user to define the derivative macro also gives the user the freedom to define the derivative as a set of interconnected components rather than using an explicit function (although this is also possible). To assist with this step, DESS also includes a macro that can create equations from systems of interconnected nodes. For more information, see the DESS GitHub: https://github.com/NREL/de-system-solver.

Methods of Error Estimation

Relative and absolute error definitions were necessary to test method implementations, quantify accuracy, and enable adaptive solvers within DESS. The absolute error was defined as the difference between the solver output and the baseline when measuring the actual error. When estimating the absolute error within the adaptive-step methods, the estimated absolute error was defined as the difference between the two estimates calculated as part of the adaptive-step method. The relative error definition for measuring actual relative error can be found in equation (3). To minimize the cases where relative error could not be calculated due to the denominator being zero, DESS used a definition of relative error that normalized the error using the average of the actual (baseline) and estimated (solver produced) values. For estimating relative error within the adaptive-step solvers, the higher order estimate was used in place of the average.

$$\text{average} = \frac{\text{actual} + \text{estimated}}{2}$$

$$\text{relative error} = \frac{\text{actual} - \text{estimated}}{\text{average}} \tag{3}$$

If relative error is undefined because both actual and estimated values are exactly zero (or if the two values are exact opposites and therefore cancel each other out), then absolute error is used. If both relative and absolute errors exist, the smaller of the two values is used as the error. In the case of the adaptive-step solvers, this means that if either the relative or absolute error threshold is met, the step will succeed, and the solver will move on to the next step. Typically, relative errors are preferable because normalized relative errors can be compared across cases with different magnitudes of model variables. However, when dealing with very small actual and estimated values, such as when the function being modeled goes to zero, relative error can sometimes blow up (because dividing by a very small denominator can produce very large values), resulting in error measurements out of proportion to the actual error present. Therefore, choosing the smaller of the relative and absolute errors ensures that if the relative error is producing unrealistically large outputs, it does not affect the measured error.

DESS also includes two tests to compare the accuracy of the different methods. Both tests compare each solver method to a baseline to judge the accuracy of each solver. The first test compares the values in the baseline with the values in the given method. It returns “true” if all the errors were within the relative/absolute error tolerance and “false” otherwise. The second test returns the average absolute error between the baseline and chosen method. This test is useful to compare the exact performance of different solvers for a specific system of ODEs. DESS also includes tests specifically for the adaptive solvers. These include a test that outputs the average time step of an adaptive method when solving a specific system, as well as a test measuring how many iterations the adaptive solver needs for each time step. Finally, DESS also includes the option to build a Python API, which uses the Rust solvers but adds additional capabilities such as the ability to plot solver outputs and time the solver calculations to compare computational efficiency between solvers.

Results and Discussion

Accuracy of Implemented Solvers

To test the accuracy and efficiency of the solver implementations in DESS, two systems of three interacting (via heat transfer coefficients) thermal masses were implemented in DESS. A baseline for each system was created using Euler's method with a step size of 5^{-10}s for a high level of accuracy, and the systems were modeled over a 1 s time period. See Figure 4 for a visualization of the systems used to test DESS.

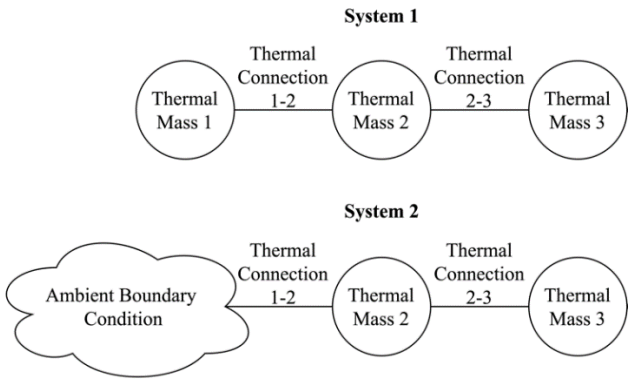

Figure 4. Depiction of both thermal mass systems used to test DESS.

The accuracy of each solver was compared for each system by calculating the average absolute error for each method for three different time steps. Parameters were chosen for the adaptive-step

solvers such that they produced an average time step equal to the fixed time steps. Since each thermal mass system modeled the heat of three different interacting nodes, the result was three different models, one for each node. Therefore, overall average accuracy was obtained by averaging the accuracy for each of the three models. The results are found in Tables 1 and 2.

