1 Introduction

In this report, we will provide the results of a careful investigation of the performance of the software packages applied to a typical initial value ordinary differential problem encountered in Covid-19 modelling.

For any mathematical model, the relevance of the 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. 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 quantities being researched.

In Section 1.1, we review how IVODES are used in epidemiology. In Section 1.2, we define the SEIR models which we will consider throughout this paper, 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 and in section 1.6, we discuss the effects of discontinuity on these 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 a problem with evaluation at output points that leads to inefficiencies in Section 1.5

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 some 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 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 will look through 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 potential for future work

1.1 Epidemiological modelling

One common form of epidemiological study is forecasting. Using previously obtained parameters, the researcher develops a mathematical model involving differential equations which is 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 would 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 model for large time periods if the infection is still growing exponentially but how measures such as social distancing allow us to reduce modelling errors so we can hope to integrate over longer time periods.

The second type of epidemiology study involve parameter estimation. In this kind of study, data points are collected about on 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 minimise the error in the fit. The parameters estimated, as such, 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 involves using an ODE solver inside a regression or optimisation 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.

(Reference Christina Christara.) In this section, we describe the models that we are going to use. They involve typical SEIR models to which we add discontinuities.

The model is as follows:

$$\frac{dS}{dt} = \mu N - \mu S - \frac{\beta}{N} IS \tag{1}$$

$$\frac{dE}{dt} = \frac{\beta}{N} IS - \alpha E - \mu E \tag{2}$$

$$\frac{dI}{dt} = \alpha E - \gamma I - \mu I \tag{3}$$

$$\frac{dR}{dt} = \gamma I - \mu R \tag{4}$$

In this SEIR model, we describe the epidemic over time. S is the number of susceptible individuals, E is the number of exposed individual, 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 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 solvers. 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 trick epidemiologists into giving wrong conclusions and try to change the mathematical models themselves when it is the solvers which are at fault.

The discontinuity 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 abrupt changes in a modelling parameter introduces a discontinuity as we will show in Section 1.6.

In 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.

In 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 create a discontinuity. Then, when the population of exposed people drops below a certain minimum threshold, the measures are relaxed which increase β back, which is another discontinuity. We will try to model this problem through a long time period corresponding to the '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 population size of Canada), 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 problem of instability

It turns out that some of the solutions 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 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 impossible but the addition of the measures such as social distancing can give us hope that our 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 value results in drastically different solutions, the ODE is said to be stable.

It is straightforward to see that the solution to problems that exhibits exponential growth are unstable. This is the case with Covid-19 modelling. The population of infected people grow exponentially for as long as there are no measures or the number of infected people is small compared to the size of the population. This means that ODE solvers will experience difficulties in obtaining accurate numerical solutions even if we are slightly wrong in our initial estimate of the number of infected people.

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.

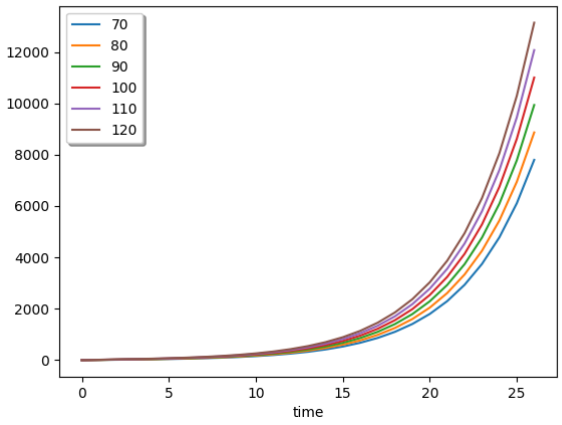

Figure 1: Instability of exponential growth

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 modelling problems exhibit this type of behaviour. At first, the problem is unstable but as measures are implemented, which leads to exponential decay rather than growth, the problem becomes very stable. We show this in Figure 2 which is the time dependent problem. At first the solutions diverge during exponential growth, but the addition of measures such as social distancing introduces exponential decay which make them converge. Thus the measures not only save lives but also they improve the accuracy of our solutions as we are less reliant on having accurate initial values.

