Efficiency Analysis of Numerical ODE Methods

Braden Schleip, Bennett Challifour

May 12, 2023

Abstract

There are many different numerical methods for approximating differential equations. These methods vary by accuracy and computation cost, which calls for a balancing act from the user.

We created an interactive program named The Methodizer which, given a first-order differential equation of the form $y' = f(t, y)$, figures the best numerical method – between Forward Euler, fourth-order Runge-Kutta, fifth-order Adams-Bashforth, and eighth-order Runge-Kutta-Fehlberg – to equip. To better quantify the priority of the methods, we created a measurement, efficacy, to compare a method's accuracy and computational efficiency.

Analysis from The Methodizer suggests that the fifth-order Adams-Bashforth method is the optimal numerical method in many cases. Computational analysis of efficacy revealed that increasing the number of steps is effective up to a certain point, depending on the function and method used. We analyzed the factors that determined when returns diminished to hypothesize its cause.

The current version of The Methodizer is mathematically limited but still produces the preferred iterative method for a large set of ODEs. Efficacy proved an effective measure of computational efficiency; if given more time, we would further explore the point at which increasing step count gives diminishing returns on accuracy and develop a method to predict the optimal number of steps.

Contents

1 Introduction

Different numerical methods may be preferable for particular situations or functions. For example, a computationally-heavy function $f(t, y)$ may exacerbate the runtime of a method, while a function such as $f(t, y) = sin(1/t)$ may require many more function evaluations for a desired level of accuracy. We sought to address these complications by creating a program, The Methodizer, that employs a set of numerical methods to determine which is preferable for the user. The user may value computational cost or order of accuracy more per their situation, and this program will give the optimal choice for each case.

Mathematical texts have explored the cost efficiency of numerical ODE methods in depth. Much of the inspiration for this project came from the textbook *"Solving Ordinary Differential Equations I"*.[1] Here, Hairer compares the relative error and the number of function evaluations for various numerical methods. We built off of this idea for the concept of efficacy so that we could easily compare different methods with a single value. Finally, we considered various solver packages for creating the "exact" solution, and we elected to use the `lsoda` subroutine after reviewing an analysis of the accuracy of several solvers.[2] Data shows that `lsoda` is consistently accurate when provided with high step counts, and we knew that we would be using high step counts for a strong baseline approximation.

Prominent in the field is the SciPy function `odeint`. This function takes a system of ordinary differential equations and an array of initial conditions. From there, it estimates the solution y over a linear space provided. This highly-optimized solver uses a Fortran pack titled `ODEPACK`, specifically a subroutine called `lsoda` – the acronym for a rather hefty program name.[3] The Livermore Solver for Ordinary Differential Equations with Automatic Method Switching can approximate stiff and non-stiff equations by switching between numerically stable methods for the given function. Initially developed in 1983 to systematically solve stiff systems, this is usually done with backward differentiation for-

[1] Hairer, Ernst, Gerhard Wanner, and Syvert P. Nørsett. 1993. Solving Ordinary Differential Equations I. Springer.

[2] Vigo-Aguiar, J, and Higinio Ramos. "Dissipative Chebyshev Exponential-Fitted Methods for Numerical Solutions of Second-Order Differential Equations." Journal of Computational and Applied Mathematics, (2003).

[3] Hindmarsh, Alan C. "ODEPACK: A Systematized Collection of ODE Solvers." Scientific Computing, (1982).

mulae, while non-stiff systems are primarily handled through Adams-Moulton methods.[3]

2 Methods

When given an input differential equation, The Methodizer applies four different numerical methods for comparison: the forward Euler method, the fourth-order Runge-Kutta (RK4) method, the Adams-Bashforth multistep method, and the eighth-order Runge-Kutta-Fehlberg (RKF8) method. This process unveils a spectrum of options for the user's preference: the forward Euler, for example, is the least computationally expensive at the cost of stability and accuracy, while the RKF8 method is exceedingly accurate and requires considerably more function evaluations.

