1 Introduction

VI ================= check if double quotes are in the right direction ================== VI In this report, we will provide the results of a careful investigation of the performance of a variety of software packages applied to a typical initial value ordinary differential problem encountered in Covid-19 modeling.

For any mathematical model, the accuracy requirements of the numerical solution should be determined by the quality of the model and the accuracy of the parameters that appear in the model. Numerical errors associated with the computational techniques that are used to obtain the solution must always be negligible compared to the accuracy to which the model is defined. *Researchers deserve to obtain numerically accurate solutions to the models that they are studying.* In this report, *we will show that the straightforward use of standard IVODE solvers on typical Covid-19 models can lead to numerical solutions that have large errors, some of the same order of magnitude as the solution itself.*

In Section 1.1, we review examples of how Initial Value Ordinary Differential Equation Solvers (IVODES) are used in epidemiology. In Section 1.2, we define the SEIR models which we will consider throughout this report, in Section 1.3, we discuss the problem of stability as it concerns problems with exponential growth, in Section 1.4.1, we explain the difference between fixed step-size and error-controlled solvers. The IVODE software packages from programming environments that are typically used by researchers are described in Section 1.4.2. We also make a note of issues with evaluation at output points that lead to inefficiencies in Section 1.5. In Section 1.6, we discuss the effects of problem discontinuities on the performance of these solvers.

In Section 2.1, we apply the solvers to the problem with a time-dependent discontinuity and show how this results in numerical solutions with relative errors of the same magnitude as the solution we are trying to compute. In Section 2.2, we will use discontinuity handling to solve the time-dependent problem. In Section 2.3, we will use a range of tolerances to discuss the effects of tolerance on the accuracy and efficiency of the solvers.

In Section 3.1, we apply the solvers to the Covid19 problem with a state-dependent discontinuity and show how none of the solvers were able to obtain accurate solutions. We will explain how even the use of very sharp tolerances does nothing to improve the models in Section 3.2 and show that the only proper way to solve this problem is through event detection, which we will describe in Section 3.3. We then show the correct solution to the problem in Section 3.4 and perform a tolerance study on this problem in Section 3.5.

In Section 4, we examine the implementation details for some of the solvers to investigate the cause of the inaccuracies. We conclude the report in Section 5 with a summary and the identification potential for future work projects.

1.1 Epidemiological modelling

One common form of an epidemiological study is forecasting. Using previously obtained parameters, the researcher develops a mathematical model involving differential equations which are solved using an ODE solver. Often, the solver will be used to integrate over a large time period so that the researcher can examine how the diseases will spread. For such problems, sharp tolerance values should be used and the solver-problem combination is expected to be resilient over large time periods. In Section 1.3, we discuss why it is unrealistic to attempt to compute a numerical solution for large time periods if the infection is still growing exponentially but how measures such as social distancing allow solvers to reduce modeling errors so reasonably accurate solutions can be computed over longer time periods.

A second type of epidemiology study involves parameter estimation. In this kind of study, data points are collected about the spread of a virus and we try to fit a mathematical model through that data. In so doing, we can estimate the parameters used by looking at which parameters minimize the error in the fit. The parameters estimated can tell us if implemented control measures are working and may point out what can be done to improve the situation. An example of such a study can be found in Section 7. Also, parameters so estimated can be used for the first kind of study. Parameter estimation studies often involve using an ODE solver inside some optimization algorithm and thus the computing time, especially with large problems, can be significant. We will therefore investigate whether or not researchers can coarsen the tolerance.

1.2 Detailed description of two specific models to be considered in this report.

In this section, we explain how an IVODE problem is defined. We then (Reference Christina Christara.) describe the models that we are going to consider in this report. They involve typical SEIR models to which we add discontinuities.

An IVODE problem is defined with:

$$y'(t) = f(t, y(t)), y(t_0) = y_0 \tag{1}$$

where f is a function that defines the derivative at that that point in time, t. A complete definition also includes the initial value of the solution. Thus given $y'(t)$ and $y(t_0)$, we find $y(t)$ using numerically methods.

To fulfill the complete problem definition, we define as models as follows:

$$\frac{dS}{dt} = \mu N - \mu S - \frac{\beta}{N} IS, \tag{2}$$

$$\frac{dE}{dt} = \frac{\beta}{N} IS - \alpha E - \mu E, \tag{3}$$

$$\frac{dI}{dt} = \alpha E - \gamma I - \mu I, \tag{4}$$

$$\frac{dR}{dt} = \gamma I - \mu R, \tag{5}$$

In this SEIR model, we describe the epidemic over time. S is the number of susceptible individuals, E is the number of exposed individuals, I is the number of infected individuals and R is the number of recovered individuals at a point in time. We also use N to represent the population size. The other parameters in this model are as follows: α is such that α^{-1} is the average incubation period, β is the transmission rate, γ is the recovery rate and μ is the replenishment rate. In this report, we assume that all these parameters are known as our goal is to investigate the performance of IVODE solvers on the test problems. We will see that we get solutions that are not efficiently computed or that may have significant errors. This issue can have serious consequences as it will fail to show the actual impact of the virus as it corresponds to the actual epidemiology theories behind the mathematical models. These incorrect numerical solutions may lead epidemiologists into reaching incorrect conclusions and thus lead them to try to change the mathematical models themselves when, in fact, it is the solvers which are at fault.

The discontinuities we are going to consider involve the parameter β. Before measures such as social distancing and others are implemented, β has a much higher value than after. In our case, β will be 0.9 before the measures and 0.005 after they are implemented. Such an abrupt change in a modeling parameter introduces a discontinuity as we will show in Section 1.6.

For the time-dependent discontinuity, we will assume that at some point in time, measures are implemented that will lead to a reduction in the parameter β. We would like to model the problem through this discontinuity but as we will show, this discontinuity introduces a numerical issue.

For the state-dependent discontinuity, we consider the following situation. If the population of exposed people reaches a certain maximum threshold, measures are introduced, which decreases the value of β. This introduces a discontinuity. Then, when the population of exposed people drops below a certain minimum threshold, the measures are relaxed, which increases β back to its original value, which introduces another discontinuity. We will try to model this problem through a long time period corresponding to 'waves' of the pandemic. We note that each time we change the parameter β, a discontinuity is introduced and thus this problem is far more discontinuous than the previous one, which had only one discontinuity. In so doing, we show that all the solvers will fail.

The other parameters are assumed to always be constant with N at 37,741,000 (the approximate Canadian population size), α at 1/8, γ at 0.06, and μ at 0.01/365.

The initial values are E(0) = 103, I(0) = 1, R(0) = 0 and S(0) = N - E(0) - I(0) - R(0).

This gives us a complete system of initial value ordinary differential equations that is in a form that can be solved by typical software packages.

1.3 Exponential growth and the issue of instability

It turns out that some of the solution's components to the SEIR model exhibit exponential growth over certain time periods. In this section, we discuss exponential growth and its impact on the computation of a numerical solution. First of all, we give a quick overview of stability for ODEs. Then we will show that the SEIR model is unstable and how measures such as social distancing can improve the stability. This is important as this essentially means that before measures are implemented, accurate models are for the most part very difficult to obtain but the addition of the measures such as social distancing can give us hope that the solvers will perform better.

The stability of an ODE is associated with the impact of small changes to the initial values on the solution to the problem. An ODE is unstable if a small change in the initial values results in drastically different solutions; otherwise, the ODE is said to be stable.

It is straightforward to see that the solution to problems that exhibit exponential growth are unstable. This is the case with some of the solution components of a Covid-19 model. The population of infected people grows exponentially for as long as there are no measures. This means that ODE solvers will experience difficulties in obtaining accurate numerical solutions.

In Figure 1, we show exponentially growing solutions corresponding to models with slightly different initial value of E(0). We can see that we get different solutions, that become even more different as time increases.

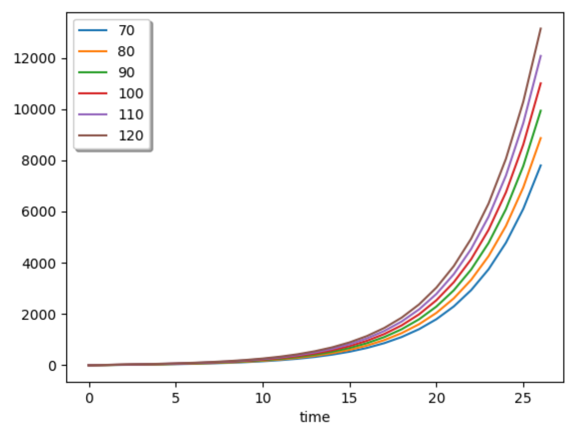

Figure 1: Instability of exponential growth

However, when we introduce measures such as social distancing, which leads to a smaller β value, the problem will exhibit slower exponential growth or can even show exponential decay. A slower exponential growth means that the solution will not be as sensitive to changes to the initial values. Exponential decay is even better as the solutions from different initial values will converge.

Epidemic modeling problems exhibit solutions with this type of behavior. At first, the problem is unstable but as measures are implemented, which lead to exponential decay rather than growth, the problem becomes stable. We show this in Figure 2 for the problem with the time-dependent discontinuity. At first, the solutions diverge during exponential growth, but the addition of measures such as social distancing introduce exponential decay which makes them converge. Thus the measures not only save lives but also improve the accuracy of the computed solutions.

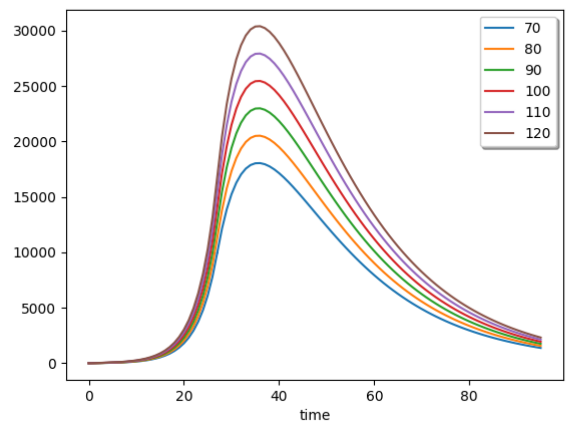

Figure 2: Unstable solutions in the region [0, 40] gain in stability as measures are implemented in the region [40. 90]

1.4 Brief overview of numerical software to be employed in the numerical solution of this model

We start by explaining how solvers attempt to solve an IVODE problem. Given the values at the initial time, t_0, the solver will use an initial step size, h, to compute a solution at time, $t_1 (= t_0 + h)$. Similarly, the solvers will attempt to take a sequence of steps until it reaches the end time. High-quality solvers will then run an interpolation algorithm either locally within each step or globally across the sequence of points to get the numerical solution. We note that a solver is said to have order, p if the difference between the actual solution and the computed solution is equal to $O(h^p)$

In the next section, we describe what a solver will attempt to do to improve the accuracy of their solutions. We then discuss the numerical solvers we are going to use throughout our investigation. We will then have an additional discussion on the implementation of the interpolation to get the final solution and how certain programming environments have not updated their solvers to use interpolation.

1.4.1 Fixed Step Size and Error Control Solvers

We begin our discussion on the software used by giving a brief overview of what role is played by the tolerance and the difference between fixed step size and adaptive step-size error control solvers.

The tolerance is a measure of how accurate we want the solution computed by the solvers to be. Generally, an absolute tolerance of 10^i means that we want the error estimate to be within 10^i whereas a relative tolerance of 10^i means that we want the ratio of the error estimate and the computed solution to be within 10^i. This is not always the case as some solvers will make a blended use of the provided absolute and relative tolerance.

A solver is said to have a fixed step size if it does not compute an error estimate that can be compared to the tolerance. The solver will have an initial step-size and this step-size is used throughout the whole integration. It will jump from one point to another point in a 'step' and will not check if the numerical solution it obtains at the end of the step is sufficiently accurate. Thus, the distance between the points, i.e the step size, is constant throughout the computation.

An error controlled solver starts with an initial step size but as it takes a step, it will compute an error estimate and based on the tolerance will repeat the computation with a smaller step-size if the error estimate is larger than the tolerance. It will repeat the process until the error estimate satisfies the given tolerance. Only then will it move to the next step. Thus it reduces the step-size to step throughout the computation. Also, if the error estimate is much smaller than the tolerance, it will increase the step-size for the next step. This allows it to make sure that the given tolerance is satisfied over the whole problem interval and that as large a step as possible is being taken to optimize the efficiency of the computation.

Error control is not simple to implement. This is where we need to caution against the use of non-standard ODE solvers or fixed step-size solvers. Also, some researchers, who have some understanding of ODEs may be tempted to write their solvers. These researchers often program a non-error control method like a simple fixed step-size Euler or Runge-Kutta method. We will show, using provided fixed step-size solvers in R, how these solvers simply cannot solve a Covid-19 model. Without error control, these solvers cannot handle the discontinuity and stability issues that are present in these models and they will give very erroneous solutions, often without even a warning that the computed solutions should not be trusted.

1.4.2 The packages

There are different types of ODE solvers that are grouped in the following classes: Runge-Kutta methods, Runge-Kutta pairs, and multi-step methods.

A Runge-Kutta method is a one-step method using function evaluations along the step. It integrates with a sequence of steps and has no error control. An example is the classical Runge-Kutta method.

A Runge-Kutta pair is a solver that uses two Runge-Kutta methods. One of the methods is used to compute a solution and a second method is used to compute an error estimate. The then resize the step based on that error as discussed previously. An example is the DOPRI5 pair that uses a fifth-order method for the solution and a fourth-order method for the error estimate.

a multi-step method is a solver that will use a linear combination of the previous step to take the next step. An example is LSODA.

R packages Scientists who solve ODE models in R commonly use the deSolve package and the $ode()$ function within it. $ode()$ provides many numerical methods to solve a problem but we have focused our investigation only on the following popular choices: 'lsoda', 'daspk', 'euler', 'rk4', 'ode45', 'radau', 'bdf' and 'adams'. The default method is 'lsoda' and the default tolerances are 10^{-6} for both the absolute and relative tolerances. We also note that we did not consider the other integrators in the deSolve package like $rkMethod()$, which present other Runge-Kutta methods, and the other ones which are called by the $ode()$ function itself.

The error control methods solvers are:

- Using 'lsoda' calls the Fortran LSODA routine from ODEPACK. It can automatically detect stiffness and choose between a stiff and a non-stiff solver.
- Using 'daspk' calls the Fortran DAE solver of the same name.
- Using 'ode45' calls an implementation of Dormand-Prince (4)5 (DOPRI5), written in Fortran. We note that it is not using DOPRI5.f.
- Using 'radau' calls the Fortran solver RADAU5 which implements the 3-stage RADAU IIA method.
- Using 'bdf' calls the stiff solver inside the Fortran LSODE package which is based on a family of BDF (Backward Differentiation) methods.
- Using 'adams' calls the non-stiff solver inside the Fortran LSODE package which is based on a family of Adam's methods.

The fixed step-size solvers are:

- 'euler' which calls the classical Euler method which is implemented in C.
- 'rk4' which uses the classical Runge-Kutta method of order 4 which is implemented in C.

We will use these two methods to demonstrate what happens when researchers program their non-error-controlled solvers.

We also make note that R uses a dubious method to find the values of the solution at output points. This results in efficiency issues as we will discuss in Section 1.5.

We next talk about the R's interface, the 'ode' function is only given a vector of the output points. The function decides itself when to use interpolation or stopping at output points as described in Section 1.5. There are ways to use interpolation explicitly but Researchers will have to dive into the documenting papers/source code to find it.

Python packages In Python, researchers use the scipy.integrate package and will normally use the *solve_ivp*() function due to its newer interface. It lets the user apply the following methods: 'RK45', 'RK23', 'Radau', 'BDF', 'LSODA' and 'DOP853'. In this report, we will investigate all of these methods. The default solver in *solve_ivp*() is 'RK45' and the default tolerance is 10^{-3} for the relative tolerance and 10^{-6} for the absolute tolerance. All of these solvers employ some form of error control.

- Using 'RK23' uses an explicit Runge-Kutta pair of order 3(2). This uses the Bogacki-Shampine pair of formulas. It is a Python implementation.
- Using 'RK45' uses an explicit Runge-Kutta pair of order 5(4). This uses the Dormand-Prince pair of formulas. It is a Python implementation.
- Using 'DOP853 uses an explicit Runge-Kutta pair of order 8. It is a Python implementation.
- Using 'Radau' uses an implicit Runge-Kutta method, the three-stage Radau IIA method of order 5. It is a Python implementation.
- Using 'BDF' uses a method based on backward-differentiation formulas. This is a variable order method with the order varying automatically from 1 to 5. It is a Python implementation.
- Using 'LSODA' calls the Fortran ODEPACK for LSODA. This switches between an Adams (non-stiff) and a BDF (stiff) method as it detects stiffness. This is implemented in Fortran.

We note that all solvers in *solve_ivp*() have error control and that only 'LSODA' is using the Fortran package itself, the others are a Python implementation and will likely be slower.

We next talk about Python's *solve_ivp*() interface. It can integrate using only the initial time and the final time where it will return the output points where the steps stopped. It can also take a *t_eval* vector which does local interpolation. It lets the solver take as big a step as it can and does not force the solver to stop at the output points. Thus it does not suffer from the inefficiencies described in Section 1.5. The interface also has a *dense_output* flag. This returns an interpolation on the whole time range and allows the user to use the result as a function.

Scilab packages In Scilab, researchers solve differential equations with the built-in $ode()$ function which has the following methods: 'lsoda', 'adams', 'stiff', 'rk', 'rkf'. The default integrator is 'lsoda'. Default values for the tolerances are respectively 10^{-5} for the relative tolerance and 10^{-7} for the absolute tolerance for all solvers used except 'rkf' for which the relative tolerance is 10^{-3} and the absolute tolerance is 10^{-4}. All of these solvers are error control solvers.

- Using 'lsoda' will call the Fortran LSODA code from ODEPACK. This switches between an Adams (nonstiff) and a BDF (stiff) method as it detects stiffness. It has error control.
- Using 'stiff' calls the stiff solver inside Fortran's LSODE package which is based on a family of BDF (Backward Differentiation) methods.
- Using 'adams' calls the non-stiff solver inside the Fortran LSODE package which is based on a family of Adam's methods.
- Using 'rk' calls an adaptive Runge-Kutta method of order 4. It uses Richardson extrapolation for the error estimation. It is implemented in Fortran in a program called 'rkqc.f'.
- Using 'rkf' calls the Shampine and Watts program based on Fehlberg's Runge-Kutta pair of order 4 and 5 (RKF45) pair. It calls the Fortran rkf45.f.

We next talk about the $ode()$ interface in Scilab. It takes a vector of time steps and the code decides whether to use interpolation or to stop the integration at the output points as described in 1.5. We note that Scilab's 'rkf' is a very old software package that it does not even have interpolation.