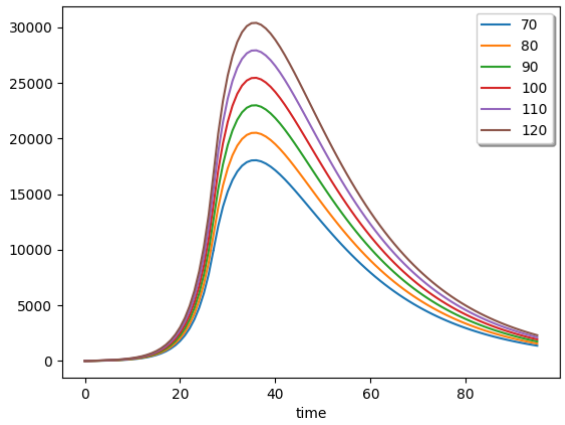

Figure 2: Gain in stability as measures are implemented

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

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 is the tolerance and the difference between fixed step size and error control solvers.

The tolerance is a measure of how accurate we want the solution of 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. This blend is what is going to be compared against error estimate which generally results in more accurate or more efficient solutions

A solver is said to have a fixed step size if it does not have error checking. 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 step it took was accurate or not. Thus, the distance between the points, i.e the step size, is constant throughout the computation. The only metric that a sharper tolerance will change is the initial step size.

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 recompute the step-size if the error estimate is significant. It will retake the step with a new smaller step size and will repeat the process until the error estimate satisfies the given tolerance. Only then will it move to the next point. Thus it constantly reduces the step-size to step through the problem. This allows it to make sure

that the given tolerance is truly satisfied over the whole problem interval.

Error control is not simple to implement. This is where we need to caution against the use of non-standard ODE solvers or other fixed step-size solvers. Also, some researchers, who have an understanding of ODEs may be tempted to write their own solvers. These researchers often program a non-error control method like a simple Euler or a 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 and give very erroneous solutions, often without even a warning that these solutions should not be trusted.

1.4.2 The packages

R packages Scientists who solve ODE models in R commonly use the deSolve package and the *ode*() function within it. *ode*() provides several numerical methods to solve a problem but we have investigated: 'lsoda', 'daspk', 'euler', 'rk4', 'ode45', 'radau', 'bdf' and 'adams'. We note that these are not all the methods provided by the *ode*() function in R but that the other methods are similar to the ones we are investigating and were thus omitted. The default method is 'lsoda' and the default tolerances are 10^{-6} for both the absolute and relative tolerance. We also note that we did not consider the other integrators in the deSolve package like $rkMethod()$, which just present other Runge-Kutta methods, and the other ones which are in-fact called by the *ode*() function itself.

The error controlled 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 and has error control.
- Using 'daspk' calls the Fortran DAE solver of the same name. This solver has error control.
- Using 'ode45' calls an implementation of Dormand-Prince (4)5 (DOPRI5). This solver has error control as it is a Runge-Kutta pair.
- Using 'radau' calls the Fortran solver RADAU5 which implements the 3-stage RADAU IIA method. This also has error control.
- Using 'bdf' calls the stiff solver inside the Fortran LSODE package which is based on a family of BDF (Backward Differentiation) methods. It has error control.
- Using 'adams' calls the non-stiff solver inside the Fortran LSODE package which is based on a family of Adam's methods. It is thus error-controlled.

The fixed step-size solvers are:

- 'euler' which uses the classical Euler method. It is a fixed step-size solver. It is an implementation in C.

- 'rk4' which uses the classical Runge-Kutta method of order 4 and thus is a fixed step-size solver.It is an implementation in C.

We will use these methods to demonstrate what happens when researchers program their own non-error controlled solvers as they usually will program similar versions of a classical Euler or a classical Runge-Kutta.

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

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.

- Using 'RK23' uses an explicit Runge-Kutta pair of order 3(2). This uses the Bogacki-Shampine pair of formulas and has error control. 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 and has error control. It is a Python implementation.
- Using 'DOP853' uses an explicit Runge-Kutta pair of order 8. The method is error controlled. It is a Python implementation.
- Using 'Radau' uses an implicit Runge-Kutta method of Radau IIA family of order 5. The error is controlled with a third-order accurate embedded formula. 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 (nonstiff) and a BDF (stiff) method as it detects stiffness. This is implemented in Fortran.