From here, we compare accuracy with respect to the number of function evaluations. The ideal method would maximize accuracy while minimizing function evaluations to be precise and efficient. However, for different scenarios, one method may be preferable over another. For example, a computationally dense function may motivate the user to sacrifice some accuracy in trade for decreased runtime.

2.1 Developing the Program

2.1.1 Early Development

To provide a reference for future testing, we applied various step sizes and methods on the simple equation $y' = y$ under the initial condition $y(0) = 1$. We fixed the step size to 0.001 and set an interval of 0 to 5. Over these 5000 steps, we tested the Forward Euler, Backward Euler, fourth-order Runge-Kutta, fifth-order Adams-Bashforth, and eighth-order Runge-Kutta-Fehlberg methods.[4] While taking the most function evaluations, RKF8 was unsurprisingly the most accurate of the set; its absolute error approached machine precision around the first 2000 steps. (See Figure 1).

We initially considered reporting the runtime of each method but later replaced this function. Figure 2 demonstrates a runtime analysis for each method on the equation $y' = y$ with 5000 steps. However, for lower step counts, the speed caused the program to output inconsistent or negligibly different runtime values. Furthermore, the runtime may vary by the machine running the program and the number of other processes currently active. For these reasons, we elected to use total function evaluations to compare the computational strain of a specific method.

2.1.2 Integrating lsoda Into Our Methods

The Methodizer uses a text interface for the user to input a first-order ODE of their making. The program then constructs various files to run the numerical

[4]See program `y=y.c`

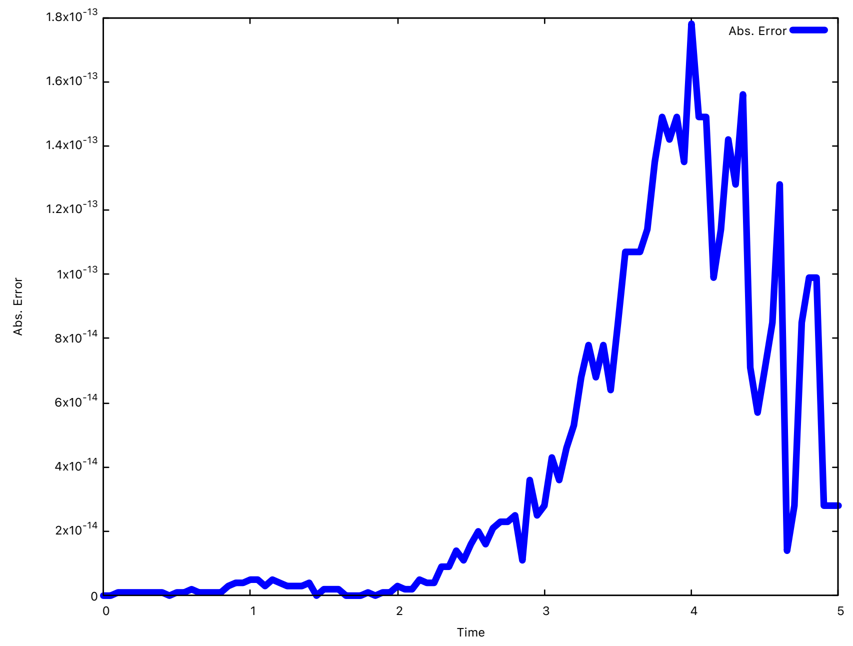


Figure 1: Absolute error of RFK8 over the interval from 0 to 5 using 5000 steps

Method Runtime for $y' = y$ in Microseconds					
Method	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
F. Euler	350	480	470	460	400
B. Euler	470	660	650	550	560
RK4	1470	1710	2270	1710	1860
AB-4	740	960	840	960	970
RKF8	6570	8740	9340	8730	8090

Standard Deviation					
Method	F.Euler	B.Euler	RK4 3	AB-4	RKF8
σ	55.4	66.8	219.5	101.4	1060.3

Figure 2: Variation in method runtime on the function $y' = y$ in microseconds for 5000 timesteps.

methods on the given equation, each of which counts the number of function evaluations and determines the accuracy per method. To make the program work for any arbitrary inputted function, we were required to generalize the finding of a pseudo-exact solution. The existence of equations without an analytical solution, such as $y' = atan(y)$, rendered a perfect comparison impossible. For consistency, we used the `lsoda` program to provide an approximate solution for the function given. Mathematically, this provides limitations for the accuracy of results, as we can only use an approximate baseline comparison.