Matlab packages In Matlab, researchers solve differential equations with the built-in $ode()$ family of functions. We will consider two of those in $ode45$ and $ode15s$. Default values for the tolerances are respectively 10^{-3} for the relative tolerance and 10^{-6} for the absolute tolerance for all solvers as they all use the $odesetoptions$.

- Using $ode45$ calls a Matlab implementation of DOPRI5.
- Using $ode15s$ employs an algorithm that is a variable-step, variable-order (VSVO) solver based on the numerical differentiation formulas (NDFs) of orders 1 to 5. Optionally, it can use the backward differentiation formulas (BDFs, also known as Gear's method) that are usually less efficient. It is a stiff ODE solver.

We next talk about Matlab's ode suite's interface. The interface takes the initial and final time only and allows the solvers to take as big a step as it needs. All plots are then done using interpolation by the plotting software so it does not suffer from the issues discussed in Section 1.5

How the packages relate We tried to find connections across the programming environment where the solvers appear to be using the same source codes. Here is what we found:

R's, Python's, and Scilab's 'lsoda' are all wrappers around the Fortran LSODA code inside ODEPACK.

R's 'bdf' is equivalent to Scilab's 'stiff' in that they use the LSODE code inside ODEPACK but Python's "BDF" is a different implementation in Python itself.

Matlab's $ode15s$ was not set to using BDF.

R's 'adams' and Scilab's 'adams' are similar in that they both use the LSODE code from ODEPACK.

R's and Python's Runge Kutta 4(5) pairs are both implementations of DOPRI5 but they have different source code as Python implements its own. R does not use the DOPRI5.f file. Scilab's 'rkf' is not the same pair as it is using the Shampine and Watts Fehlberg's Runge-Kutta pair, not the Dormand-Prince pair. $ode45$ in Matlab is a Matlab implementation of DOPRI5.

Scilab's 'rk' which is of order 4 and R's 'rk4' are not the same solvers. Scilab's 'rk' is adaptive (error-controlled with Richardson extrapolation for the error estimate) whereas R's 'rk4' is a fixed step-size method.

R's 'Radau' and Python's 'Radau' have different source codes as Python implements its own 'Radau' code while R appears to be called the Fortran code for Radau5.

1.5 Observation on obtaining solution approximations at output points

VI ==================== Include paragraph on interpolation ==================== VI

In this section, we discuss an issue that we encountered with some of the ODE solvers in R and Scilab when it comes to plotting. In an ideal scenario, the user's desired output points should not interfere with the efficiency of the solvers. However, in these two platforms, an old method for outputting is used which makes asking for a lot of output points very inefficient.

Normally, an ODE solver will work as follows: it will have a default initial step-size, will take a step, and will then adjust the step-size based on the error estimate associated with the solution approximation for the given step and then it will use this new step-size to take the next step. This process is repeated until the solver reaches the end of the interval. However, often the users of an ODE solver will require outputs at specific points and these points may lie at points that are internal to the steps. The current state-of-the-art approach to get solution approximations of these output points is to perform a high accuracy interpolation on the given step and to return the value of that interpolant at the required point. The accuracy of the interpolation on new solvers is at least of order p if the numerical ODE solution is of order p.

In R and Scilab, this new method is not used in all the solvers. Instead, R and Scilab will use the output points to dictate the step-size. This issue arises when many output points appear before the steps that would normally be taken by the solver. These solvers will thus use an initial step size or the difference between the start point and the next output point as the initial step-size and step to this next point. The solver will then repeat the process between each consecutive pair of output points. Thus the space between points will limit the maximum step-size that can be taken and will lead to additional function evaluations because the solver needs to pause at each output point. This will lead to a considerable drop in efficiency as we will show, for instance in Tables 9 and 10. These tables show that a problem that can be solved with 150 function evaluations will be solved with 500 function evaluations when there are many outputs. This increase in the number of function evaluations will correspond to increases in CPU time.

This old method of implementing output points also means that the accuracy of the solution depends on the space between the output points. Both in an error-control and a fixed step-size solver, the step-size in this interval will be the maximum of the difference between two output points. Thus, we get the unusual behavior that the accuracy is increased by putting the points closer together and the accuracy is decreased by putting them further apart. We will point out these inconsistencies as they become relevant later in this report. We also note that spacing the points closer together is not a good way to control the accuracy as no researcher will know beforehand how close the points should be.

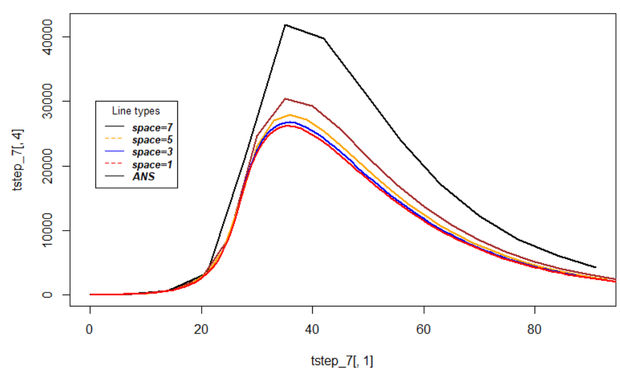

Figure 3: R 'ode45' spacing between points experiment

Figure 3 shows an experiment where we solve an ODE problem using R's 'ode45', which is an implementation of DOPRI5 which has error control but is not using interpolation, to show that it is doing more function evaluation than needed. We set both the absolute and relative tolerance to 0.1 and thus expect low accuracy but very good efficiency. However, the space between the points is still a limiting factor for the step-size and the solution is somewhat accurate although very inefficient. We recorded the number of function evaluations in Table 1 and it can be seen that R is using a lot more function evaluations than are needed to satisfy such coarse tolerances.

Table 1: R DOPRI5 spacing experiment

spacing	nfev
1	572
3	188
5	116
7	80

From Figure 3 and Table 1, we note that we did not ask the solver for an accurate solution and that it is giving us an excessively accurate solution. This excess accuracy comes at a price of around 500 more function evaluations. Accuracy should ideally be completely determined by the tolerance but using this old method for dealing with output points substantially interferes with this ideal. This results in the solver not being allowed to take as big a step as that should be based on the tolerance, and this leads to substantial inefficiency.

We advise users to use interpolation software whenever readily available so that the solvers can run as efficiently as possible. When faster CPU times are required, the researcher can look to see if their chosen solver is using interpolation or if it is taking unnecessary steps. If their chosen solver is using the old method, we advise using a different solver that has an internal interpolant whenever possible or running their solvers with an appropriately spaced out set of points and running an interpolation on the output to obtain a smooth plot of the solution. We also reiterate that the interpolant should be at least of order p if the ODE solver gives a solution of order p so that the interpolation error does not interfere with the solution more than the ODE error.

1.6 Discontinuities and their effects on solvers

The main purpose of this report is to discuss how to model discontinuities and how these affect the process of computing a numerical solution to the model. In this section, we will show what happens when a solver meets a discontinuity and how this leads to erroneous solutions.

We need to understand that all the solvers use numerical methods based on Taylor series, one of the core assumptions of which is that the function and all its relevant derivatives are continuous. If the right-hand side function is discontinuous, this theory no longer holds and the solvers are no longer guaranteed to converge to the actual solution as the step-size goes to 0.

We will see that discontinuities will have huge impacts on the efficiency of the solvers, that some solvers, even with error control, will require an extremely sharp tolerance to step over the discontinuity, and that fixed-step solvers simply cannot solve these problems accurately.

It is important to note that the step that first meets a discontinuity will almost always fail. This is because for the code to step over a discontinuity, the step size needs to be much smaller than the one that is being used before the discontinuity. The codes will thus have to retake the step with smaller step size and as long as the error estimate is not small enough, it will need to continue reducing the step. This leads to high numbers of function evaluations near the discontinuity which will lead to longer CPU times.

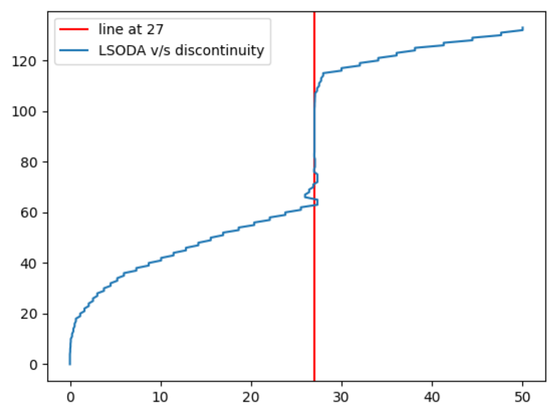

Figure 4: Function evaluations for 'LSODA' when there is a discontinuity at t=27

In Figures 4 and 5, we run 'LSODA' and 'DOP853' from Python on a discontinuous problem where the discontinuity is introduced at time 27 and plot the time at which the i^{th} function evaluation occurs. We thus show the spike in the number of function evaluations at the discontinuity as the solvers repeatedly retake that step with smaller and smaller step-sizes.

Following this discussion, we also recommend epidemiologists carry out a manual discontinuity detection experiment to see if their model has any discontinuity. This trivial experiment is done by collecting at what time the solver made the i^{th} call to the solver. The pseudo-code of which is as shown below:

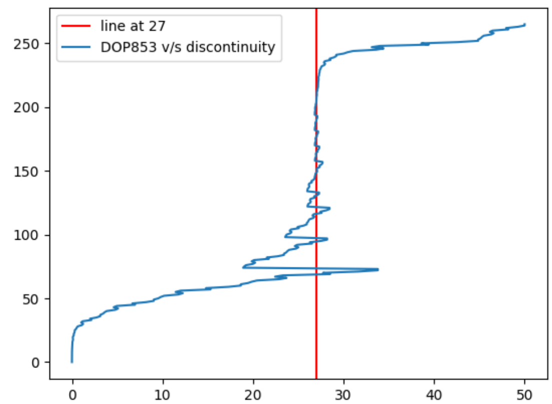

Figure 5: Function evaluations for 'DOP853' when there is a discontinuity at t=27

```
times = []
function_calls = []
count = 0

function model(t, y)
    global times, function_calls, count
    times.append(t)
    function_calls.append(count)
    count += 1

    // code to find the derivatives
    return <derivatives>

plot(times, function_calls)
```

In the experiment outlined in this pseudo-code, we plot the time against the cumulative count of the function calls. An almost straight vertical line on this graph will indicate that the function was called repeatedly at a specific time and thus that the solver repeatedly changed the step-size in this region to step over a discontinuity. Thus the epidemiologist can detect a discontinuity and can perform further tests. In the remainder of this report, we will outline the ways to accurately and efficiently solve problems with such discontinuities.

2 Time dependent discontinuity problem

In the time-dependent discontinuity problem, we change the value of the parameter β from 0.9 to 0.005 at t=27. This introduces a discontinuity in the problem. We will show that this leads to inaccuracies in the solutions computed by the solvers, especially with fixed-step solvers. We then introduce a form of discontinuity handling using cold starts to show an efficient way to solve time-dependent discontinuity problems.

2.1 Naive treatment of Covid-19 time discontinuity models

A naive implementation of the problem is to use an if-statement inside the right-hand side function, $f(t, y)$, to implement the changes in β as measures are implemented. An if-statement makes the function f(t, y) and its derivatives discontinuous. This introduces issues as outlined in Section 1.6.

In pseudo code, this looks like:

```
function model_with_if(t, y)
    $\vdots$
    beta = 0.005
    if t < 27:
        beta = 0.9
    $\vdots$
    return (dSdt, dEdt, dIdt, dRdt)
```

Also, to stay true to a naive treatment, we will always use the default tolerances in this section. Discrepancies across the programming environments that can be due to tolerance issues are investigated in Section 2.3.

2.1.1 Time discontinuity model in R

From Figure 6, we can see that all the methods except 'euler' and 'rk4' are on the same line. This means that the solutions agree to "eyeball" accuracy which typically means that they agree to about two significant digits. 'rk4' is somehow close to the actual solution but 'euler' is completely wrong. We note that all the other methods have error control while these two are fixed step-size solvers.

We also note that 'rk4' is doing better than 'euler' for this specific problem as it has a higher order. But the way it is performing is still better than expected. We show that this is entirely because of the outputting issue discussed in Section 1.5. If we use a bigger step-size, 'rk4' gives results that just are as bad as 'euler'. Figure 7 shows an experiment with 'rk4' used with different step-sizes (space between the output points) plotted against an accurate solution in red. We can see that as soon as we change the step-size for 'rk4', it does not give good results at all. Analyzing the source for 'rk4' and 'euler' shows that these methods select the step size using the output points requested. Spacing out the output points affects the step-size which affects the accuracy of the fixed step-size solver.

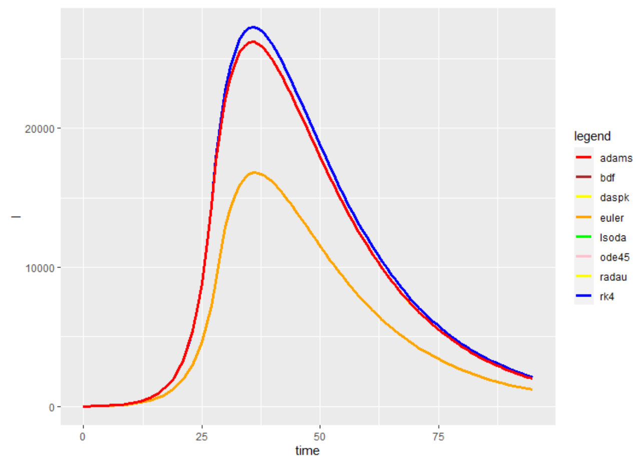

Figure 6: Time Discontinuity model in R

If a user wants to use 'rk4' or 'euler', the user would have to choose a small step-size. However, we cannot know beforehand how small is small enough. Furthermore, there is the issue that a sufficiently small step-size can vary from one part of the domain to another as the problem difficulty changes. A fixed step-size solver will have to choose the smallest step required anywhere in the domain and this can lead to substantial inefficiency. A better solution is to not use fixed step-size solvers. Reputable methods with error control should be preferred as we have shown that these solvers can step over a discontinuity by resizing the step repeatedly, as needed.

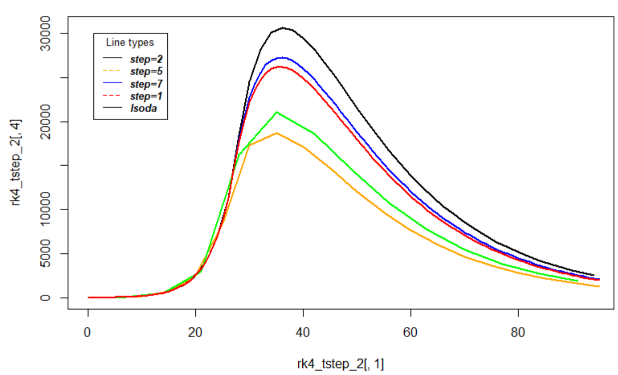

Figure 7: The R version of 'rk4' with several fixed step-sizes

2.1.2 Time discontinuity model in Python

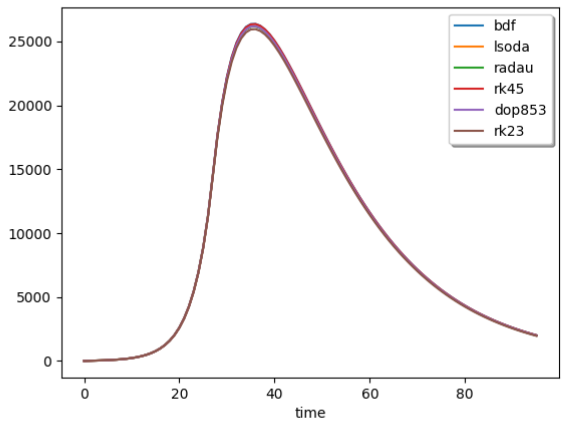

Figure 8: Time Discontinuity model in Python

From Figure 8, we can see that all the methods in Python's *solve_ivp*() work reasonably well. There is some blurring at the peak, indicating some disagreement among the methods, but all the methods provide reasonably accurate results. Python only provides error-controlled packages and thus we can see that error-control is all that is needed to step over this discontinuity. This observation also leads us to another conclusion that a reasonably sharp tolerance with an error-control method is what is required to step over this type of discontinuity. (Recall that all Python methods use a default absolute tolerance of 10^{-6} and a relative tolerance of 10^{-3}.)

2.1.3 Time discontinuity model in Scilab

From Figure 9, in Scilab, all the methods give similar solutions except for 'rkf'. This is interesting as we know that 'rkf' uses error control. This is explained by noting that 'rkf' uses coarser default absolute and relative tolerances. We will show during a tolerance analysis in Section 2.3 that with a sharp enough tolerance, 'rkf' also provides a reasonably accurate solution.

The other methods are all error-controlled and give similar results as expected. We note that all of the other methods have a higher default tolerance than 'rkf' and thus this result is not surprising.

These results also point out that an error control solver with a sharp tolerance can step over this type of discontinuity.

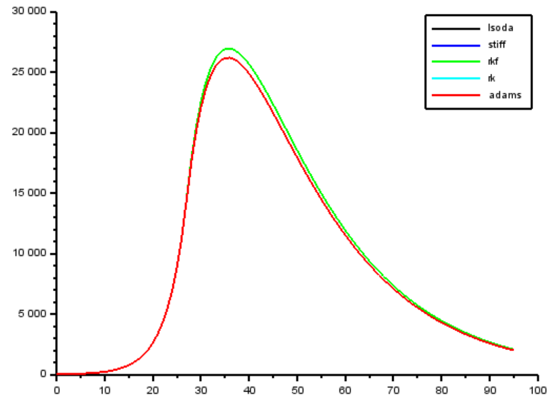

Figure 9: Time discontinuity model in Scilab

2.1.4 Time discontinuity model in Matlab

Figure 10 shows that *ode*45 and *ode*15*s* are not in agreement. This is strange because both are error controlled. We note that the same behavior of *ode*45 is seen in 'rkf' in Scilab but the methods are based on different algorithms. In Matlab, both *ode*45 and *ode*15*s* have the same default tolerances so we rule out that a tolerance difference is the result of this behavior. We will see that *ode*45 can give a similar result to *ode*15*s* answers when the tolerance is sharp enough in Section 2.3.

2.2 Better way to treat discontinuities in the time models

A better way to solve the time-dependent discontinuity problem is to make use of cold starts. This means that we integrate before and after the discontinuity with *separate* calls to the solver. Cold starting a solver at the time-dependent discontinuity improves the accuracy as we will see in this and the next section. It also improves the efficiency as fewer function calls are required since we do not have the spike in function calls due to repeated step-size resizing described in Section 1.6.

A cold start will entail no values from previous computations influencing the new integration. It will also involve using small initial step sizes and for methods of varying order like the 'BDF' and 'Adams', they start anew with the default order which is of order 1.