Table 1. Average accuracy of each method for three different time steps for the first three-thermal-mass system. For adaptive-step solvers, the average time step was used.

h	0.1 s	0.05 s	0.01 s
Euler's	1.19×10^{-1} °C	6.17×10^{-2} °C	1.26×10^{-2} °C
Heun's	3.77×10^{-2} °C	6.36×10^{-3} °C	2.16×10^{-4} °C
Midpoint	3.77×10^{-2} °C	6.36×10^{-3} °C	2.16×10^{-4} °C
Ralston's	3.77×10^{-2} °C	6.36×10^{-3} °C	2.16×10^{-4} °C
RK23	2.68×10^{-2} °C	2.99×10^{-4} °C	5.03×10^{-7} °C
RK4	1.14×10^{-3} °C	4.87×10^{-5} °C	6.04×10^{-8} °C
RK45	9.35×10^{-6} °C	1.46×10^{-7} °C	5.60×10^{-9} °C

Table 2. Average accuracy of each method for three different time steps for the second three-thermal-mass system. For adaptive-step solvers, the average time step was used.

h	0.1 s	0.05 s	0.01 s
Euler's	6.80×10^{-2} °C	1.76×10^{-2} °C	3.59×10^{-3} °C
Heun's	3.90×10^{-2} °C	6.24×10^{-3} °C	1.39×10^{-4} °C
Midpoint	3.27×10^{-2} °C	5.96×10^{-3} °C	1.43×10^{-4} °C
Ralston's	2.38×10^{-2} °C	8.73×10^{-3} °C	9.53×10^{-5} °C
RK23	1.14×10^{-2} °C	9.36×10^{-3} °C	4.15×10^{-5} °C
RK4	5.12×10^{-3} °C	1.75×10^{-3} °C	8.67×10^{-7} °C
RK45	5.76×10^{-3} °C	7.69×10^{-4} °C	9.24×10^{-6} °C

These results show that for the given time steps, methods of higher order tend to produce more accurate results, which makes sense for properly implemented solvers. More importantly, the two adaptive-step methods often produced results of the same or higher magnitude of accuracy compared to their fixed-step counterparts for the same average time step. This is most clearly illustrated in the first system of thermal masses, where the RK23 method produces more accurate results for a given time step than the RK second-order methods, and the RK45 method produces more accurate results than the RK fourth-order method. In the second system of thermal masses, it is notable that for very small time steps, RK4 ended up being more accurate than RK45, in an exception to the trend of the adaptive solvers being more accurate for the same average time step.

Efficiency of Implemented Solvers

The computational time was also calculated for each method for several different accuracy levels. Computational time is meant as a relative measure, as different hardware will likely produce faster or slower overall results. For each solver, computational time was calculated by reproducing the same model 10 different times and averaging the computational time taken for each repetition. Results can be found in Figures 5 and 6.

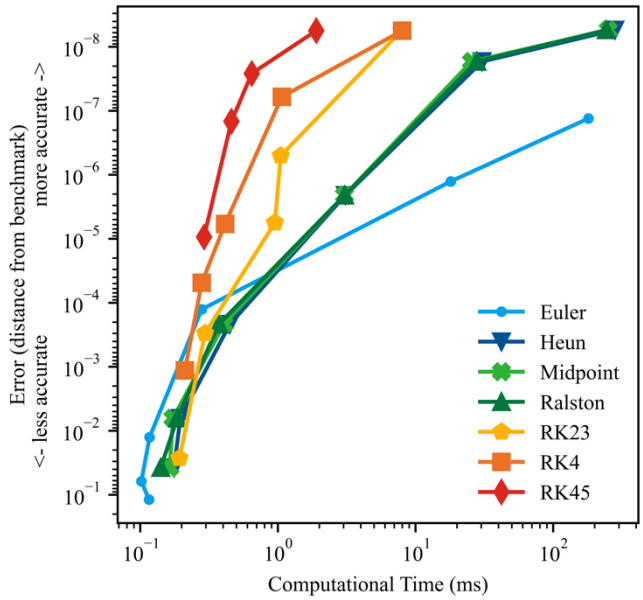

Figure 5. Absolute error of each solver for various computational times, for the first system of thermal masses.

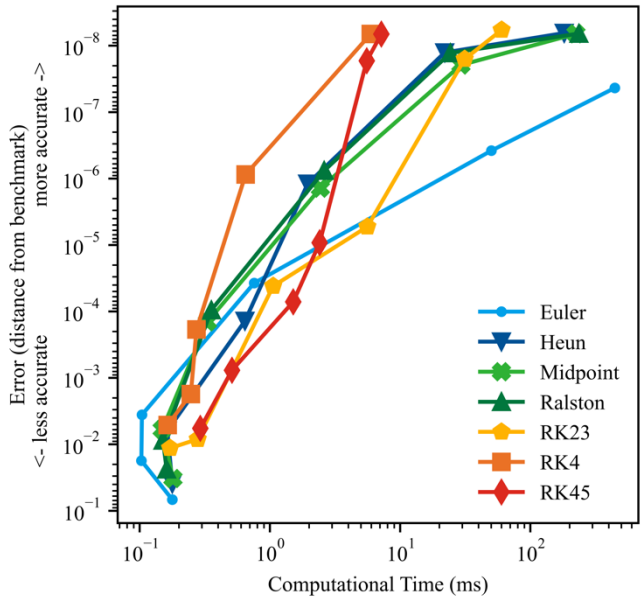

Figure 6. Absolute error of each solver for various computational times, for the second system of thermal masses.