However, `lsoda` was originally written in Fortran, and thus we needed to convert the subroutine into a C program. Simon Frost, the principal data scientist at Microsoft, wrote a version of `lsoda` optimized for C that was publicly accessible. We incorporated this complicated C program into the files we had previously created so that we could use `lsoda` to compare the accuracy of each method. Thus, we define the ODE we want to solve in the method fex. In the *test* method, we provide initial parameters and step size. The *test* method then calls `lsoda` using the information provided to solve the ODE over the given interval. The outputted values are stored in an array so that we may compare our approximation methods.

To ensure its accuracy, `lsoda` requires the user to specify relative tolerance $RTOL$ and absolute tolerance $ATOL$, which define an error control vector, $e(i)$, of estimated local errors in $Y(i)$ at a given time step i. Each of these errors is subject to the condition:[5]

$$\text{root mean square-norm} \quad \frac{e(i)}{EWT(i)} \leq 1 \tag{1}$$

where

$$EWT(i) = RTOL(i) \cdot |Y(i)| + ATOL(i). \tag{2}$$

Due to machine precision, relative and absolute tolerance are restricted to a minimum magnitude of 10^{-13}. Figure 3 compares the `lsoda` results with the analytical solution for the equation $y' = y$ with a time step of 0.001.

Thus, after integrating `lsoda` into our methods, we were no longer restricted by the type of function. [6]

2.1.3 Creating An Interactive UI

We then built a text-based user interface that would allow the user to test the accuracy of four numerical methods: Forward Euler, RK4, Adams-Bashforth, and RKF8 – adding a general implementation for Backward Euler proved cumbersome. In The Methodizer, the user may write their own ODE, set initial conditions, and choose a step size. We quickly ran into an issue: the function could not write itself while the program executed. To sidestep this, the program

[5]Hindmarsh, Alan C. "ODEPACK: A Systematized Collection of ODE Solvers." Scientific Computing, (1982).

[6]See yatany.c in the folder integrating lsoda

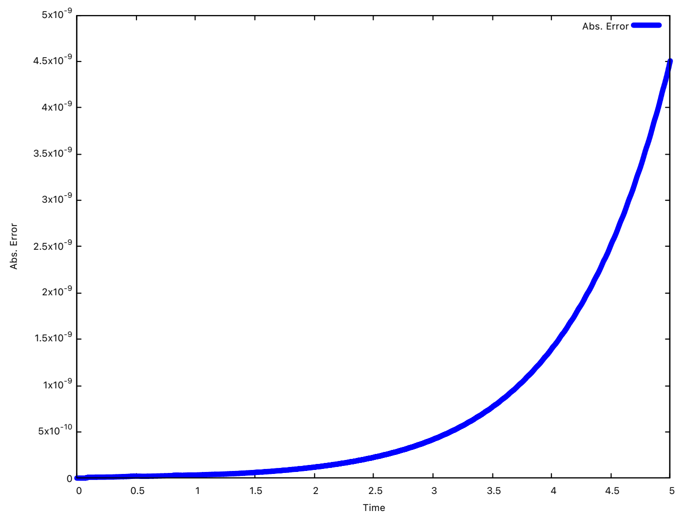


Figure 3: Absolute error of `lsoda` for $y' = y$

writes a unique file per unique function inputted, followed by a specialized Makefile that compiles, loads, and links the file to the `lsoda` library. The program summarizes the effectiveness of each of the four methods with their number of function evaluations, accuracy, and efficacy.[7]

2.1.4 Why C?

The code for The Methodizer is written in C to prioritize runtime. The language achieves this through a lack of automatic garbage collection, array index bound checking, and external libraries, optimizing the program without unnecessary external checks. To demonstrate its speed, consider the simple program which sums the numbers from 0 to 1,000,000. C can complete this program in 930 microseconds, while the equivalent program in Python took 90,207 microseconds – almost 100 times slower. The Methodizer will be required to compute hundreds, if not thousands, of arbitrarily complex function evaluations, so runtime quickly became a crucial factor in its usability.