To solve the time dependent discontinuity problem, we will integrate from time 0 to the time that measures are implemented, t=27, with one call to the solver and then use the solution values at t=27 as the initial values to make another call that will integrate from t=27 to t_f. The pseudo-code is as follows:

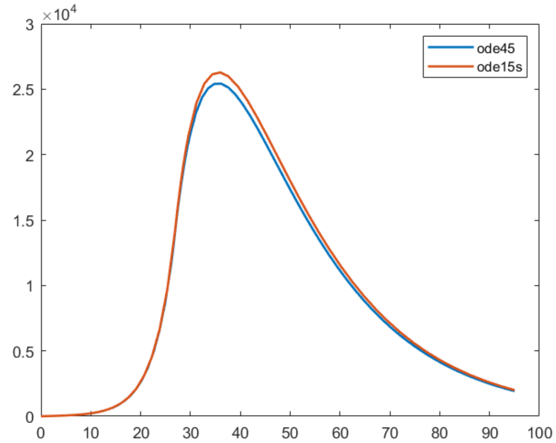

Figure 10: Time discontinuity model in Matlab

```
initial_values = (S0, E0, I0, R0)
tspan_before = [0, 27]
solution_before = ode(intial_values, model_before_measures,
tspan_before)

initial_values_after = extract_last_row(solution_before)
tspan_after = [27, 95]
solution_after = ode(intial_values_after,
model_after_measures, tspan_after)

solution = concatenate(solution_before, solution_after)
```

This technique can be applied by researchers to any problem where it is known when the discontinuity is introduced. This is much better than introducing a time-dependent if-statement into the model.

2.2.1 Solving time dependent discontinuity model in R using a cold start

From Figure 11, we see that the ‘euler’ method still fails even with the cold start discontinuity handling. This is as expected as it has no error control and thus it still suffers from accuracy issues and will require smaller steps to achieve even “eyeball” accuracy.

We see that breaking the problem into two parts makes ’rk4’ perform better. The method has a higher-order, meaning that it does not need as small an initial step size as ’euler’ to solve the two continuous problems but this exceptionally good performance is still unexpected. We will show in Figure 12 that this is only

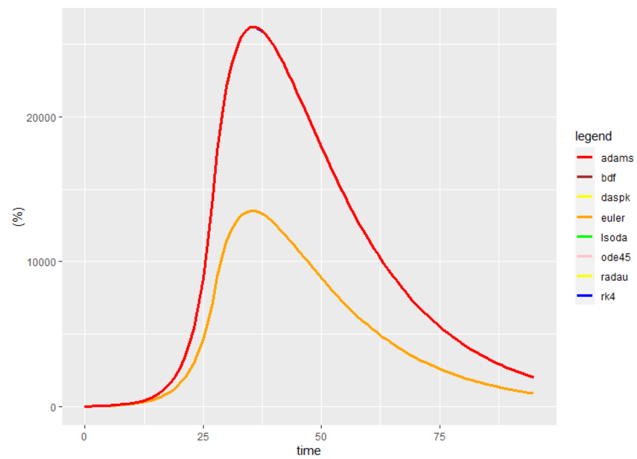

Figure 11: Solving time dependent discontinuity model in R

due to the use of very small step size and the performance of ‘rk4’ is associated with the method of outputting points as described in Section 1.5.

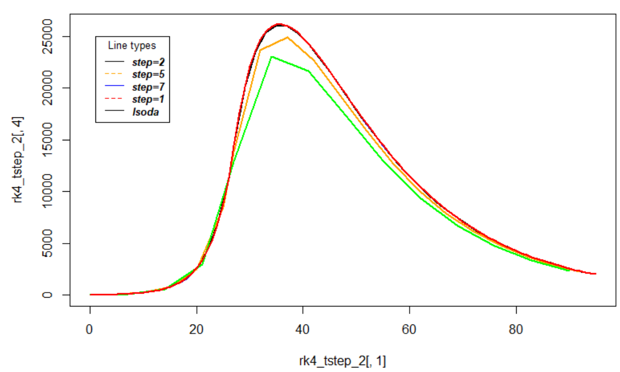

Figure 12: The R version of ‘rk4’ with bigger step-size and with discontinuity handling

Thus our recommendation to avoid fixed step size solvers still holds since researchers will typically know how small the step size needs to be to obtain sufficient accuracy.

We also note again, that all the error-controlled solvers perform well. We will see, from the efficiency data, that using cold starts is more efficient. Using cold starts, the error control solvers do not have to step over a discontinuity and we will not have the rise in the number of function evaluations as we discussed in

1.6. Table 2 shows that discontinuity handling reduces the number of function evaluations.

Table 2: R Time Discontinuity problem efficiency data

method	no discontinuity handling	with discontinuity handling
euler	96	97
rk4	381	382
lsoda	332	272
ode45	735	599
radau	679	585
bdf	423	263
adams	210	176
daspk	517	521

Our analysis of the efficiency data in Table 2 starts by noting that the non-error controlled solvers in the 'euler' and 'rk4' methods have the same number of function evaluations, the additional one being due to integrating twice at time 27. This indicates that they are just stepping from output point to output point using the same fixed step-size both with and without the discontinuity handling.

Next, we note significant decreases in the number of function evaluations for all the remaining solvers except 'daspk'. These reductions in the number of function evaluations will have a significant impact on the CPU time for the difficult problem. This is entirely explained in 1.6 where the error-controlled solvers have to repeatedly resize the step-size as they encounter the discontinuity.

VI ======================= Finally, we explain the almost constant value of dapsk's number of function evaluations through the fact that it may have some inherent discontinuity handling, and thus even without the treatment we did, it could still detect discontinuity and integrate as usual. IT COULD ALSO BE BECAUSE daspk DEPENDS ON THE SPACING BETWEEN THE POINTS. ======================= VI

In Section 2.3, we will see that this discontinuity handling also allows us to use coarser tolerances, which suppress the efficiency of the computation.

2.2.2 Solving time dependent discontinuity model in Python using a cold start

The Python solvers did not have significant accuracy issues even without discontinuity handling. This is because all the available methods use error control and the default tolerances are sharp enough. From Figure 13, we can see that the Python solvers again give sufficiently accurate results. Furthermore, the slight blurring at the peak has disappeared indicating that there is an even better agreement among the solvers. The addition of discontinuity handling will

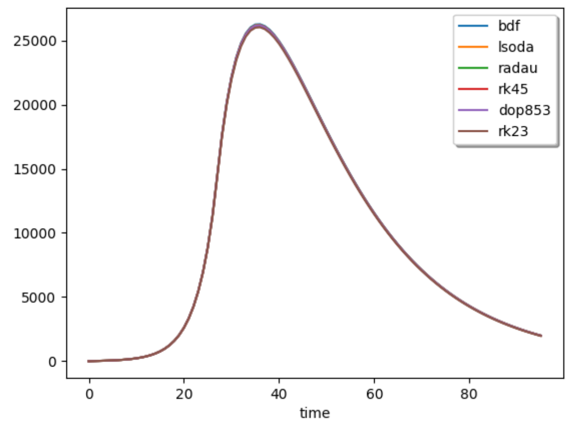

Figure 13: Solving time dependent discontinuity model in Python

also drastically reduce the number of function evaluations. This can be seen in Table 3.

Table 3: Python Time Discontinuity problem efficiency data

method	no discontinuity handling	with discontinuity handling
lsoda	162	124
rk45	134	130
bdf	202	146
radau	336	220
dop853	329	181
rk23	152	127

We note that we are not using *dense_output* here. However, the Python solvers do not seem to allow the space between the output points to affect the accuracy. They appear to be using some form of local interpolation within each step where output is required.

From Table 3, we see that across the board, the methods take fewer function evaluations. There are some huge changes for 'BDF', 'DOP853' and 'Radau'. There are slight decreases in 'LSODA' and 'RK23' and only a very small decrease in 'RK45'. In Section 2.3, we will see that this discontinuity handling also allows us to use coarser tolerances.

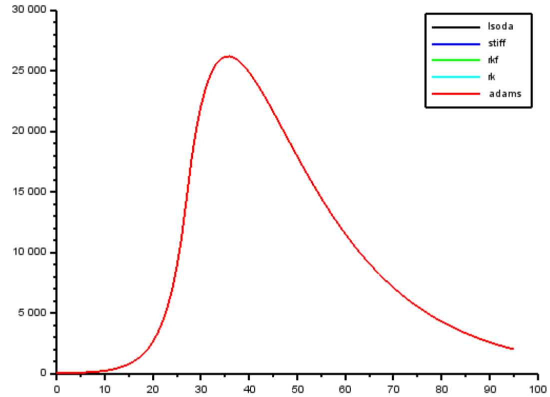

Figure 14: Solving time dependent discontinuity model in Scilab

2.2.3 Solving time dependent discontinuity model in Scilab using a cold start

We can see from Figure 14 that all the methods show good agreement and thus the time-dependent discontinuity model is being solved to a reasonable accuracy. The 'rkf' method is also giving reasonable results. This is despite 'rkf' having a coarser default tolerance. In Section 2.3, we will see that this discontinuity handling also allows us to use coarser tolerances for all solvers and thus explains why the default tolerance used by 'rkf' is sufficient to allow it to solve the problem reasonably well.

The addition of discontinuity handling will also drastically reduce the number of function evaluations as seen in Table 4.

Table 4: Scilab Time Discontinuity problem efficiency data

method	no discontinuity handling	with discontinuity handling
lsoda	346	292
stiff	531	362
rkf	589	590
rk	1649	1473
adams	304	221

From Table 4, we see that all the methods use fewer function evaluations. We see substantial decreases in the number of function evaluations for 'lsoda', 'stiff', 'rk' and 'adams'.

The odd function value counts for 'rkf' (the number of function evaluations does not decrease) occurs because 'rkf' is using the method for outputting points as outlined in Section 1.5. The results, when we space out the output points more, are 335 without discontinuity handling and 292 with discontinuity handling. We see that the number of function evaluations increased.

We note that the high number of function evaluations in 'rk' with and without discontinuity handling is because it is using Richardson extrapolation to get an error estimate.

2.2.4 Solving time dependent discontinuity model in Matlab using a cold start

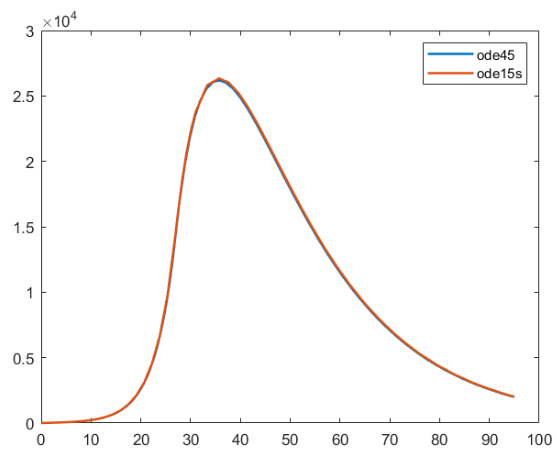

Figure 15: Solving Time discontinuity model in Matlab

From Figure 15 we can see that both solvers give similar solutions. We remember that with an if-statement inside the function $f(t, y)$ that the two solvers gave different solutions. As we will show in Section 2.3, the discontinuity handling allows us to use a coarser tolerance and thus allows $ode45$ to give a reasonably accurate result.

We also show in Table 5 that discontinuity handling allows us to use fewer function evaluations.

Table 5: Matlab Time Discontinuity problem efficiency data

method	no discontinuity handling	with discontinuity handling
ode45	175	164
ode15s	144	113

From Table 5, $ode45$ uses 11 less function evaluations while $ode15s$ uses 31 less function evaluations which translates to faster CPU times.

2.3 Efficiency data and tolerance study for the time discontinuous problem

It is not uncommon for researchers to use an ODE solver in a loop or within an optimization algorithm so that they can study models with different problem-dependent parameters. In so doing, some may be tempted to coarsen the tolerances whenever the computation that they are performing is taking too long. In this section, we investigate how coarse we can set the tolerance while still obtaining reasonably accurate results for the time-dependent discontinuity model.

We investigate 'lsoda' across R, Python, and Scilab as they all appear to use the same source code. We use this experiment to show that discontinuity handling allows us to use coarser tolerances.

We will also investigate 'rkf' in Scilab as it has a smaller default tolerance than the other Scilab solvers and *ode*45 in Matlab, both of which failed to solve the time-dependent discontinuity model. We will prove that it can solve the problem without discontinuity handling only at sharper tolerances than the default tolerances. We also investigate solvers based on Runge-Kutta pairs of the same order as the pair used in 'rkf' and *ode*45 in the other programming environments; R and Python have a version of DOPRI5 but do not share the same source code. The DOPRI5 in Python is a Python implementation and the one in R is an interface to a Fortran implementation. We also note that R is not using DOPRI5.f but another Fortran implementation of DOPRI5. *ode*45 in Matlab uses DOPRI5 but it is implemented in the Matlab programming language.

2.3.1 Comparing LSODA across platforms for time discontinuous problem

In this section, we run R's LSODA solver with multiple tolerances with and without discontinuity handling. We will set both the relative and absolute tolerances to particular values and see how coarse we can keep the tolerance while still obtaining reasonably accurate results. We also look at efficiency data to observe decreases in the number of function evaluations, which will lead to significant decreases in computation times.

Time discontinuity LSODA tolerance study in R From Figures 16 and 17, we can see that the addition of discontinuity handling allows the solver to use coarser tolerances and still get a reasonable result; we need a tolerance of 10^{-3} and sharper tolerances without discontinuity handling but can use a tolerance of 10^{-2} and sharper with it. This supports the observation that the use of discontinuity handling when solving a discontinuous problem will improve the accuracy of the solution. Also, using coarser tolerances gives us more efficiency, as we will see in Table 6. This allows researchers to coarsen the tolerance employed by a solver when the solver is running too slowly.

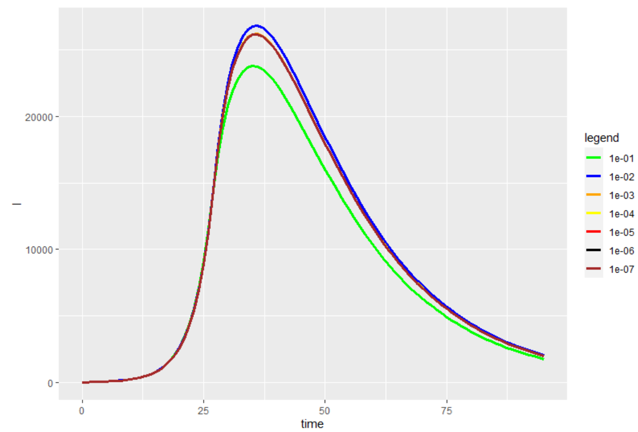

Figure 16: Time discontinuity model tolerance study on the R version of LSODA without a cold start

Table 6: R LSODA time Discontinuity tolerance study

tolerance	no discontinuity handling	with discontinuity handling
1e-01	197	200
1e-02	214	206
1e-03	264	212
1e-04	264	224
1e-05	317	244
1e-06	332	272
1e-07	393	298

From Table 6, we see that for the coarser tolerances, the number of function evaluations is roughly the same. But with sharper tolerances, a lot more function evaluations are required and thus if we had a user-provided function that was expensive to evaluate, we will see clear reductions in computation times.

A similar number of function evaluations for the coarser tolerances should not distract from the fact that the solver without discontinuity handling at these tolerances gives results that are not as accurate as the results obtained using the solver with discontinuity handling. The small differences of 3 function evaluations for the 0.1 tolerance case and 8 function evaluations in the 0.01 case do not excuse the fact that the solutions are wrong.

Time discontinuity LSODA tolerance study in Python In this section, we run the Python version of the LSODA solver with multiple tolerances with and without discontinuity handling. We note that the Python solvers were giving sufficiently accurate results in both cases apart from some small disagree-

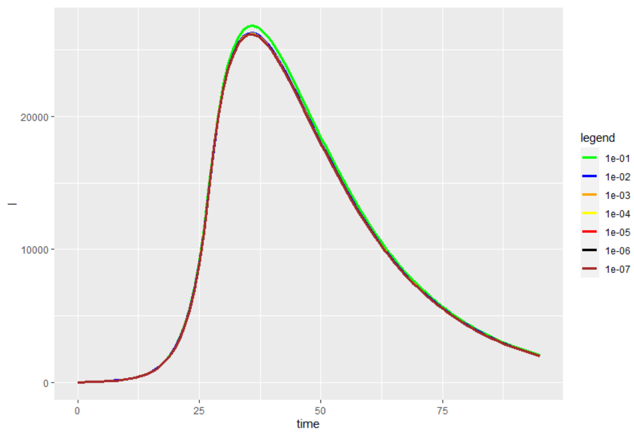

Figure 17: Time discontinuity model tolerance study on the R version of LSODA with a cold start

ments in the case where no discontinuity handling is employed but we will see how coarse we can choose the tolerance while still obtaining reasonably accurate results. We set both the relative and absolute tolerances to particular values. We also look at efficiency data to see the decreases in the number of function evaluations.

From Figures 19 and 18, we see that the use of the discontinuity handling lets us use a coarser tolerance since a tolerance of 10^{-2} was enough to get a reasonably accurate result with the discontinuity handling whereas a tolerance of 10^{-3} was needed otherwise. This tells us that using discontinuity handling will improve our results for a more complex time-dependent discontinuity problem.

In turn, the use of coarser tolerances give us more efficiency. (See Table 7.)

We also note that the results using LSODA in Python and R are very similar which stems from the fact that they are using the same source code.

Table 7: Python LSODA Time Discontinuity tolerance study

tolerance	no discontinuity handling	with discontinuity handling
0.1	79	86
0.01	98	93
0.001	156	116
0.0001	185	146
1e-05	259	186
1e-06	283	228
1e-07	361	272

Again, in Table 7, we see that that at coarse tolerances, the number of

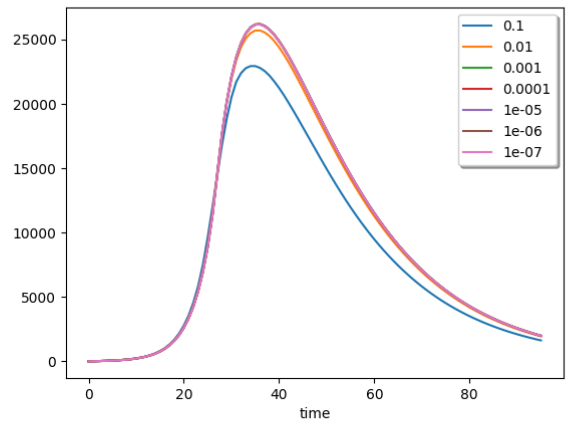

Figure 18: Time discontinuity model tolerance study on the Python version of LSODA without a cold start

function evaluations is roughly the same. This similar number of function evaluations does not excuse the fact that the coarser tolerances are giving erroneous values when discontinuity handling is not employed.