For rough estimates, Euler's method is the most efficient. However, for estimates requiring more accuracy, Euler's method quickly becomes more computationally expensive, and the higher-order methods become more efficient for the same level of accuracy.

Overall, the adaptive solvers tend to be more computationally efficient than the comparable fixed-step methods for high levels of accuracy, although for lower levels of accuracy the fixed-step methods are sometimes more efficient. One notable exception can be found in the second system of thermal masses, where RK4 ends up being more computationally efficient than RK45, even for high levels of accuracy. This could be due to the increased complexity of the second thermal mass system, which necessitates an increased number of iterations and decreased step size for RK45, negating some of the adaptive-step solver's efficiency benefits.

Current Limitations of the Data and Solvers

DESS has only been tested on two systems of thermal masses. Therefore, the robustness of the package will be further supported with additional test cases in a future DESS update. Furthermore, all the solvers implemented in DESS are explicit ODE solvers. Although there are many ODEs for which these solvers can be very effective, fixed and adaptive explicit solvers may not be sufficient for systems with functions that change magnitude very quickly or have a sudden change in boundary condition or behavioral regime (often called "stiff" functions). We intend to release a future update that will add more sophisticated solvers (e.g., implicit solvers) as well as more complicated examples, such as higher-order systems and nested systems with self-contained solvers that sync up at some user-specified or calculated time step interval.

Conclusion and Future Work

DESS demonstrates an implementation of ODE solvers in Rust that allows for flexibility and the use of a variety of solvers and enhances ease-of-use for powertrain modeling. The solvers in DESS are implemented in such a way that the user can specify relationships between different nodes, which DESS translates into a system of equations. It is often easier to conceptualize a system in terms of relationships between different nodes rather than as a system of ODEs, making this capability very useful. It is also clear that the adaptive solvers implemented in DESS have the potential to increase accuracy while also decreasing computational cost for more accurate estimates. However, there are certain cases where the fixed-step solvers (especially the classic Runge-Kutta fourth-order method (RK4)), will be the most computationally efficient and accurate, as can be seen in the results for the second system of thermal masses, where RK4 is more efficient than the Cash-Karp adaptive-step method (RK45), even for very accurate results. The method of error estimation for the adaptive-step solvers means that if the function being estimated has a derivative that changes very quickly, the two function estimations made by the adaptive-step solver might deviate, leading to larger error estimations and a greater number of repeated solver steps. It is possible this could cause the decrease in efficiency seen in the data for RK45 and lead to occasions where the fixed-step solvers will be more efficient. For less accurate estimates, Euler's method still tends to be the best option because it can give a rough estimate at very low computational cost.

There are several additional capabilities that will be implemented in future versions of DESS to increase the program's capabilities and promote ease of use. All the solvers implemented so far have been explicit solvers, which have limitations on the types of ODEs they can effectively solve. Implicit solvers (which require a few initial data points instead of just one) are better able to handle stiff functions [11]. Therefore, future versions of DESS will include some implicit solvers to increase the versatility of the program. Also, unit-checking capabilities are being developed for future versions of DESS, which will help ensure that unit errors are not made when using DESS to create models.