There were also evident disadvantages to using C. The input-output framework is difficult to use; C requires that every printed variable is cast to a specified type, and heap memory must be manually allocated and freed, lest we encounter a fatal segmentation fault error. Furthermore, errors are reported during compiling rather than writing, so incremental development and testing slowed the programming process.

[7]See textUI.c in the Final Development folder.

2.2 Comparing Outputs

After terminating its computation, The Methodizer will output each method's number of function evaluations and accuracy. Then, each method is presented and processed to determine its efficiency with respect to computation cost. So, we sought to quantify the relationship between a method's computational cost and accuracy. Under this relationship, we prefer lower computational costs and higher accuracy. Furthermore, a derivation with a doubling of computational cost resulting in a doubling of accuracy should provide an equivalent measurement of efficiency as the previous derivation.

First, we determined how to quantify the accuracy of a method. Define error as the difference between the pseudo-exact solution and the numerical approximation. Because error should scale logarithmically, a naive average is insufficient. Thus, we can take the log of the absolute value of each error value, then find the arithmetic average of those values:

$$A = \frac{\sum \log_{10} |e_i|}{N} \tag{3}$$

with A being the measurement of accuracy, each e_i term being the error at each step i taken by the method, and N being the total number of steps. We arbitrarily chose to use a base-10 logarithm for a more immediate intuitive grasp of scale in review.

This method finds a slight complication: due to machine precision, steps in which the error is arbitrarily close to 0 cause the sum to tend toward infinity. As such, it is necessary to filter values before including them in the total by ensuring each $e_i > 10^{-16}$. The count should also be adjusted to only increment when one of these values passes.

So we can write a more robust equation:

$$A' = \frac{\sum \log_{10} |e'_i|}{N'} \tag{4}$$

with e'_i and N' being the filtered values.

Another note, with this equation for accuracy, is that it will regularly return a negative value, as most of the $|e_i|$ values should be less than 1. This is expected and accounted for when we include the factor of computation cost.

Now that average accuracy is quantified, we add a factor of computation cost. This is simple: for a doubling of computation cost to perfectly offset a doubling of accuracy, we take the base-10 logarithms of the number of function evaluations for the method. Thus, letting σ be the number of function evaluations for a given method, we can quantify this interpretation of the efficiency, called efficacy, represented with E:

$$E = -\left[A' + \log_{10}(\sigma)\right]. \tag{5}$$

By this metric, a higher efficacy value of E corresponds to a more efficient method with respect to both accuracy and computation cost.

Abs. Error of $y' = 2ty/(1+t)^2$				
Step size	F.Euler	RK4	Adams-Bashforth	RKF8
1	-1.75E+01	7.59E+00	-6.49E+00	-7.33E+00
0.5	-1.18E+01	-1.02E+00	4.37E+00	-9.99E-01
0.1	-3.24E+00	7.43E-01	1.74E+00	7.43E-01
0.05	-1.70E+00	4.36E-01	4.96E-01	4.36E-01
0.01	3.53E-01	9.75E-02	2.08E-02	9.77E-02
0.005	-1.78E-01	4.94E-02	5.20E-03	4.97E-03
0.001	-3.57E-02	9.97E-03	2.08E-04	9.37E-05

Figure 4: Absolute error of each numerical method with respect to step size, on the function $y' = 2ty/(1+t)^2$.