At sharper tolerances, where the comparison is fair, the number of function evaluations is much smaller with discontinuity handling than without; we make 40 fewer function evaluations at 0.001 and 0.0001 but we do many fewer function evaluations for sharper tolerances. We note that if the function for the evaluation of the right-hand side of the ODE was more time-consuming, this reduced number of function evaluations will cause a decrease in the CPU times. This reduced number of function evaluations stems from the facts discussed in Section 1.6.

Time discontinuity LSODA tolerance study in Scilab In this section, we run the Scilab version of the LSODA solver with multiple tolerances with and without discontinuity handling. We will set both the relative and absolute tolerances to particular values and see how coarse we can keep the tolerance while still getting reasonably accurate results.

From Figures 20 and 21 we can see that for tolerances from 10^{-1} to 10^{-4}, the Scilab version of LSODA without discontinuity handling fails but we are able to use a tolerance as coarse as 10^{-2} with the discontinuity handling.

It is interesting to see how far off the solution without discontinuity handling is at a tolerance of 10^{-1}. We also note that this behavior is different from the R and the Python version LSODA but this may be due to the way Scilab handles the tolerances.

Again, in Table 8, we see that the number of function evaluations is roughly the same at coarser tolerances but that at sharp tolerances, where both types

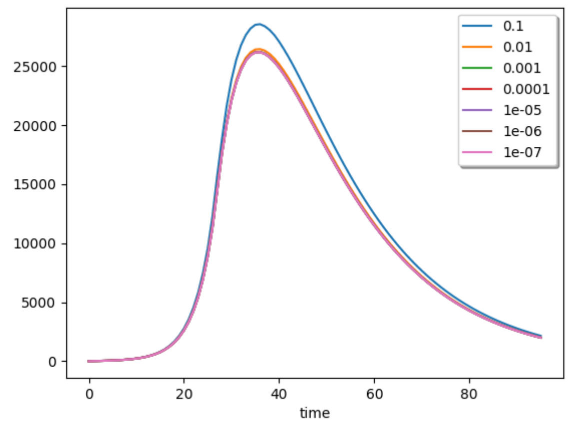

Figure 19: Time discontinuity model tolerance study on the Python version of LSODA with a cold start

Table 8: Scilab LSODA Time Discontinuity tolerance study

tolerance	no discontinuity handling	with discontinuity handling
0.1	80	82
0.01	98	92
0.001	156	116
1e-4	185	146
1e-5	255	186
1e-6	280	228
1e-7	361	272

of computations give reasonably accurate solutions and thus allow for a fair comparison, the solver with discontinuity handling performs better than the solver without discontinuity handling. We can use up to 90 fewer function evaluations through the use of discontinuity handling.

2.3.2 Comparing solvers based on Runge-Kutta pairs across platforms for the time discontinuous problem

Time discontinuity tolerance study on the R version of DOPRI5 In this section, we use the R version of DOPRI5, which is the 'ode45' method of the *ode*() function, with multiple tolerances with and without discontinuity handling. We will set both the relative and absolute tolerances to particular values and see how coarse we can choose the tolerance while still getting reasonably accurate results. We also look at efficiency data to see the decreases in

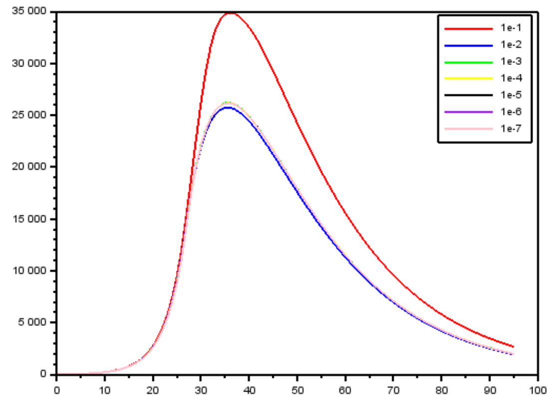

Figure 20: Time discontinuity model tolerance study on the Scilab version of lsoda without a cold start

the number of function evaluations when discontinuity handling is employed.

From Figures 22 and 23, the addition of discontinuity handling lets us use a coarser tolerance and still get a reasonably accurate answer. Without discontinuity handling, we had to use 10^{-4} for both the absolute and relative tolerance but with discontinuity handling, we seem to be able to use 10^{-1}.

However, as we will see in the Python version of DOPRI5, the results from Figures 22 and 23 are suspicious and stem from the fact that R is not using interpolation to produce the results. It is using the old method that depends on the selected output points which affects efficiency and accuracy as discussed in Section 1.5.

Table 9: R dopri5 Time Discontinuity tolerance study

tolerance	no discontinuity handling	with discontinuity handling
1e-01	572	574
1e-02	572	574
1e-03	572	574
1e-04	612	574
1e-05	692	587
1e-06	735	599
1e-07	926	702

Table 9 also confirms our suspicions since, at coarser tolerances, 10^{-1} to 10^{-3}, the number of function evaluations does not change at all. This indicates that something else, not the tolerance nor the discontinuity, is the limiting factor

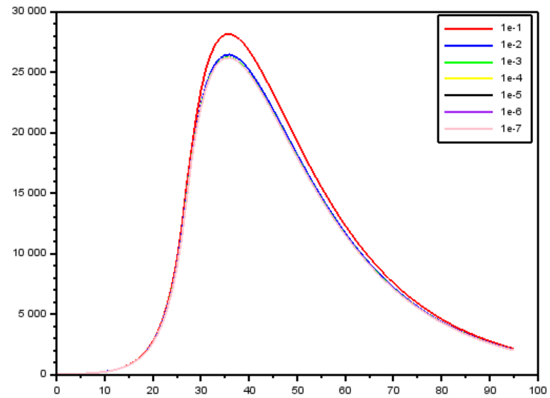

Figure 21: Time discontinuity model tolerance study on the Scilab version of lsoda with a cold start

for the number of function evaluations and that this other factor requires around 572 or 574 function evaluations.

We suspect that the R DOPRI5 version is not using interpolation or some other dense output technique to produce its solutions and that it is integrating using the output points to determine the step-size. That is, even though DOPRI5 is an algorithm that does not have a fixed step-size, R is forcing it to step from one output point to the next, and thus our set of sampling points is a limiting factor. We then do the following experiment where we give R a smaller set of output points with the points are further spaced out from each other and see what happens.

From Figures 24 and 25, we can now see a more drastic change in the solution from the solvers when the output points are further spaced out. Also, see Table 10 where we will see that the number of function evaluations actually changes with the tolerance.

Using these two figures, we also see that discontinuity handling is allowing us to use coarser tolerances. We can use even a tolerance of 10^{-1} with discontinuity handling while getting a reasonably accurate result, whereas, without discontinuity handling, we need to use a tolerance of 10^{-3} or sharper tolerances to get a reasonably accurate answer.

Our analysis of Table 10 begins by noting that the set of output points is no longer a limiting factor. We can see the number of function evaluations change with the tolerance now and this indicates that the tolerance is affecting the step-size. This confirms our suspicions that the R implementation of DOPRI5 is not using a dense output mode or some form of interpolation. Instead, it takes steps to and stops at every required output point. This is an inefficient way of obtaining solution values at specified output points.

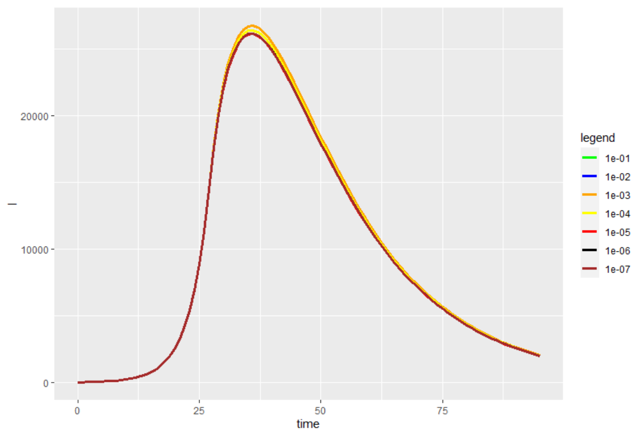

Figure 22: Time Discontinuity model tolerance study on the R version of DOPRI5 without discontinuity handling

Table 10: R DOPRI5 Time Discontinuity tolerance study with spaced output points

tolerance	no discontinuity handling	with discontinuity handling
1e-01	116	112
1e-02	142	125
1e-03	168	131
1e-04	246	162
1e-05	352	235
1e-06	614	349
1e-07	796	542

Regarding the accuracy of the solver as we coarsen the tolerance we can see from Figures 24 and 25 that even at a tolerance of 10^{-1}, the solver with the discontinuity handling is still able to produce reasonably accurate solutions whereas it requires a tolerance of 10^{-3} for the solver without the discontinuity handling.

The new table, Table 10, does offer some more insights. Again we can see that at coarser tolerances, the decrease in the number of function evaluations is small but as the tolerance is sharpened, the number of function evaluations decreases significantly. The relatively similar number of function evaluations at the coarser tolerances does not excuse the fact that the solver without discontinuity handling is not getting a reasonably accurate answer.

Time discontinuity model tolerance study on Python's version of DOPRI5 In this section, we run Python's version of DOPRI5, which is aliased

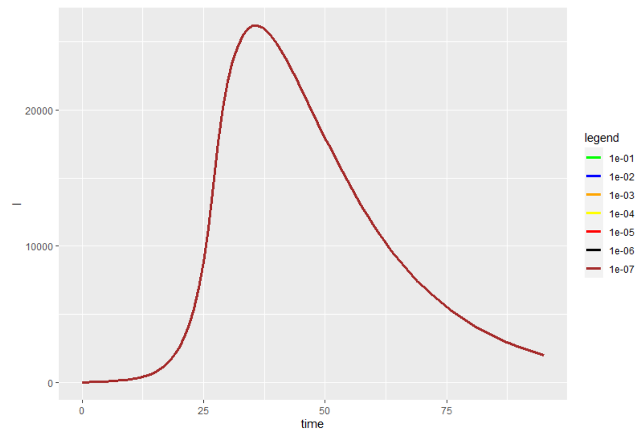

Figure 23: Time Discontinuity model tolerance study on the R version of DOPRI5 with discontinuity handling

under 'RK45' from the $solver\_ivp()$ function, with multiple tolerances with and without discontinuity handling. We will set both the relative and absolute tolerances to particular values and see how coarse we can choose the tolerance while still obtaining reasonable accurate results. We also look at efficiency data to see the decreases in the number of function evaluations.

From Figures 27 and 26, we can see clear differences at the different tolerance values. This is in contrast with the first tolerance study on the R version of DOPRI5. From studying Python's *solve_ivp* interface and source code, we note that Python is using dense output/interpolation. We can explain the R version of DOPRI5 performance entirely because it does not use interpolation by default but instead stops at every output point.

We then compare the Python version of DOPRI5 with and without discontinuity handling. We can see that the use of discontinuity handling allows us to use coarser tolerances in Python while obtaining reasonably accurate results. We see that we need a tolerance of 10^{-5} and sharper to get reasonably accurate solutions without discontinuity handling while tolerance of 10^{-2} is small enough when discontinuity handling is employed. We will also see in Table 11 that the solver with discontinuity handling is much more efficient.

From Table 11, we see that at coarser tolerances, the number of function evaluations is lower with the discontinuity handling than without discontinuity handling. We should also point out that in Python, DOPRI5 at coarse tolerances gives very erroneous results and these do not excuse the small gain in efficiency.

At sharper tolerances where we get reasonably accurate results both with and without discontinuity handling, and thus a fair comparison can be done, we can see that the code with discontinuity handling performs much better. At a tolerance of 10^{-7}, the drop in the number of function evaluations is very significant and would lead to much faster execution times, whereas for a tolerance

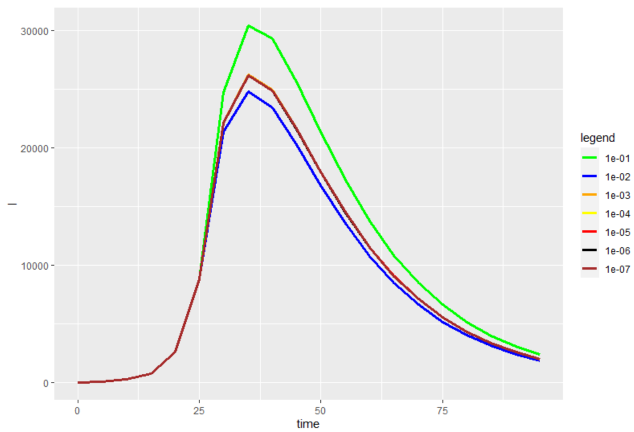

Figure 24: Time Discontinuity model tolerance study on the R version of DOPRI5 without discontinuity handling and output points more spaced out

Table 11: Python DOPRI5 Time Discontinuity tolerance study

tolerance	no discontinuity handling	with discontinuity handling
0.1	68	70
0.01	86	88
0.001	146	124
0.0001	224	172
1e-05	326	250
1e-06	488	370
1e-07	752	568

of 10^{-5} or sharper, the decrease in the number of function evaluations is 75 or more.

Time discontinuity model tolerance study on the Scilab version of RKF45 In this section, we run the Scilab version of RKF45 aliased as 'rkf' in the *ode*() function with different tolerances. We note that the default tolerance for the Scilab 'rkf' function was not enough to solve the problem to reasonable accuracy without discontinuity handling but using cold starts did solve the problem even with that default tolerance.

By running 'rkf' at various tolerances, we will show that it can also compute with reasonably accurate solutions at sharper tolerances without discontinuity handling. Thus the anomaly we saw in Section 2.1 occurred entirely because the solver has a coarser default tolerance in Scilab than the other methods.

We will also see that using discontinuity handling lets us use fewer function evaluations which, given a more complex problem, will result in a significant

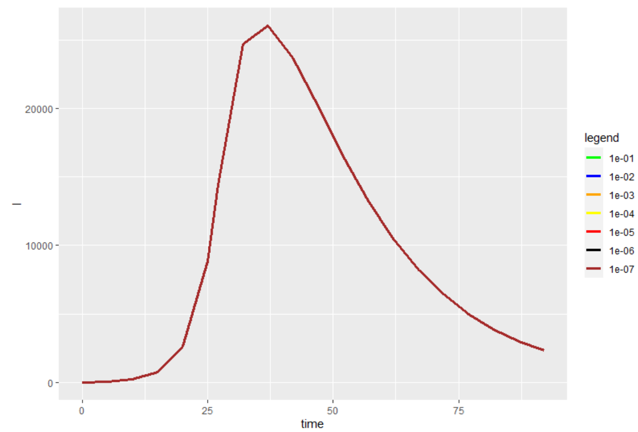

Figure 25: Time Discontinuity model tolerance study on the R version of DOPRI5 with discontinuity handling and output points more spaced out

improvement in computation times.

We see from Figure 28 that using 10^{-4} for both the absolute and the relative tolerance gives reasonably accurate answers and that anything coarser does not work. We then remember that the relative tolerance defaults to 10^{-3} and the absolute tolerance defaults to 10^{-4} for 'rkf' which is slightly coarser than what is needed to get a reasonably accurate solution.

Figure 29 is also interesting as it seems to indicate that a tolerance of 10^{-1} is enough to get the correct solution with discontinuity handling. This is surprising but consistent with our observations for R and Python Runge-Kutta pairs.

Table 12: Scilab RKF45 Time Discontinuity tolerance study

tolerance	no discontinuity handling	with discontinuity handling
0.1	577	584
0.01	577	584
0.001	583	584
1e-4	641	590
1e-5	674	608
1e-6	847	764
1e-7	924	830

We can see from Table 12 that Scilab's 'rkf' is not using interpolation. We can say this because even at extremely low tolerances, it is still using the same number of function evaluations. There is also no difference with and without discontinuity handling. We also note that the tolerance did not change the number of function evaluations and thus something else is regulating the number

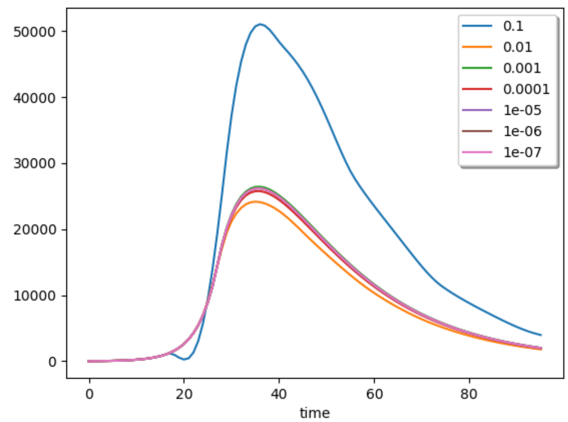

Figure 26: Time Discontinuity model tolerance study on the Python version of DOPRI5 without discontinuity handling

of function evaluations. Doing the same experiment with the points further spaced out shows us that it was the spacing of the output points that was causing the issue.

We start by replicating the experiments in the previous sections.

Figures 30 and 31 shows a clearer distinction in why discontinuity handling is important. We can see that without it we need a tolerance of 10^{-3} to get reasonably accurate results but with the discontinuity handling, we can use a tolerance of 10^{-1}. We note that the use of such a coarse tolerance may mean that we still did not have the output points spaced out enough but the impact on the number of function evaluations, shown in Table 13, is clear.

Table 13: Scilab RKF45 Spaced Out Time Discontinuity tolerance study

tolerance	no discontinuity handling	with discontinuity handling
0.1	133	134
0.01	166	152
0.001	208	176
1e-4	322	254
1e-5	417	338
1e-6	606	482
1e-7	864	704

Table 13 shows how the number of function evaluations with discontinuity handling is smaller. We also note that at coarse tolerance the number of function evaluations is similar but that at those tolerances, the code without discontinuity

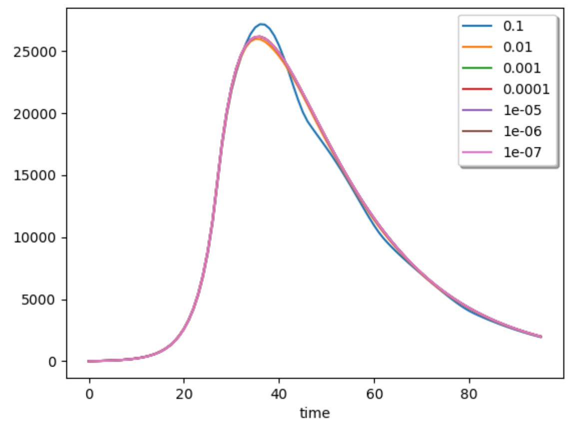

Figure 27: Time Discontinuity model tolerance study on the Python version of DOPRI5 with discontinuity handling

handling was not obtaining reasonably accurate results. We can thus conclude that using discontinuity handling lets us use coarser tolerances and leads to a smaller number of function evaluations while improving accuracy.