References

1. MathWorks, "Pricing and Licensing: MATLAB Pricing," https://www.mathworks.com/pricing-licensing.html, accessed Feb. 2024.
2. Wolfram, "Mathematica Price: Industry," https://www.wolfram.com/mathematica/pricing/industry/, accessed Feb. 2024.
3. Maplesoft, "Project License – Commercial," https://www.maplesoft.com/products/maple/pricing/project-license.aspx, accessed Feb. 2024.
4. Beierlieb, L., Bauer, A., Leppich, R., Iffländer, L., et al., "Efficient Data Processing: Assessing the Performance of Different Programming Languages," *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*: 83–87, 2023, doi:10.1145/3578245.3584691.
5. Brandefelt, L. and Heyman, H., "A Comparison of Performance & Implementation Complexity of Multithreaded Applications in Rust, Java and C++," Bachelor's Degree thesis, School of Electrical Engineering and Computer Science, KTH Institute of Technology, Stockholm, Sweden, 2020.
6. Ardito, L., Barbato, L., Coppola, R., and Valsesia, M., "Evaluation of Rust code verbosity, understandability and complexity," *PeerJ Computer Science* 7:e406, 2021, doi:10.7717/peerj-cs.406.
7. crates.io, "Search Results for 'ode solver'," https://crates.io/search?q=ode%20solver&sort=downloads, accessed Feb. 2024.s
8. crates.io, "ode_solvers," https://crates.io/crates/ode_solvers, accessed Feb. 2024.
9. peroxide, "Module peroxide::numerical::ode," https://peroxide.surge.sh/numerical/ode/index.html, accessed Feb. 2024.
10. Mathru, "Explicit Methods," https://rustmath.gitlab.io/mathru/documentation/analysis/differentialeq/ode/explicit/, accessed Feb 2024.
11. Chapra, S. and Canale, R., "Numerical methods for engineers, Seventh Edition," (New York, McGraw-Hill Education, 2015), ISBN:978–0–07–339792–4.
12. Euler, L., "Institutionum calculi integralis volumen primum," (St. Petersburg Academy, 1768).
13. Heun, K., "Neue Methoden zur approximativen Integration der Differentialgleichungen einer unabhängigen Veränderlichen," *Zeitschrift für Mathematik und Physik* 45(1):23-38, 1900.
14. Runge, C., "Ueber die numerische Auflösung von Differentialgleichungen," *Mathematische Annalen* 46(2):167-178, 1895.
15. Ralston, A., "Runge-Kutta methods with minimum error bounds," *Mathematics of Computation* 16(80): 431–437, 1962.
16. Kutta, W., "Beitrag zur näherungsweisen Integration totaler Differentialgleichungen," *Zeitschrift für Mathematik und Physik* 46:435–453, 1901.
17. Bogacki, P. and Shampine, L., "A 3(2) pair of Runge - Kutta formulas," *Applied Mathematics Letters* 2(4):321–325, 1989.
18. Cash, J. and Karp, A., "A variable order Runge-Kutta method for initial value problems with rapidly varying right-hand sides," *ACM Transactions on Mathematical Software* 16(3):201–222, 1990.

Contact Information

Robin Steuteville,
Robin.steuteville@nrel.gov,
+1 303-275-4750
Chad Baker,
Chad.baker@nrel.gov,
+1 303-275-3832

Acknowledgments

This work was authored by the National Renewable Energy Laboratory, operated by Alliance for Sustainable Energy, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. Funding provided by U.S. Department of Energy Office of Energy Efficiency and Renewable Energy Vehicle Technologies Office under the Energy Efficient Mobility Systems program. The views expressed herein do not necessarily represent the views of the DOE or the U.S. Government. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this work, or allow others to do so, for U.S. Government purposes.

This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Science Undergraduate Laboratory Internships Program (SULI).

The authors would particularly like to thank Jake Holden for his guidance and support.

Appendix

Cash-Karp Method [18]

$$k_1 = f\big(t_0, y(t_0)\big)$$

$$k_2 = f(t_0 + \frac{1}{5}h, y(t_0) + \frac{1}{5}k_1 h)$$

$$k_3 = f(t_0 + \frac{3}{10}h, y(t_0) + \frac{3}{40}k_1 h + \frac{9}{40}k_2 h)$$

$$k_4 = f(t_0 + \frac{3}{5}h, y(t_0) + \frac{3}{10}k_1 h - \frac{9}{10}k_2 h + \frac{6}{5}k_3 h)$$

$$k_5 = f(t_0 + h, y(t_0) - \frac{11}{54}k_1 h + \frac{5}{2}k_2 h - \frac{71}{27}k_3 h + \frac{35}{27}k_4 h)$$

$$k_6 = f(t_0 + \frac{7}{8}h, y(t_0) + \frac{1631}{55296}k_1 h + \frac{175}{512}k_2 h + \frac{575}{13824}k_3 h + \frac{44275}{110592}k_4 h + \frac{253}{4096}k_4 h)$$

$$O_5 = y(t_0) + \frac{37}{328}hk_1 + \frac{250}{621}hk_3 + \frac{125}{594}hk_4 + \frac{512}{1771}hk_6$$

$$O_4 = y(t_0) + \frac{2825}{27648}hk_1 + \frac{18575}{48384}hk_3 + \frac{13525}{55296}hk_4 + \frac{277}{14336}hk_5 + \frac{1}{4}hk_6$$

(1)

Where,
$k_1, k_2, k_3, k_4, k_5, k_6 =$ intermediate estimated average derivatives
$t_0 =$ initial time
$h =$ time step
$y(t_0) =$ initial value at t_0
$O_5 =$ estimated fifth-order solution
$O_4 =$ estimated fourth-order solution