3 Results

3.1 Methodizer Results

When given the same parameters, results from The Methodizer were consistent with mathematical predictions. As the step size decreased, the error after integrating over the interval dropped significantly for each method. In alignment with the theoretical expectations, RKF8 was the most accurate method, with global error proportional to $O(h^8)$. However, once the step size reaches around 0.001, the differences in the errors of the Adams-Bashforth and RKF8 methods became negligible. The results from changing step size for the function $y' = 2ty/(1+t)^2$ are summarized in Figure 4.

Note that after the step size dropped below 0.001, the methods became less accurate. At this point, the accuracy of the approximations dips below the tolerances of `lsoda` .

Figure 5 compares the efficacy values of different numerical methods on the function $y' = y$. Above 100 function evaluations, the Adams-Bashforth method produces the highest efficacy values. This analysis demonstrates that, for $y' = y$, Adams-Bashforth is most computationally efficient. Nevertheless, RKF8 returns the most accurate results at the cost of an increased runtime.[8]

3.2 Efficacy Analysis

The efficacy value of a given numerical method varies depending on the function to which it is applied, the pertinent interval, and the step size. As we attempt to optimize the efficacy for a given input, the user is restricted in their choice of function and interval. Thus, choosing the right step size becomes necessary to optimize a numerical method.

The plot in Figure 6 demonstrates an efficacy analysis on the function $y' = y$ over the interval [0,3]. We evaluate the method's efficacy at various step counts

[8]To see a demonstration of The Methodizer, navigate to `https://drive.google.com/file/d/1gVfzS-4zufOjdumD_swbjxMQuOpHI-zl/view?usp=sharing`

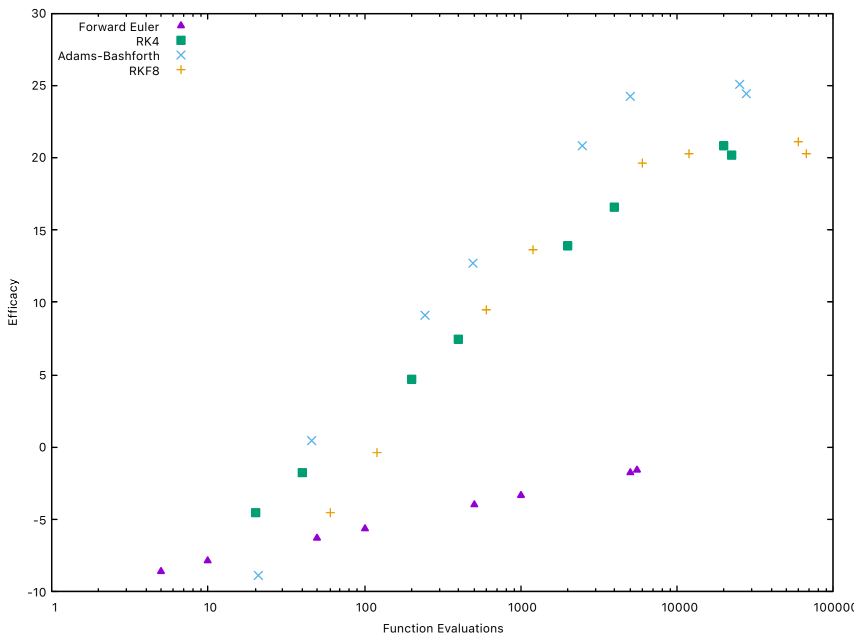


Figure 5: Efficacy Dependant on Function Evaluations for $y' = y$

ranging from 10 to 15000, and the relationship between the two is graphed. We see from here that, for this method, efficacy scales logarithmically with respect to the number of steps – because step count is in the formula for efficacy, we can then determine that, up to a point, the method becomes more efficient as the number of steps increases. This pattern confirms that increasing the number of steps is computationally efficient. Most noticeable in the graph is the point where the line ceases to be straight. Beyond approximately 3700 steps, the efficacy of the function diminishes as the number of steps increases.

The step count of this breaking point depends on both the method and the function involved. Figure 7 shows the eighth-order Runge-Kutta-Fehlberg method subjected to the same parameters. Note that this method finds optimization at around 2600 steps and peaks around the same efficacy value as shown in the previous figure. Considering the equation for efficacy, this pattern suggests that the payoff of increasing step count begins to diminish when the method reaches machine precision.