Time discontinuity model tolerance study on the Matlab version of DOPRI5 We perform the same experiment using $ode45$ in Matlab. We set both the absolute and relative tolerance to a particular tolerance and we see how the solvers perform. We remember that the default tolerance $ode45$ did not give a reasonably accurate solution. We note that it did not have a smaller default tolerance than $ode15s$. In this section, we show that with a sharper tolerance, $ode45$ is also capable of solving the problem without discontinuity handling but we will see that it is more efficient with discontinuity handling. Discontinuity handling will, again, allow us to use coarser tolerances.

We first note from Figure 32 that at sufficiently sharp tolerances, we can get a reasonably accurate answer without discontinuity handling when the default tolerances did not give a reasonably accurate solution.

From Figures 32 and 33 we see that discontinuity handling allows us to use coarser tolerances while getting a reasonably accurate answer. We note that we could use a tolerance of 10^{-1} with discontinuity handling but we had to use a tolerance of 10^{-3} to get a reasonable solution. We will also see that discontinuity handling allows the solver to use fewer function evaluations in Table 14.

Table 14 show that at coarser tolerances the solver without discontinuity handling use fewer function evaluations. However, at these tolerances, the solver did not give a reasonably accurate solution. At the shaper tolerance, where the solver without discontinuity handling gives a reasonably accurate solution, the number of function evaluations for the solver with discontinuity handling is

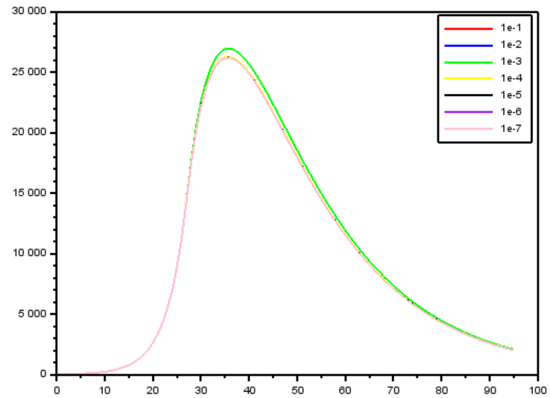

Figure 28: Time discontinuity model tolerance study on the Scilab version of RKF45 without discontinuity handling

Table 14: Matlab's DOPRI5 Time Discontinuity tolerance study

tolerance	no discontinuity handling	with discontinuity handling
0.1	85	146
0.01	121	146
0.001	169	158
0.0001	229	200
1e-05	355	302
1e-06	547	446
1e-07	823	692

better.

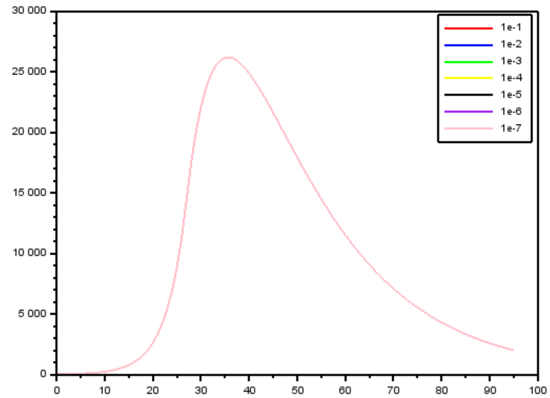

Figure 29: Time discontinuity model tolerance study on the Scilab version of RKF45 with discontinuity handling

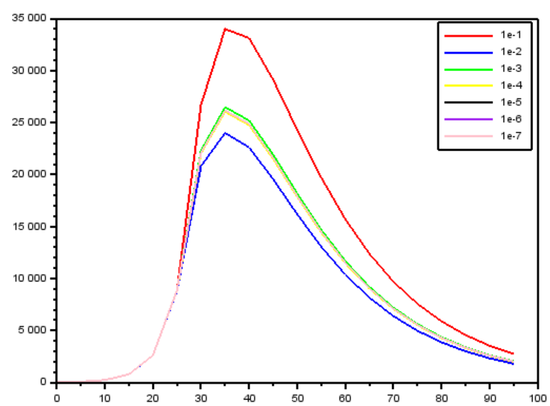

Figure 30: Time discontinuity model tolerance study on the Scilab version of RKF45 without discontinuity handling

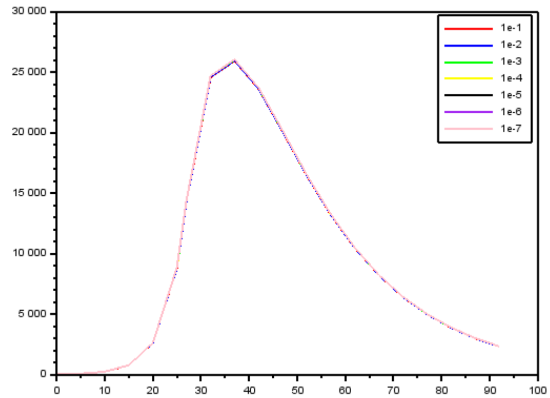

Figure 31: Time discontinuity model tolerance study on the Scilab version of RKF45 with discontinuity handling

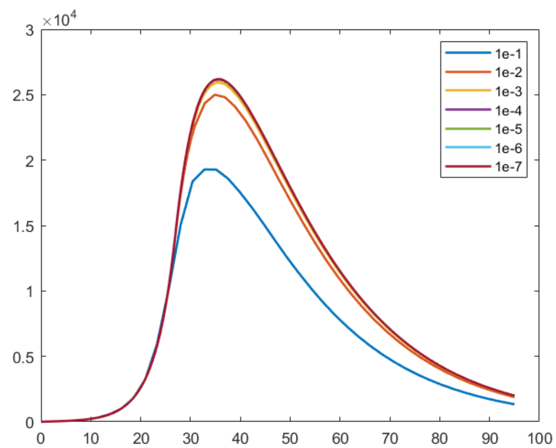

Figure 32: Time discontinuity model tolerance study on the Matlab version of DOPRI5 without discontinuity handling

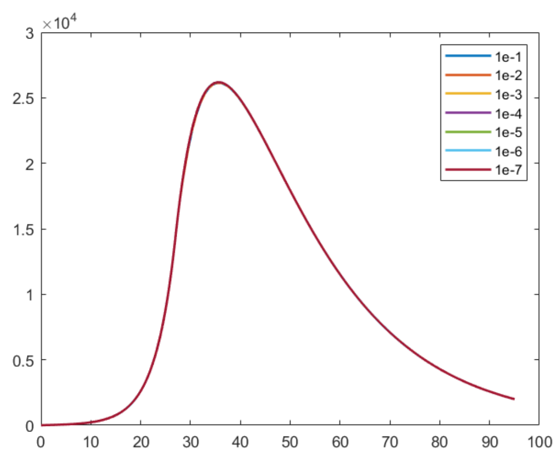

Figure 33: Time discontinuity model tolerance study on the Matlab version of DOPRI5 with discontinuity handling

3 State dependent discontinuity problem

In this section, we model the state-dependent discontinuity problem. We start by noting that this problem cannot be solved with the form of discontinuity handling used in the previous problem as we do not know when the discontinuity arises. Also, this problem will be harder than the time-dependent discontinuity problem as the parameter β will be changed more than once as we attempt a long-term forecast to model the waves of the pandemic.

As in Section 2, changes in the modelling parameter β introduce discontinuities in the function $f(t, y)$ and thus some solvers will thrash when trying to solve the problem (as described in 1.6). But the parameter is changed more than once, the problem has more discontinuities than the previous problem. We will show that the presence of several discontinuities while not being able to do time-dependent cold starts makes the problem hard enough that all the ODE solvers we considered, even at very sharp tolerances, will not be able to solve the problem with reasonable accuracy.

The problem uses the state variable, E, which is the number of Exposed people, to determine when to change the parameter β. When the number of exposed people is greater than 25000, measures will be introduced and thus β will change from 0.9 to 0.005. When the number of exposed people drops back to 10000, the measures will be relaxed and β is set to 0.9. We run this model over a longer time period toggling the parameter β back and forth to model the waves of the pandemic. This scenario corresponds to the case of an unvaccinated population where the only means of controlling the spread of the virus is through measures such as social isolation and so on. The ability of the virus to infect people is not diminished as time progresses, and when measures to stop the spread of the virus are removed, the infection rate of the virus returns to its original value.

We start with a naive treatment of the problem with if-statements inside the function that defines the right-hand side of the ODE system. We proceed to show how the problem cannot be solved this way even at sharp tolerances and finally, we will introduce a way to efficiently and accurately solve the problem using event detection.

3.1 Naive treatment of Covid-19 state dependent discontinuity model

The naive treatment of this problem is to use global variables for tracking when measures are implemented or not and to toggle these global variables as we reach the required thresholds. Global variables are needed because we need to know if the number of Exposed people is going up or down to know whether we need to check for the maximum or the minimum threshold. We then have an if-statement that will choose the value of parameter β based on whether measures are being implemented. The pseudo-code for this problem thus looks as such:

```
measures_implemented = False
direction = "up"
function model_with_if(_, y):
    $\vdot$
    global measures_implemented, direction
    if (direction == "up"):
        if (E > 25000):
            measures_implemented = True
            direction = "down"
    else:
        if (E < 10000):
            measures_implemented = False
            direction = "up"

    if measures_implemented:
        beta = 0.005
    else:
        beta = 0.9
    $\vdot$
    return (dSdt, dEdt, dIdt, dRdt)
```

3.1.1 State dependent discontinuity model in R

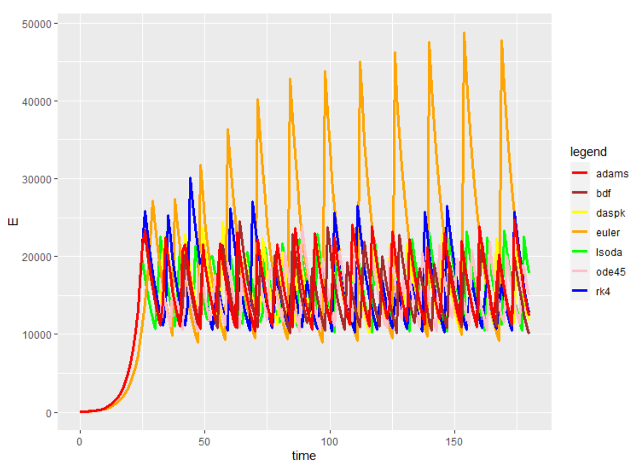

Figure 34: State dependent discontinuity model in R

Figure 34 shows how difficult this problem is with a naive treatment. We note that none of the solutions are aligned and that none of the solvers got the accurate solution (described in Section 3.4) as none of the computed solutions cleanly oscillate between 10000 and 25000 with clear peaks and troughs.

We note that all the solvers, even the error-controlled ones, did not issue a warning about the integration and thus researchers may be tempted to think that their code has solved the problem to a reasonable accuracy. Having no warning also tells us that the error estimation and error control algorithms applied in all the solvers did not detect anything abnormal; the solvers return with an indication that to the given tolerances, that the provided solutions are accurate.

As we are modeling E, we expect that each graph should go from 25000 to 10000 and back to 25000 repeatedly but none of these graphs do so in the required pattern. We would also expect the solvers with error control to repeatedly reduce the step-size to satisfy the tolerance and compute solutions that align with each other but Figure 34 shows that this is not the case.

We also note that the result for 'euler' is especially poor as it reaches a maximum of 40000. This is again as expected as 'euler' has no error control; 'rk4', the other fixed step-size method, is also performing terribly as we see the solution it computes reach approximately 30000 in its third peak. This is all even though the space between the output points is as small as it was when performing the first experiment. Because of this, we will not run any spacing of output points experiments in this section. The step-size for these fixed-step solvers is not small enough and further step-size reductions are needed.

Another important fact to note is how poorly 'radau', as shown in Figure 35, is performing. This is not a problem in the R programming environment as similar results will be seen in Python in the next section and in the Fortran code in Section 4. It grows exponentially even after the parameter β should be switched to 0.005.

We note that in the way that the problem is coded, with the parameter β depending on the state that β is at 0.005 after the first discontinuity is met but that 'radau' is simply ignoring it. We perform an analysis with the Fortran code in 4 to show that β is indeed 0.005 while this exponential growth is happening.

We then proceed to show that sharp tolerances are not enough to solve this problem as was the case for the time-dependent discontinuity problem. We repeated the experiment at the sharpest tolerance usable before some of the solvers failed. This was at 10^{-13} in the R environment. We set both the absolute and relative tolerance to that value and the results are shown in Figure 36.

We can see from Figure 36 that the situation has only marginally improved. None of the solvers give solutions that are in agreement and none of them cleanly oscillate between 10000 and 25000. We note that the error-controlled solvers are following the correct pattern and that until about time 20-30, some of them give solutions that are in agreement, showing that sharp tolerance error-control can step over one state-dependent discontinuity. (See the comparison against the final solution in Section 3.1.5 to see that even this sharp tolerance solution is not accurate enough.)

The fixed step-size method 'euler' and 'rk4' are the same as in Figure 34. This is because the tolerance does not change anything for them.

We can also point out that at such sharp tolerances, 'radau' is no longer

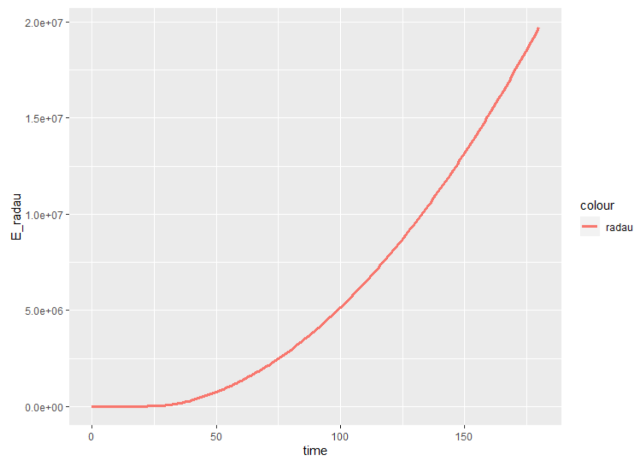

Figure 35: State dependent discontinuity model of Radau in R

computing solutions exhibiting the abnormal behavior we saw previously. From Figure 37, we can see that it oscillates between 10000 and 25000 as it should. From supplementary experiments, we observe that ‘radau’ starts performing in a measure that is comparable to the other solutions at about a tolerance of 10^{-9}.

3.1.2 State dependent discontinuity model in Python

We also perform the experiment with if-statements and global variables in Python with equally inaccurate results.

Figure 38 shows what happens when the problem is coded with global variables and if-statements in Python. We can see that the results are similar to those in R. This is even though all solvers in Python have error control.

We note that all the solvers except ‘RK23’ give solutions that at least oscillate between 10000 and 25000, though in completely dissimilar patterns. The solutions have peaks and troughs at different times and no warnings were given by the solvers.

The ’RK23’ solver, in purple, computes a solution with a completely different pattern than the other solvers. It never reaches 25000 and only oscillates between around 10000 and 15000.

Again, as shown in Figure 39, ’Radau’ computes a solution that grows exponentially even though the parameter β is eventually set to start an exponential decay as happens with all other solvers.

We then used very sharp tolerances to solve the problem but, as is the case in the R environment, none of the solvers obtained a reasonably accurate solution. The highest tolerance we could use in Python without any one method failing was 10^{-12}. Both the absolute and relative tolerance was set to this value and Figure 40 shows the results from this sharp tolerance experiment.

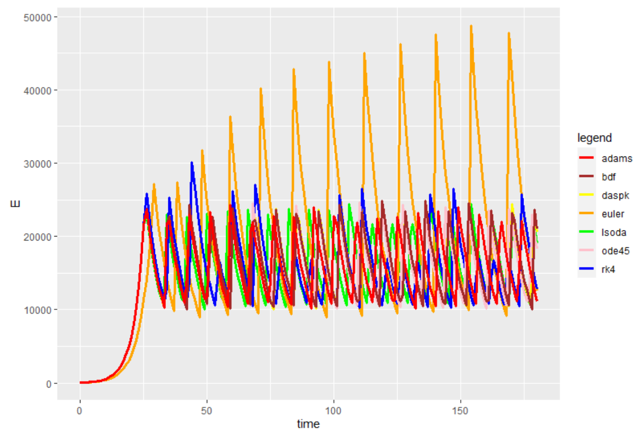

Figure 36: State dependent discontinuity model in R at high tolerances

Figure 40 shows that the results did improve. However, the solvers give solutions that are not in agreement. We note that none of the solvers are oscillating beyond 25000 as was the case with the fixed-step solvers in R. At sharp tolerances, the solutions are aligned for the first few discontinuities with only some blurring until about t=25 when the solvers give substantially different solutions. Though the pattern is correct, none of the solvers give solutions that are in agreement telling us that none got the reasonably accurate solution. Also, none of the solvers follow the solution provided in Section 3.4. (See the comparison against the final solution in Section 3.1.5 to see that even this sharp tolerance solution is not accurate enough.)

We note that ‘RK23’ is now following the correct pattern in that it oscillates between 10000 and 25000 whereas it only reached 15000 at the default tolerances.

Again, as shown in Figure 41, ‘Radau’ begins to give reasonable solutions at these sharp tolerances; those solutions follows the pattern we are expecting but as we will show in Section 3.4, they are still not reasonably accurate solutions. ‘Radau’ starts reasonably performing well at around a tolerance of 10^{-10}.

VI ======= add the event detection solution to the picture ====== VI

We should also note that the R and Python implementation of ’Radau’ are different. The ’Radau’ solver in Python is implemented in Python with the NumPy library whereas R uses the Fortran code. Thus we eliminate the possibility of a bug in the code as well as any problem stemming from the interface from R to Fortran or from Python to NumPy. The problem is simply in how the algorithm interacts with this naive implementation of the state-dependent discontinuity. In our experiments with the Fortran code, in Section 4, the same behavior is observed.

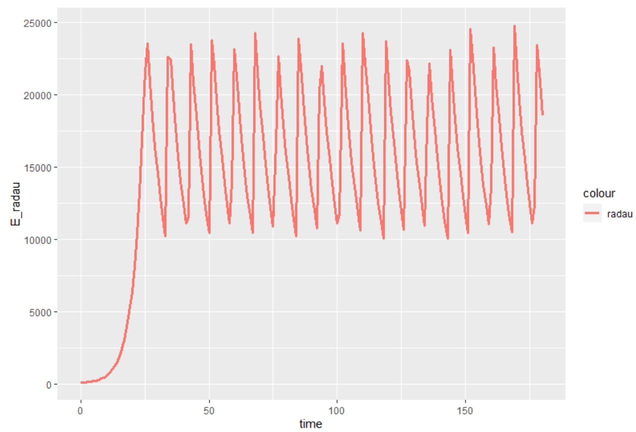

Figure 37: State dependent discontinuity model of Radau in R at high tolerances

3.1.3 State dependent discontinuity model in Scilab

We perform the experiment with if-statements and global variables in Scilab and the results are as shown in Figures 42 and 43.

Figure 42 shows the same issues that we saw before, in Scilab. None of the solvers give solutions that are aligned which prompts us to conclude that none of them are getting an accurate solution. All of the solvers in Scilab have error control and we can also see that their solutions all follow the correct pattern of oscillating between 10000 and 25000. However, as we will discuss in Section 3.4, none of the solutions are very accurate. We note that the spacing between points is not important in this analysis as at the current spacing, even the solvers who depend on the spacing are getting inaccurate answers.

We then repeat the experiment at sharp tolerances. Scilab's 'rkf' does not allow the use of very sharp tolerance as it has a cap of 3000 derivatives so it was omitted in this experiment. The sharpest tolerance we can use in Scilab before the other methods fail is 10^{-13}; the results are shown in Figure 43.

Again, in Figure 43 we can see that the use of sharp tolerances is not enough to force the solvers to compute accurate solutions. The solutions did improve as all the solvers seem to follow the correct pattern but none oscillate between 10000 and 25000 with clear peaks and troughs at those values respectively. For the time period between 0 to 30, the solutions all seem to show reasonable agreement but as we go further in time, all of the solutions diverge. We also note that none of the solvers compute solutions in reasonable agreement with the solution discussed in Section 3.4. (See the comparison against the final solution in Section 3.1.5 to see that even this sharp tolerance solution is not accurate enough.)

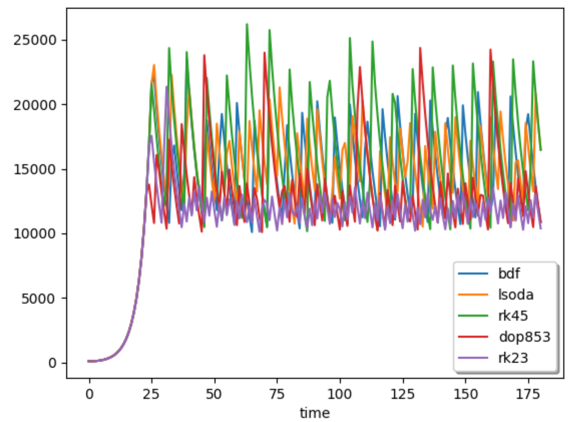

Figure 38: State dependent discontinuity model in Python

3.1.4 State dependent discontinuity model in Matlab

We perform the experiment with if-statements and global variables in Matlab and the results are as shown in Figures 44 and 45.

We see the same incorrect solutions in Matlab at the default tolerances in Figure 44. The solvers do not even consistently reach 25000. We then use a sharper tolerance to see how the solvers act.

Figure 45 shows the experiment with if-statements and global variables at sharp tolerances. We get surprisingly good solutions compared to the solutions in the previous environments. However, as we will see in Section 3.4, these solutions as formed with such sharp tolerances, are computed extremely inefficiently and they are not as accurate as the solution presented in Section 3.4, especially for later time periods. (See the comparison against the final solution in Section 3.1.5 to see that even this sharp tolerance solution is not accurate enough.)

3.1.5 State dependent discontinuity solutions comparison

In all the previous subsections, we have maintained that even the sharp tolerance solutions though more in agreement are not accurate. Here, we present a comparison between LSODA in Python at default tolerance, at the sharpest tolerance, and the final solution we will present shortly. We can see from Figure **??** that the solution both at default and the sharp tolerance does not agree with the accurate solution. We also note that the default tolerance uses 2357 function evaluation, the sharpest uses 4282 evaluations while the final solution uses 535 function evaluations.

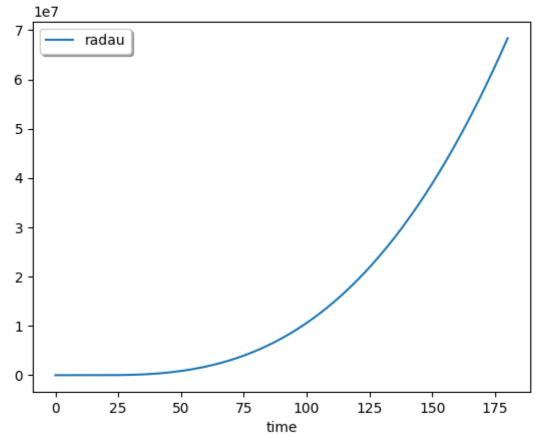

Figure 39: State dependent discontinuity model of Radau in Python

3.2 Why the solvers fail even with sharp tolerances

In this section we discuss why sharp tolerances were not enough to force the solvers to solve the problem in the naive way it is coded, i.e, using global variables and if-statements.

Whenever there is a change in the value of β, the step that first encounters that change will almost always fail. As discussed in Section 1.6, the step-size at a discontinuity will always have to be much smaller than the step-size of a step on a continuous region. Thus the first encounter of a solver with any discontinuity will always be in the context of a failed step.

During this failed step, the value of the E will cross the threshold. The global variables will thus be toggled. But then, when the solver attempts to retake the step, it will be using the wrong β value.

This observation is crucial as it allows us to conclude that just before discontinuity, the function evaluations should be based on the previous β value but they are in fact using the new β value. There is no trivial way to code this behavior in the ODE function, $f(t, y)$, if we do not know the time of the discontinuity.

VI ====== Clarification The solvers need to figure out how to step up to the discontinuity. Then to the left of the discontinuity, the step that we used will employ function evaluations that use the previous β, and then after the discontinuity, the next step will employ function evaluations that use the new β value. There cannot be a step that uses both types of function evaluations ===== VI

At extremely sharp tolerances, even smaller than 10^{-12}, the first step that encounters the discontinuity can also fail. The solver will still have to retake the step but, as discussed before, it will have to use the wrong β value. In

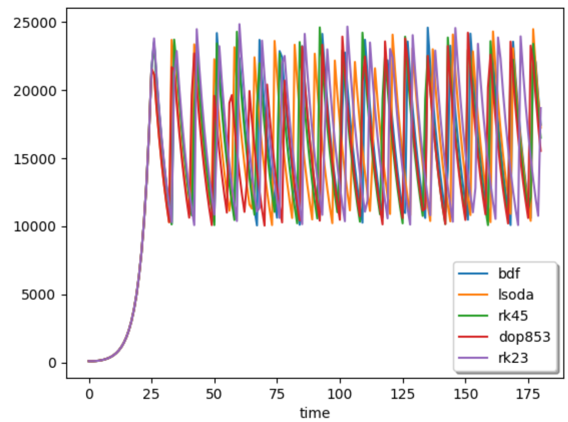

Figure 40: State dependent discontinuity model in Python at sharp tolerances

the next few sections, we will present the correct way to code problems with state-dependent discontinuities so that we get accurate solutions efficiently.

3.3 Introducing event detection

In the time-dependent discontinuity problem, we saw that if we used error-controlled software, then the solvers can work through one discontinuity at sufficiently high tolerances. We also showed that this was not the most efficient way to solve it. For the state-dependent discontinuity problem, we showed in the previous section why using the solvers with sharp tolerances will not be able to solve this problem. Because we do not know when the discontinuities occur, we cannot use the discontinuity handling technique, involving a cold restart, that we used to solve the time-dependent discontinuity problem. However, the idea that we developed in Section 2.2 about integrating continuous sub-problems separately and combining them into a final solution can still be applied here.

To integrate continuous sub-problems, we need a way to detect that a threshold has been met, and then as soon as we reach such a point, we can perform a cold start. This will make the solver integrate the problem one continuous subinterval at a time. In this section, we will explain the capability of modern solvers to detect events and we will show how to encode the E(t) thresholds (either E(t)=25000 or E(t)=10000) as events so that the times at which they occur can be determined, and then we can perform a cold start when they are reached.

To perform event detection, an ODE solver will require two functions from the user: the usual ODE right-hand side function, $f(t, y)$ and another function which we will call the root function (commonly denoted by $g(t, y)$), that determines the events.

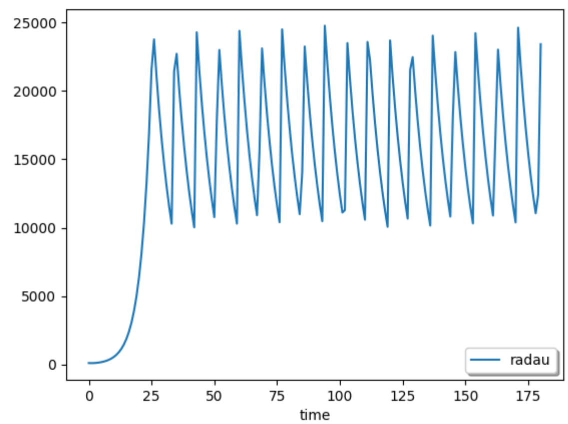

Figure 41: State dependent discontinuity model of Radau in Python at sharp tolerances

The root function is a function that, given the value of the solution to the ODE at the current step will return a real number. The ODE solution is said to have a root whenever the value of the root function is zero. The key idea is that each event must be written so that it occurs at the root of one of the root functions.

The solver calls the root function at the end of each successful step that it takes and will record its value. It will then compare the value of the root function with the corresponding value from the previous step to see if there has been a change of sign. If the value of the root-function changed sign, the solver raises a flag to say that it has detected a root and will then run a root-finding subroutine on that step until it finds the exact point where the root-function returns zero. Most solvers will then return, allowing us to perform a cold start.

Using event detection thus entails defining a function that takes the value of the ODE solution at the current point and returns a real number which is zero whenever we want it to detect an event. For example, if we want to detect whenever x is 100, it is sufficient to define (x - 100) to be the root function. In the next section, we will elaborate on how to use event detection to accurately and efficiently solve the state-dependent discontinuity problem.

We also mention that many modern solvers have event detection built-in. Thus users should just be able to use event-detection solvers from their preferred programming environments without any additional software.

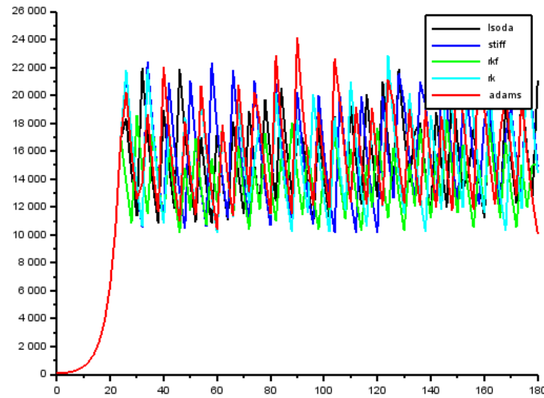

Figure 42: State dependent discontinuity model in Scilab

3.4 Solving the state dependent discontinuity model using event detection

Each toggling between the values of the parameter β introduces a discontinuity. As none of the provided solvers are designed to solve discontinuous problems, we get the erroneous solutions reported in 3.1. We have seen that though sharp tolerances do result in better solutions, none of the solvers were in agreement with each other. The use of such sharp tolerances leads to inefficiencies as well. We will now present an approach using event detection that is both accurate and efficient.

The solution is to use the thresholds that we have defined in our model to define events and integrate only up to each threshold using the event detection capability of the solver. We can then cold start from there and repeat the process with another right-hand side function corresponding to the new β value and with a different root function that encodes the next threshold we are looking for. We repeat this process until we reach the end of the time interval. This approach allows the solvers to integrate continuous sub-problems one at a time and these sub-problems can then be combined into a final solution.

For our specific problem, event detection is used as follows: We start by solving the problem with β at 0.9 and with a root function that detects when E is equal to 25000. Once we detect the time at which E=25000, we do a cold start. We extract the solution of the solver at the time of the event and use those values as the initial value for our next call to the solver. This next call will have β at 0.005 and a root function that detects a root when E=10000. We again integrate up to that new threshold and cold start when we reach it. The new cold start will have β at 0.9 and the root function looking for 25000 as the event. This is repeated until we reach the desired end time.

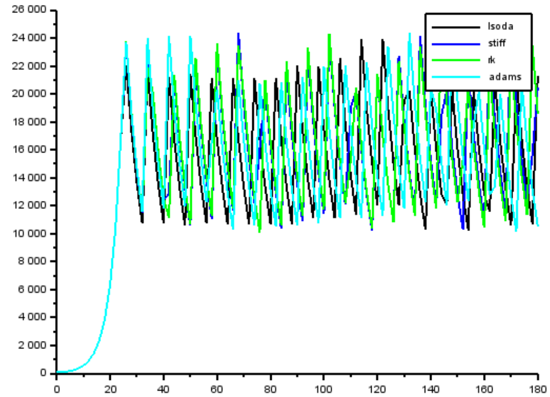

Figure 43: State dependent discontinuity model in Scilab with sharp tolerances

The pseudo-code is as follows:

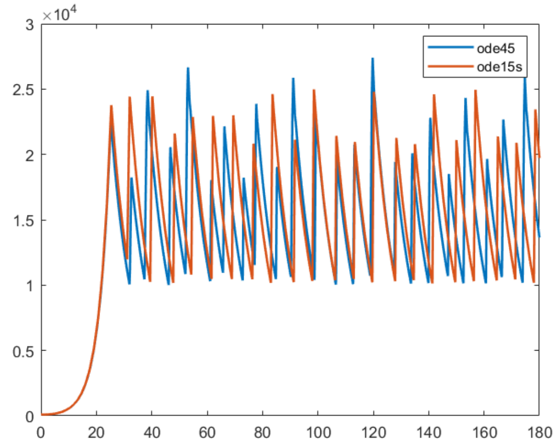

Figure 44: State dependent discontinuity model in Matlab

```
function model_no_measures(t, y):
    beta = 0.9
    // code to get dSdt, dEdt, dIdt, dRdt
    return (dSdt, dEdt, dIdt, dRdt)

function root_25000(t, y):
    E = y[1]
    return E - 25000

function model_with_measures(t, y):
    beta = 0.005
    // code to get dSdt, dEdt, dIdt, dRdt
    return (dSdt, dEdt, dIdt, dRdt)

function root_10000(t, y):
    E = y[1]
    return E - 10000

res = array()
t_initial = 0
y_initial = (S0, E0, I0, R0)
while t_initial < 180:
    tspan = [t_initial, 180]
    if (measures_implemented):
        sol = ode(model_with_measures, tspan, y_initial,
            events=root_10000)
        measures_implemented = False
    else:
        sol = ode(model_no_measures, tspan, y_initial,
            events=root_25000)
        measures_implemented = True
    t_initial = extract_last_t_from_sol(sol)
    y_initial = extract_last_row_from_sol(sol)
    res = concatenate(res, sol)

// use res as the final solution
```

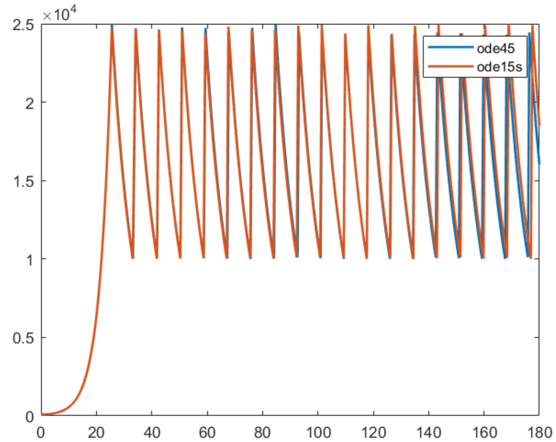

Figure 45: State dependent discontinuity model in Matlab with sharp tolerances

Some programming environments, such as Python, by default, do not stop the integration when the first event is detected. To do a cold start, we need the solver to stop at events, and to make this happen, in some programming environments we need to set appropriate flags.

3.4.1 Solving state dependent discontinuity model in R

Several of the solvers in R have event detection capabilities. These are: 'adams', 'bdf', 'lsoda', 'radau', and will be used in this section to solve the model using the approach described in the previous subsection. From Figure 47, we can see that all the solvers give solutions that are in agreement except 'Radau'. This is in contrast with what happened previously when we were integrating a discontinuous problem, even at sharp tolerances.

The case of 'Radau' is interesting as it was giving a poor quality solution at the default tolerances, without event detection but it is now giving at least a solution that is exhibiting a correct pattern. We note that at high tolerances 'Radau' with event detection approach the results from the other solvers, as shown in Figure 48. We will also note the terrible performance of Radau in Table 15.

We will show in Table 15 that introducing event detection also made the solvers significantly more efficient while giving us better results.

We note that it is unfair to compare the efficiency of the solvers at the default tolerances with the efficiency of the solvers when they use event detection's as the results for the former are inaccurate.

We can see from Table 15 that with event detection we are gaining an improvement of around 1000 function evaluations for 'lsoda', 13000 in 'radau', 4000 in 'bdf', and 1300 in 'adams' while having more accuracy. This significant

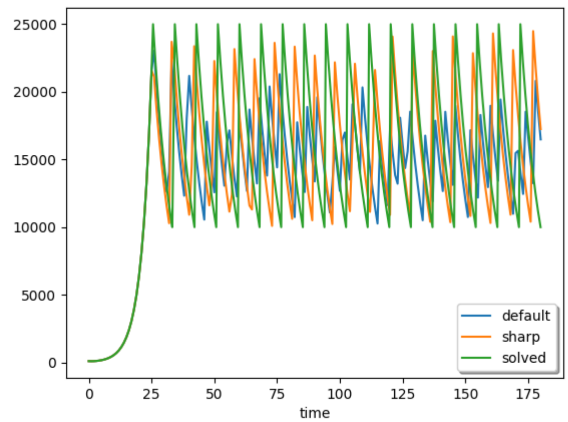

Figure 46: State dependent discontinuity model solutions comparison

Table 15: R state discontinuity model

method	no event	no event with sharp tol.	with event detection	with event detecton at sharp tol.
lsoda	2135	4658	1248	3435
radau	1002	21835	2151	14681
bdf	3300	9803	1678	7963
adams	1368	3467	817	2689

decrease in the number of function evaluations will lead to much faster CPU times, especially when the right-hand side function, $f(t, y)$ is more complex.

Also, we can see from the table that the solvers use fewer function evaluations compared with event detection than without event detection at the default tolerances.

We also note that the Fortran code for Radau does not have event detection and that event detection was added through the R interface.

3.4.2 Solving state dependent discontinuity model in Python

All the solvers in Python have event detection and thus all will be used in this part of the study. In Python, *solve_ivp*() does not stop when an event is detected by default. We thus need to set the terminal flag of the root functions. (Example: $root\_10000.terminal = True$). Again, Figure 49 shows that all the solvers give solutions that are in agreement, suggesting that this is the correct solution. This is different from our results even at sharp tolerances. We will also see that this is a much more efficient approach across all the solvers.