To explore the effect of the function chosen on the efficacy analysis, we ran the program for the equations $y' = y$ and $y' = 3y$ to create the plots shown in Figure 8. As the coefficient of y increases in the differential equation, the step count for optimal efficacy increases.

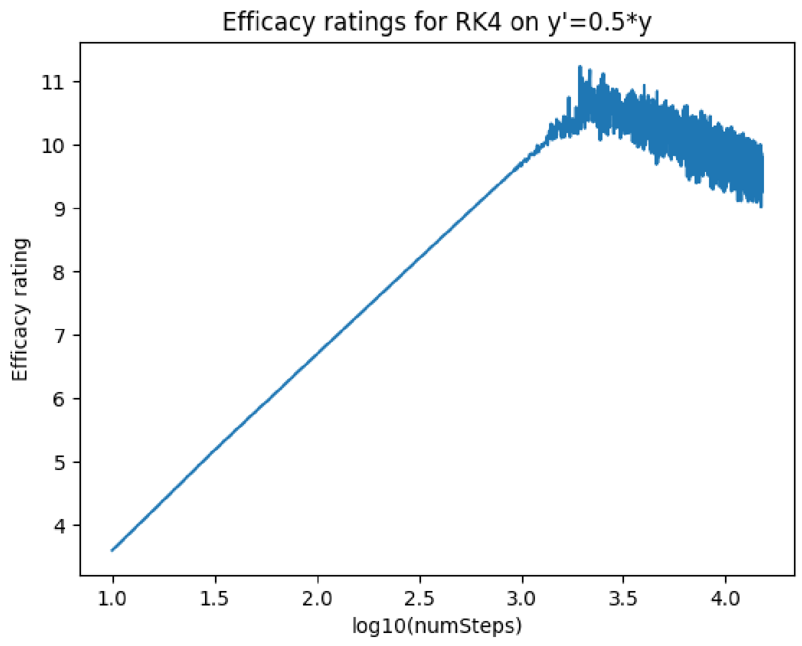


Figure 6: A semilog plot of efficacy ratings with respect to number of steps taken by the RK4 method on the function $y' = 0.5y$ over the interval [0,3].

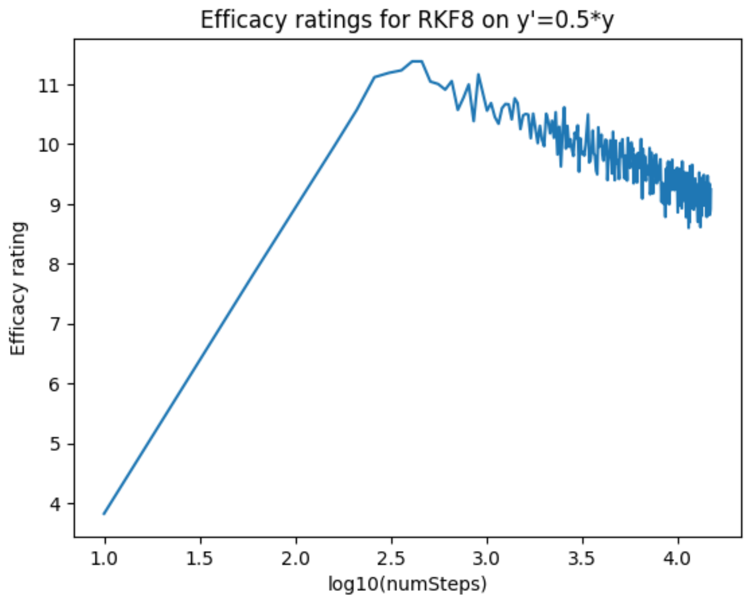
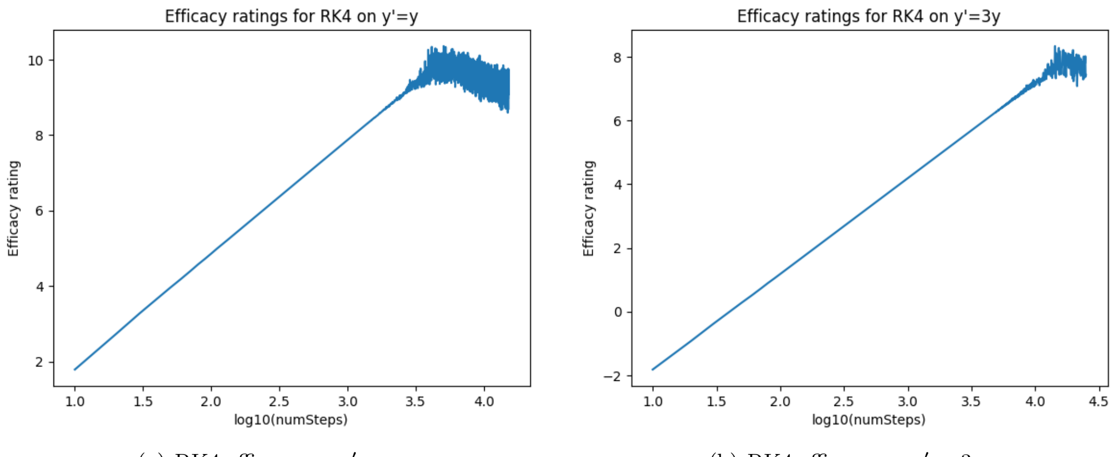