We note that even 'Radau' is in agreement with the solution using default

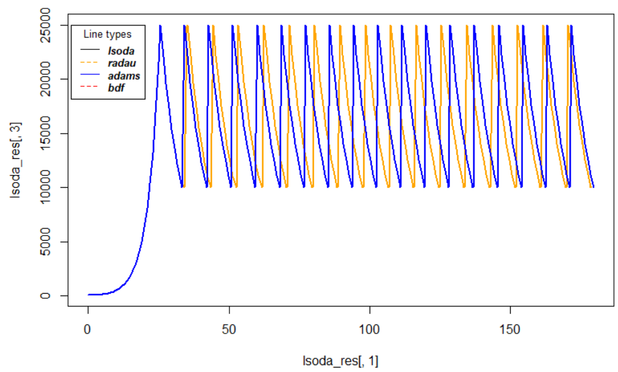

Figure 47: Solving state discontinuity model in R

tolerances in Python and R when using event detection even though the "Radau" implementation in Python is different from the one in R. Python uses a Python implementation while R uses the Fortran code.

As is the case with R, we cannot compare the default tolerance efficiency data to the event detection efficiency data as the former corresponds to inaccurate results. So, in Table 16, we compare the sharp tolerance efficiency data with the data from the event detection computation.

Table 16: Python state discontinuity model

method	no event	no event with sharp tol.	with event detection
lsoda	2357	4282	535
bdf	2301	11794	808
radau	211	74723	990
rk45	1484	17648	674
dop853	11129	21131	1514
rk23	4307	246644	589

Table 16 shows that the number of function evaluations when the solvers use event detection is far less when they do not; 'LSODA' used around 3000 fewer function evaluations, 'BDF' used 11000 less, 'Radau' used 74000 less, 'RK45' used 17000 less, 'DOP853' used 20000 less and 'RK23' used 246000 less. The reduction in CPU times from this will be significant across all the solvers, especially with a more complex right-hand side function.

What is more surprising is that the solvers with event detection also perform better than the solver using no event detection with default tolerance code without event detection. In all solvers except 'Radau', event detection performed

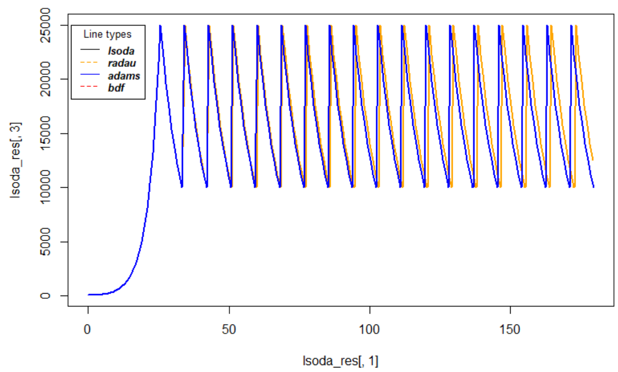

Figure 48: Solving state discontinuity model at sharp tolerances in R

better than the default tolerance code without event detection. We note that 'Radau' at default tolerance gives an extremely inaccurate solution and thus its better performance here should not be trusted.

3.4.3 Solving state discontinuity dependent model in Scilab

There is only one solver with root functionality in Scilab; it is 'lsodar', the root-finding version of 'lsoda'. Judging from the solutions we obtained from Python and R, it seems that 'lsodar' gave a correct solution as well. It oscillates in the correct pattern and goes sharply between 10000 and 25000.

Table 17: Scilab state discontinuity model

method	no event	no event with sharp tol.	with event detection
lsoda	2794	4636	1327

From Table 17, we can see that the root-finding code use fewer function evaluations that 'lsoda' both at sharp and default tolerances.

3.4.4 Solving state dependent discontinuity model in Matlab

Both $pde45$ and $ode15s$ have an event detection capability. We applied event detection to model the problem with the solvers in the Matlab environment and the results are shown in Figure 51. We remember that the solutions in Matlab without event detection were surprisingly accurate but were in disagreement with each other at points further in time. We can see that with event detection, the solutions are all in agreement at the default tolerances even at points further

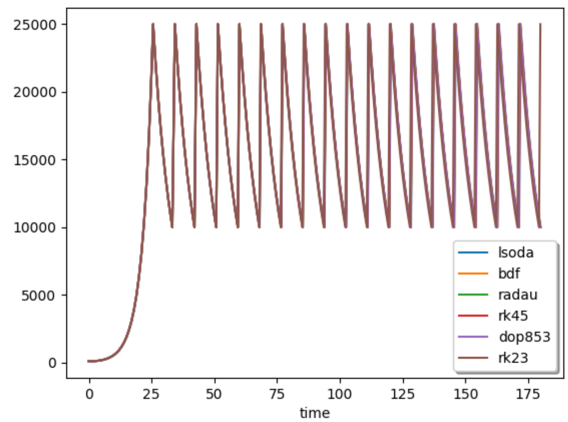

Figure 49: Solving state dependent discontinuity model in Python

in time. We also see, in Table 18, that the use of event detection is also more efficient than without event detection.

Table 18: Matlab state discontinuity model problem

method	no event	no event with sharp tol.	with event detection
ode45	2023	22411	859
ode15s	1397	11550	620

We can see in Table 18 that the computation with event detection uses fewer function evaluations than both the code without event detection at default and sharp tolerances. We see that the computations with sharp tolerances, although they give acceptable solutions, use 20000 more function evaluations in $ode45$ than the computation with event detection and 11000 in the case of $ode15s$ than the computation with event detection.

3.5 Efficiency data and tolerance study for the state dependent discontinuity problem

In this section, we will investigate how sharpening the tolerance improves the results in the case of the non-event detection experiment. We will also investigate coarsening the tolerance with event detection to show how coarse a tolerance we can use while getting acceptable results.

We will perform this analysis on LSODA across R, Python, and Scilab, as they appear to use the same source code, and with R and Python versions of DOPRI5 which do not use the same code but do use the same Runge-Kutta

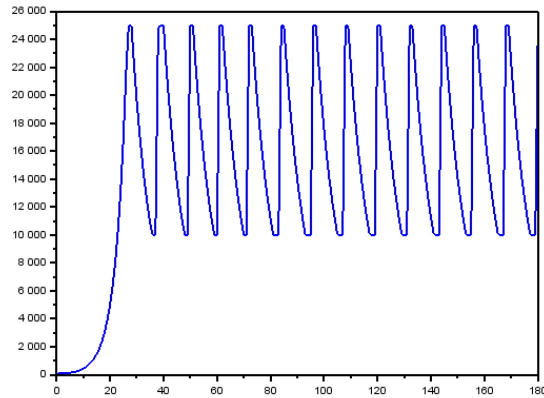

Figure 50: Solving state discontinuity model in Scilab

pair and with the Scilab version of RKF45 which is not the same code, nor the same pair but is a Runge-Kutta pair of the same order. We also use *ode*45 in Matlab as it is an implementation of DOPRI5 in Matlab.

3.5.1 Comparing LSODA across platforms for state discontinuous problem

In this section, we use the R version of LSODA at multiple tolerances. We set both the relative and the absolute tolerance to a particular value and analyze the solution.

We know that without event detection, LSODA does not give accurate results even at very sharp tolerances. We will also examine how coarse we can set the tolerance to still have the event detection computation yield reasonable results.

State dependent discontinuity LSODA tolerance study in R Figure 52 shows that LSODA applied to the same problem at different tolerances gives vastly different results. We would expect the solutions at the sharper tolerances to be along very similar curves but that is not the case. The computation is suffering from the fact that the first step that encounters a discontinuity fails while still switching the global variables. This further supports our statement that for any state-dependent discontinuity, we cannot get reasonable results simply by sharpening the tolerance.

From Figures 53 and 52, we can see the clear advantage of using event detection. Event detection even allows us to use very coarse tolerances while solving the problem to a reasonable accuracy. Event detection allows us to use tolerances of 10^{-3} and sharper to get reasonable results while the computation without event detection still failed at a tolerance of 10^{-13}. We also analyze the

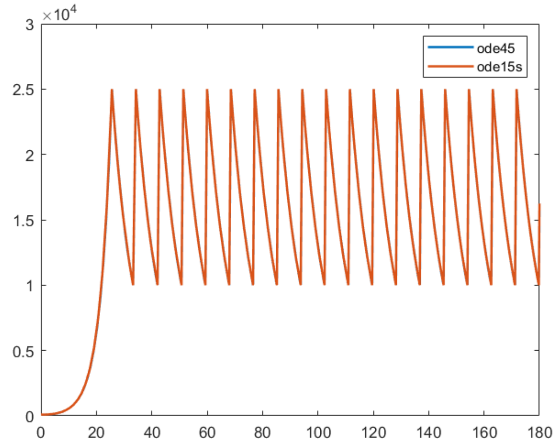

Figure 51: Solving state discontinuity model in Matlab

differences in efficiency between the two codes in Table 19.

Table 19: R version of LSODA applied to state discontinuity model tolerance study

tolerance	no event detection	with event detection
1e-01	675	560
1e-02	1856	522
1e-04	1863	752
1e-06	2135	1248
1e-07	2676	1874
1e-08	2730	2060
1e-10	3337	2604
1e-11	3603	3054

Table 19 shows a decrease in the number of function evaluations across all tolerances which will translate into faster CPU times when the right-hand side function is more complex. We note that the comparison is unfair as the computations without event detection do not give a reasonably accurate answer. Furthermore, the latter computation uses more function evaluations. This supports our conclusion that event detection is the appropriate way to solve state-dependent discontinuity problems.

State dependent discontinuity model LSODA tolerance study in Python In this section, we use the Python version of LSODA at multiple tolerances to see how it performs. We set both the absolute and relative tolerance

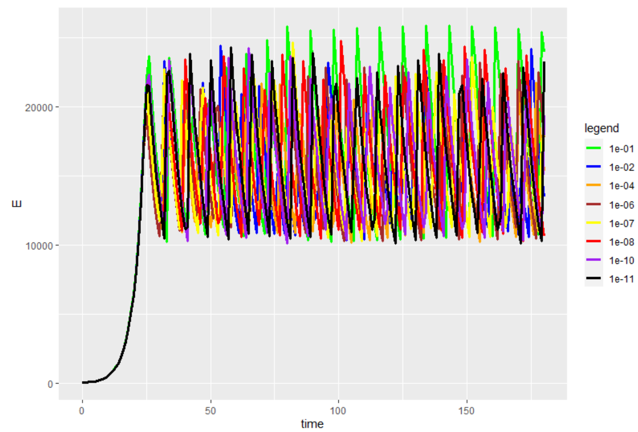

Figure 52: State dependent discontinuity model tolerance study on the R version of LSODA without event detection

to a particular value.

We note that LSODA without event detection even at very sharp tolerances in Python was still giving accurate results but we will see how the solutions change as the tolerance is increased.

We will also show that coarse tolerances can be used with the computation that uses event detection.

Again Figure 54 exposes that LSODA applied to the same problem at different tolerances give substantially different results. We would expect the computations at the sharper tolerances to give substantially similar results but this is not the case. This confirms that, even at sharp tolerances, the step-size when first encountering the discontinuity is still too big. That first step will fail but will still switch the global variables. As a result, this problem cannot be solved sufficiently accurately by sharpening the tolerance.

From Figures 55 and 54, we can see that the addition of event detection allows for the use of a coarser tolerance. We also note that the computations with event detection blur as we go further in time. This is because the coarser tolerance computations are not giving a sufficiently accurate solution. In Python, it is at a tolerance of 10^{-4} and sharper that we get reasonably accurate results.

We analyse the efficiency of the computations in Table 20. We must note that this analysis is unfair as the computation without event detection does not give an accurate solution to the problem. Still, we will see that the event detection computation uses fewer function evaluations while getting a more accurate answer.

State dependent discontinuity model LSODA tolerance study in Scilab We perform the same experiment in Scilab. We set the absolute and relative tolerance to the same values and run the solvers. For the different

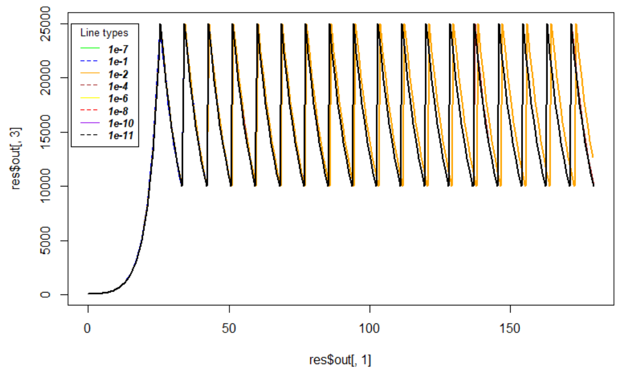

Figure 53: State dependent discontinuity model tolerance study on the R version of LSODA with event detection

Table 20: Python version of LSODA applied to state discontinuity model tolerance study

tolerance	no event detection	with event detection
0.1	1207	425
0.01	1627	454
0.0001	1968	689
1e-06	2122	1305
1e-07	2684	1807
1e-08	2730	2099
1e-10	3337	2639
1e-11	3603	3098

tolerance values, we plot the solutions and analyze how the solutions computed without event detection change as the tolerance is sharpened; we also examine how coarse a tolerance we can use with the event detection solvers.

Again, Figure 56 exposes the behavior whereby the same solver applied to the same problem at different tolerances gives substantially different results. We would expect the code at the sharper tolerances to give very similar curves but clearly, LSODA even at sharp tolerances does not.

From Figure 57, we can see that the use of the event detection allows us to use a smaller tolerance. We can use a tolerance of 10^{-3} and still get an accurate answer whereas, without event detection, even tolerance of 10^{-12} is not sufficient.

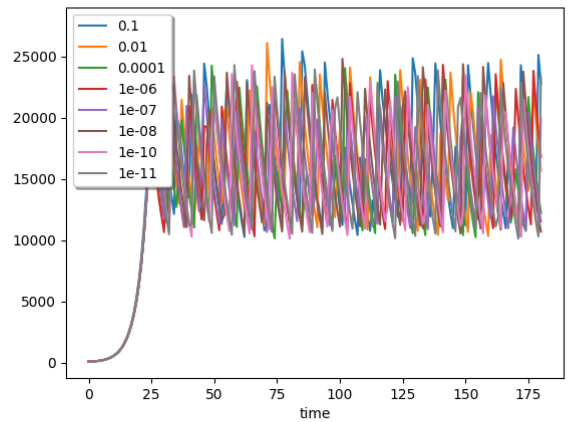

Figure 54: State dependent discontinuity model tolerance study on the Python version of LSODA without event detection

3.5.2 Comparing Runge-Kutta pairs across platforms for state discontinuous problem

In this section, we consider solvers based on Runge-Kutta pairs of the same order: DOPRI5 in R aliased as 'ode45', DOPRI5 in Python aliased as 'RK45', RKF45 in Scilab aliased as 'rkf' and *ode*45 in Matlab.

We remember that without event detection, none of these solvers across the platforms solved the problem correctly even with sharp tolerances. We will show what happens to these solvers as the tolerance is sharpened. We also coarsen the tolerance for the case where solvers use event detection where that is possible to see how coarse the tolerance can be while still obtaining sufficient accuracy.

Tolerance study on state discontinuity using the R version of DOPRI5 The R version of DOPRI5 does not have event detection but we still perform the experiment on this solver without event detection. We pick several values for the absolute and relative tolerances and run the solvers. In so doing we see how the code performs as the tolerance is sharpened.

From Figure 58, we see that LSODA, whereby solver applied to the same problem with different tolerances, gives significantly different solutions. The global variables will be switched during the first step that encounters the discontinuity and thus the problem cannot be solved accurately. This strengthens our conclusion that the problem with state-dependent discontinuities cannot be solved without event detection.

We then report on the efficiency data for this case in Table 22.

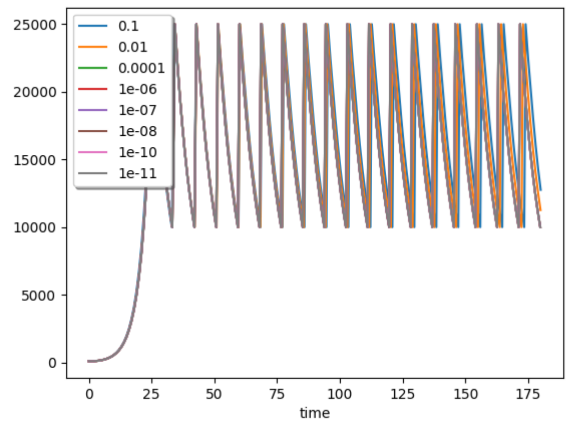

Figure 55: State dependent discontinuity model tolerance study on the Python version of LSODA with event detection

Tolerance study on state discontinuity using the Python version of DOPRI5 We perform the same experiment in Python. The absolute and relative tolerances are set to a range of values and the solver is run both with and without event detection. We report on how the code performs as the tolerance is increased in the case without event detection. Since the Python version of DOPRI5 has event detection, we will see how coarse the tolerance can be set while still giving us an accurate solution. We must note that the solver crashes if we ask for a tolerance of 0.1.

In Figure 59, we can see that even at sharp tolerances, the solver is not able to compute a reasonably accurate solution. The global variables are switched at the first encounter with the discontinuity and thus the problem cannot be solved accurately simply by decreasing the tolerance.

In contrast, when using event detection, the code can use very coarse tolerances. We can see that a tolerance of 10^{-4} is sharp enough to solve the given problem accurately; the blurring that occurs is due to the coarser tolerances. We present the efficiency data in Table 23 to show how the code with event detection is also far more efficient.

We can see in Table 23 that across all the different tolerances, the solver with event detection requires fewer function evaluations, around several thousand fewer for the sharper tolerances.

State dependent discontinuity RKF45 tolerance study in Scilab Scilab uses RKF45 which is a different Runge-Kutta pair from what is used in DOPRI5 but the pairs have the same order. It does not have event detection but we can still perform the experiment on the solver without event detection. We pick several values for the absolute and relative tolerances and run the solvers.

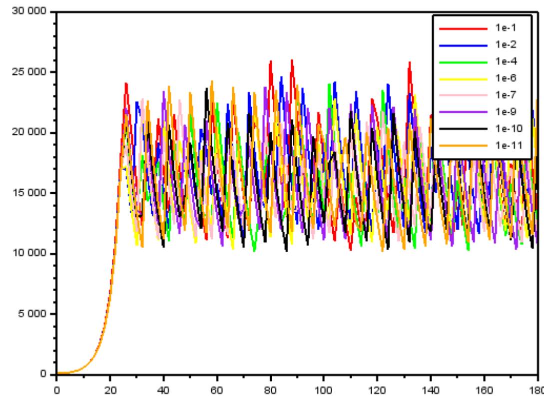

Figure 56: State dependent discontinuity model tolerance study on the Scilab version of LSODA without event detection

In so doing we see how the solver performs as the tolerance is sharpened.

The Scilab version of 'rkf' can only integrate up to time 90 as it has a hard cap of 3000 derivative evaluations but this is enough to see that even at sharper tolerances, the solutions are not in agreement. Figure 61 shows that the problem cannot be solved by simply using sharper tolerances. We can conclude that event detection is required.

Tolerance study on state discontinuity using the Matlab version of DOPRI5 We apply different tolerances to the state problem with and without event detection on the *ode*45 function which is a Matlab implementation of DOPRI5.

From Figure 62, we can see that the solution obtained with a tolerance of 0.1 is of poor quality without event detection. It does not follow the correct pattern of oscillating between 10000 and 25000. The computations of the other tolerances follow the correct pattern but are not in agreement.

In Figure 63, we can see that the computations corresponding to most tolerances give solutions that are in agreement. A tolerance of 0.1 now follows the correct pattern but is not in agreement with the other tolerances at further points in time. For tolerances of 10^{-2} and sharper, we get accurate solutions. We also see show event detection allows us to use fewer function evaluations.

Table 25, although being an unfair comparison since the solver without event detection did not give accurate solutions, shows that this way of solving the problem is also less efficient. At the tolerance of 0.1, the smaller number of function evaluations for the solver without event detection is not relevant since the solution at a tolerance of 0.1 is very inaccurate. At all the other tolerances, the code with event detection is both more accurate and more efficient, usually

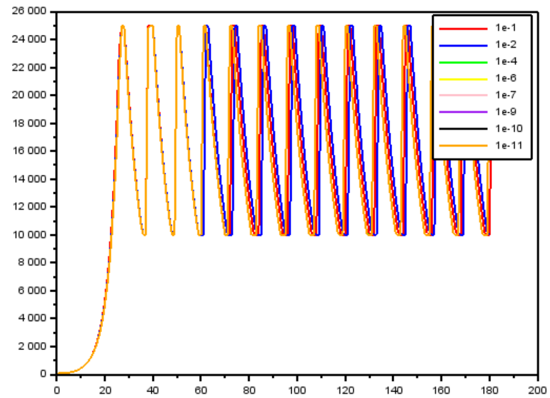

Figure 57: State dependent discontinuity model tolerance study on the Scilab version of LSODA with event detection

using less than half the number of function evaluations.

Table 21: Scilab version of LSODA applied to state discontinuity model tolerance study

tolerance	no event detection	with event detection
0.1	1141	287
0.01	1606	262
0.0001	1968	523
0.000001	2122	983
0.0000001	2684	1307
1.000D-08	2730	1567
1.000D-10	3380	1963
1.000D-11	3603	2331

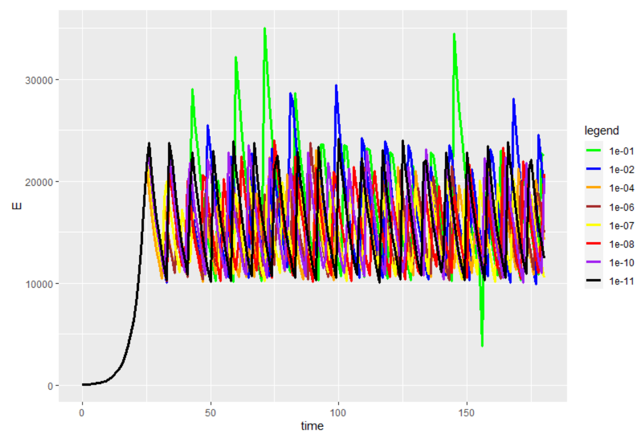

Figure 58: State dependent discontinuity model tolerance study on the R version DOPRI5 without event detection

Table 22: R version of DOPRI5 state discontinuity model tolerance study

tolerance	no event detection
1e-01	1082
1e-02	1142
1e-04	2014
1e-06	2027
1e-07	2193
1e-08	2919
1e-10	5194
1e-11	7690

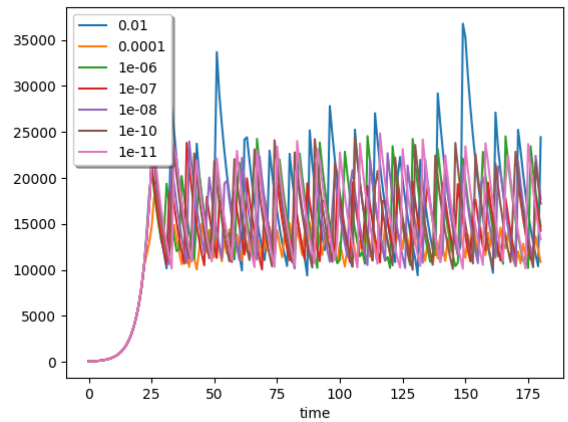

Figure 59: State dependent discontinuity model tolerance study on the Python version of DOPRI5 without event detection

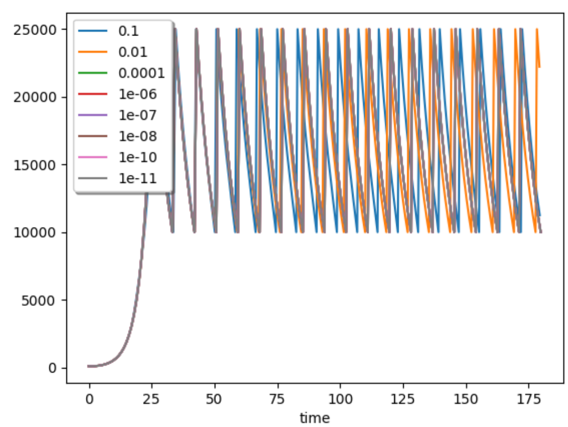

Figure 60: State dependent discontinuity model tolerance study on the Python version of DOPRI5 with event detection

Table 23: The Python version of DOPRI5 state discontinuity model tolerance study

tolerance	no event detection	with event detection
0.01	1400.0	664.0
0.0001	8462.0	806.0
1e-06	6248.0	1232.0
1e-07	6848.0	1754.0
1e-08	7082.0	2354.0
1e-10	10262.0	5066.0
1e-11	13058.0	7688.0

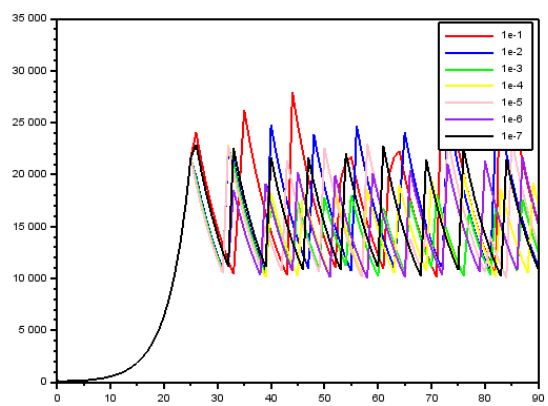

Figure 61: State dependent discontinuity model tolerance study on the Scilab version of RKF45 without event detection

Table 24: The Scilab version of RKF45 State Discontinuity tolerance study

tolerance	no event detection
0.1	547
0.01	732
0.001	1294
1e-4	1956
1e-5	2364
1e-6	2662
1e-7	2802

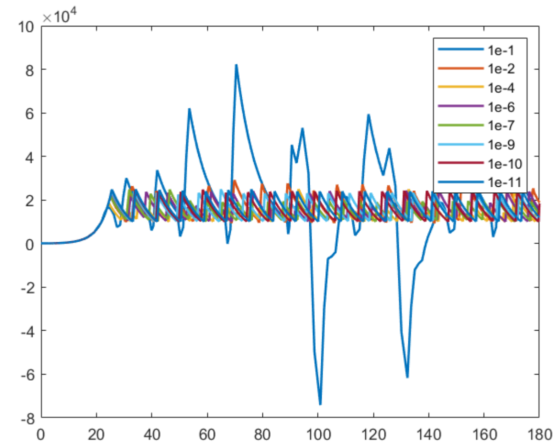

Figure 62: State dependent discontinuity model tolerance study on the Matlab version of DOPRI5 without event detection

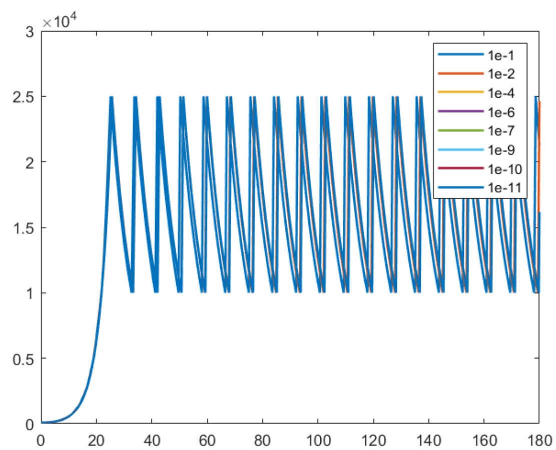

Figure 63: State dependent discontinuity Model tolerance study on the Matlab version of DOPRI5 with event detection

Table 25: Matlab DOPRI5 state discontinuity model tolerance study

tolerance	no event detection	with event detection
0.1	415	650
0.01	1339	661
0.0001	4891	901
1e-06	5803	1411
1e-07	7225	1873
1e-09	9739	4039
1e-10	12385	6043
1e-11	16357	9277

4 Investigation of the cause of inaccuracies that arise for some of the numerical software packages when they are applied to the models

4.1 Radau

In this section, we try the state-dependent discontinuity problem with the Fortran solver radau5.f. We investigate how the original Fortran solver deals with the discontinuity. We note that in both R and Python that 'Radau' exhibits an unusual behavior where the solver that is computed does not oscillate between 10000 and 25000 but rather grows exponentially.

We also note that the event detection in 'Radau' in R is added through a C interface and that may explain why Radau in R and Python gives different results with root finding

We first try the Fortran solver at a tolerance of 10^{-6} which is the default in R.

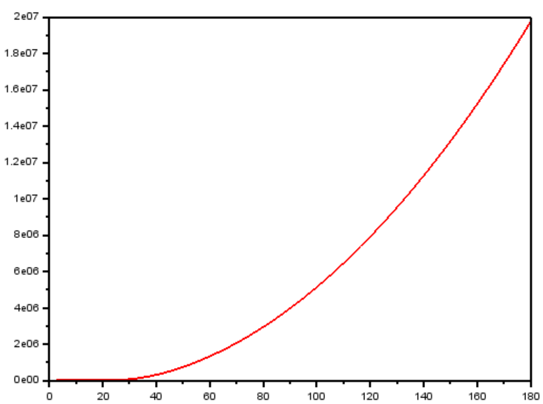

Figure 64: Fortran's radau5.f at tolerance of 10^{-6}

From Figure 64, we again see the unusual behaviour. We also note that it behaves exactly as in R.

We then repeat the process with a tolerance of 10^{-12}. In Figure 65, we can see that the computed solution now follows the correct pattern, although it is still not the correct solution that we described in 3.4.

From this investigation of the Fortran source code, we can conclude that the issue is not in the interface from R to the Fortran solver or the Python implementation.

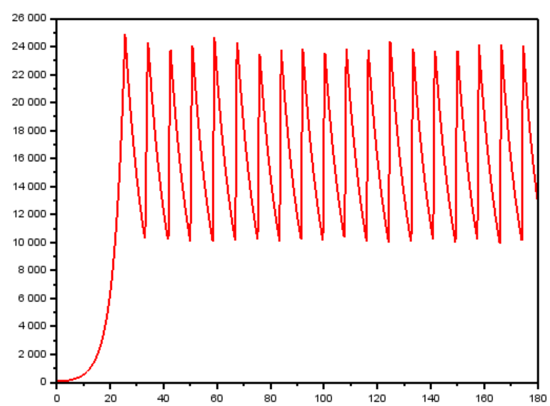

Figure 65: Fortran's radau5.f at tolerance of 10^{-12}

5 Summary, Conclusions, and Future Work

5.1 Summary and Conclusions

Our starting assumption for both models is a reasonable implementation that might typically be employed by a computational scientist. This includes fixed-step size solvers as well as implementations based on the introduction of if-else statements into the functions that define the ODE systems. We reported on the stability and discontinuity issues associated with SEIR models. We showed how stability affects our solutions even if there is a small change in the initial values. We showed how discontinuities reduce the efficiency of the solvers and presented a straightforward way to detect that the problem at hand is discontinuous.

We then used ODE software packages in R, Python, and Scilab to model two Covid-19 problems, one with a time-dependent discontinuity and one with a state-dependent discontinuity.

For the time-dependent discontinuity problem, we have shown that error-controlled ODE solvers can step over one discontinuity with sufficiently sharp tolerances while fixed step-size solvers cannot. We have shown that although error-controlled solvers can solve the problem, the use of discontinuity handling in the form of cold starts leads to more efficient solutions that allow us to use coarser tolerances.

For the state-dependent discontinuity problem, we have shown that even error control solvers cannot successfully step over multiple discontinuities. We have shown that if the discontinuity is state-dependent, we cannot straightforwardly implement the model using the model function $f(t, y)$. We then introduced event detection and showed how it can be used to model state-dependent discontinuity problems by encoding the thresholds as events and applying cold starts. Using event detection provides an efficient and accurate way to solve such problems.

From the usage of the different packages, we also found a certain inconsistency. We note that R and Scilab do not use the interpolation capabilities for the solver by default. We would advise software implementers to use the capabilities of the solver's interpolation. Using the method of forcing the solver to integrate exactly to given output points reduces the efficiency of the algorithm. The algorithm is no longer allowed to take as big a step as it should. We also recommend against using fixed step-size solvers.

We recommend using some form of discontinuity handling rather than introducing an if-statement into the right-hand side function that defines the ODE.

When a researcher has a problem that has a time-dependent discontinuity that occurs at a known time, they should use the form of discontinuity handling presented in this report. Using cold starts allows the researcher to integrate continuous subintervals of the problem in separate calls leading, to efficient and accurate solutions.

When a researcher has a problem that has a state-dependent discontinuity, they should map out the thresholds at which these discontinuities occur and look to use event detection with these thresholds as events. They can then cold start at each event and integrate continuous subintervals of the problem in separate calls to the solvers. This leads to efficiency and accuracy that is not possible using a naive treatment.

5.2 Future Work

In Section 3.1, we see that 'Radau' exhibits unusual behavior when solving the state-dependent problem. Further analysis needs to be done on the algorithm itself as two different implementations of the algorithm gave similarly poor quality solutions.

We also propose to do the same discontinuity analysis on Covid-19 PDE models to see how error-controlled and non-error-controlled PDE solvers differ. We can also use BACOLIKR or other root-finding capable software to analyze how they improve the solutions to discontinuous PDE problems.

6 Bibliography

7 Appendix: Parameter fitting in an SEIR Model

In [Chritian L. Althaus's paper reference], the epidemiologist uses data from the Ebola spread in three different West African countries to understand the impact of the implemented control measures. To do this, the researcher needed to estimate parameters like the basic and effective reproduction number of the virus.

These parameters are estimated by doing a best fit optimization on the parameters applied to an SEIR model. The experiment is to use an ODE model with certain values of these parameters and calculate the error of these models based on real-life data. The model with the minimum error is the 'best fit' model and the corresponding parameters. These estimates are then used to understand the spread of the virus.

We note that the ODE model is run inside an optimization algorithm and thus its efficiency is critical as the algorithm will need to solve the ODE model with each different set of parameters.

The following is the pseudo-code for our attempt at replicating the experiment reported in [Chritian L. Althaus's paper reference]:

```
data = read_csv("ebola_data.csv")

function model(t, y, parms):
    // define the SEIR model
    return (dSdt, dEdt, dIdt, dRdt)

function ssq(parms):
    // get the model
    out = ode(model, initial_value, times, parms)

    // we calculate the error from the data points as such:
    ssq = abs(out.C - data.C) + abs(out.D - data.D)
    return ssq

parms = c(beta=0.27, f=0.74, k=0.0023)
fit = optimise(par=parms, errorFunc=ssq)

// fit will contain the optimal parameter values...
```

The figure that was reported in [ref to althaus paper] is shown in Figure 66. Our results are as shown in Figures 67, 68 and 69. We see good agreement between our results and those reported in [Ref to Althaus]

Figure 1. Dynamics of 2014 EBOV outbreaks in Guinea, Sierra Leone and Liberia. Data of the cumulative numbers of infected cases and deaths are shown as red circles and black squares, respectively. The lines represent the best-fit model to the data. Note that the scale of the axes differ between countries.

Figure 66: Original figure in Ebola paper

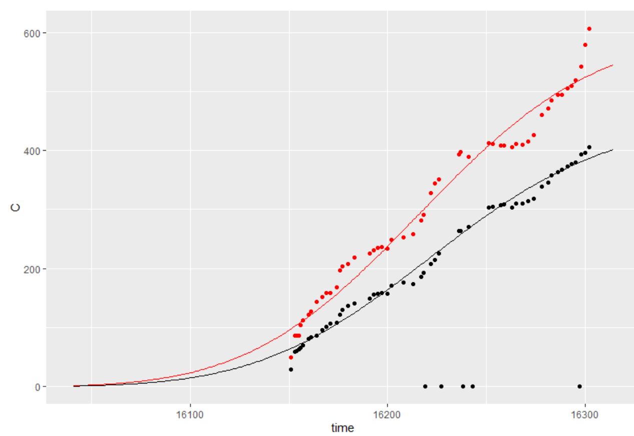

Figure 67: Our Guinea Figure

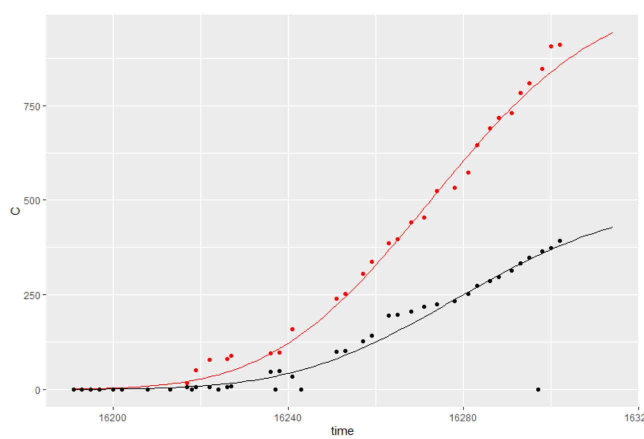

Figure 68: Our Sierra Leone Figure

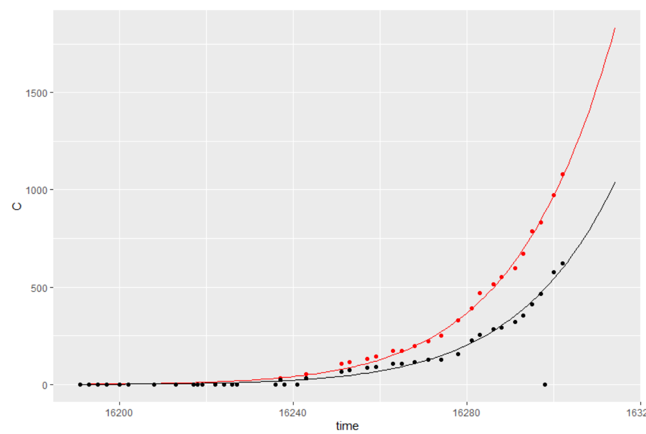

Figure 69: Our Liberia Figure