Figure 7: A semilog plot of efficacy ratings with respect to the number of steps taken by the RKF8 method on the function $y' = 0.5y$ over the interval [0,3].



(a) RK4 efficacy on $y' = y$

(b) RK4 efficacy on $y' = 3y$

Figure 8: Semilog plots of efficacy and step counts for the functions $y' = y$ and $y' = 3y$ through RK4.

4 Conclusions

4.1 Limitations of The Methodizer

By comparing the efficacies of each method, The Methodizer accurately informs the user which numerical method prioritizes accuracy and computational cost. While we generalized The Methodizer to a wide set of ODEs, limitations persisted. We fix the integration interval between 0 and 5; a future version would allow user-inputted boundaries. Furthermore, because the program numerically calculates each value, it is susceptible to division-by-zero errors. We theorize that this could be fixed by setting an offset near machine precision. The Methodizer's most prominent limitation is its restriction to one-dimensional systems, as most dynamic systems are not one-dimensional.

Further development of The Methodizer would include support for coupled ODEs and possibly a wider variety of iterative methods. `lsoda` allows solving stiff and non-stiff systems, but our program is limited to non-stiff systems.

To make The Methodizer more efficient, we would adjust the file compilation. For each unique input, only 17 lines of the source program need rewriting. A more optimal program would use the `fseek()` function to replace only these lines in `ode.c`, eliminating hundreds of lines of code from The Methodizer file, `textUI.c`.

4.2 Exploring Efficacy Analysis

Introducing the metric of the efficacy of a method revealed both limitations of our programs and new facets that we would have liked to explore more in-depth. Still, efficacy provided a reliable metric of the computational efficiency of a function for a given step count.

After The Methodizer runs its analysis and reveals the best choice of method, we wanted to create a program that, given the function and method, would return the optimal number of steps to maximize the efficacy. We, unfortunately, ran into issues with reliably finding accuracy. These issues could be a consequence of the generalization of the program – without always having an analytic solution, using `lsoda` could result in errors that damage the reliability at small step sizes.

At the end of the Efficacy Analysis section, we discovered a relationship between the value of λ in the equation $y' = \lambda y$ and the optimal step count. It would be intriguing to explore the cause of this pattern; we hypothesize that this change is due to the increasing truncation error, as each derivative changes more rapidly. From there, it takes more steps for the method to reach the limit of machine precision, and thus the optimal step count is higher. If we continued the analysis, we would explore the relationship between the amount that a function increases and the optimal number of steps.