We note that all solvers in *solve_ivp*() has error control and that only 'LSODA' is using the famous Fortran package itself, the others are a Python implementation and thus might be slow.

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' which is called by not specifying any methods as the first argument. 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}. This means that "rkf" has a coarser default tolerance. We will see that this is relevant for this investigation.

- the default is 'lsoda' which 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. It has error control.
- Using 'adams' calls the non-stiff solver inside Fortran's LSODE package which is based on a family of Adam's methods. It is error-controlled.
- Using 'rk' calls an adaptive Runge-Kutta method of order 4. It has error control and uses Richardson extrapolation for the error estimation. It is implemented in Fortran in a file 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 is error-controlled and calls the Fortran rkf45.f.
- daskr????

How the packages relate We tried to find the 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 LSODE's code inside ODEPACK but Python's 'BDF' is a different implementation in Python itself.

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

R's and Python Runge Kutta 4(5) pairs are both implementations of dopri5 but they have different source code as Python implements its own. Scilab's 'rkf' is not the same pairs as it is using the Shampine and Watts Fehlberg's Runge-Kutta pair not the Dormand-Prince pairs.

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) whereas R's 'rk4' is a fixed step-size method.

R's 'Radau' and Python's 'Radau' have different source code as Python implements its own 'Radau' code while R seems to be calling the Fortran code.

1.5 Observation on implementation of solution at output points

In this section we talk about a problem that we encountered by some ODE solvers in R and Scilab when it comes to plotting. In an ideal scenario, the user's desired output points would 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 such: it will have a default initial step-size, will take a step and will adjust the step-size based on the error at the given step and will then 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 the output at specific points and these points may lie in the middle of a step. The new method to get these output points is to perform an interpolation at the given step and to return the value of that interpolant at the required point.

In R and Scilab, this new method is not used in all the solvers. Instead, R and Scilab will use the vector of output points to dictate the step-size. These solvers will thus use an initial step size or the difference between the start point and the next point as the initial step-size and step to this next point. The solver will then repeat the process between each consecutive pair of 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 need to pause at each output point. This will lead to a considerable drop in efficiency as we will show, for instance in Tables 8 and 9. These table shows that a problem that can be solved with 150 function evaluation will be solved with 500 function evaluations. This increase in the number of function evaluation will correspond to increases in CPU-time when the ODE function is complex.

This old method of implementing specific points outputs also means that the accuracy of the solver depends on the space between the desired output points. Both in an error-control and a fixed step-size solver, if the solver needs to pause between each output point the step-size in this interval has a maximum of the difference between the two points. Thus, we get the bizarre behaviour 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. We also note that spacing the points closer together is not a precise way to control the accuracy as no researcher will know beforehand how close the points should be.

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 poor accuracy but very high performance. 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 will show that R is using a lot more function evaluations than

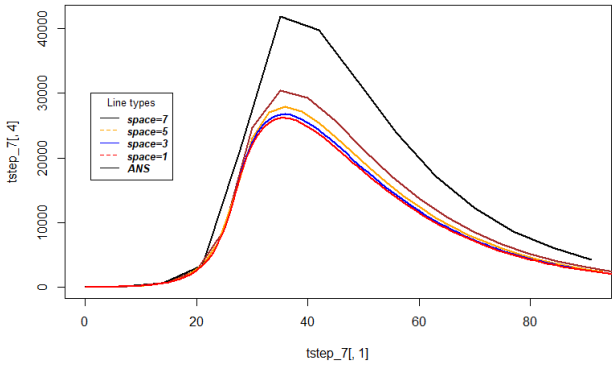

Figure 3: R 'ode45' spacing between points experiment

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 (too close to the plot of the ANS). This excess accuracy comes at a price of around 500 more function evaluations which is a problem, for instance if the user is running a parameter-estimation-like experiment. Accuracy should ideally be completely determined by the tolerance but using this old method of outputting goes against that. This results in the solvers not being allowed to take as big a step as it can.

We advise users to use an interpolation software whenever readily available like Python's *dense_output* mode in *solve_ivp* 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 making unnecessary stops. If their chosen solver is using the old method, we advise using a different solver whenever possible or running their solvers appropriately spaced out for a set of points and running an explicit interpolation on its output to calculate a smooth curve.

1.6 Discontinuities and their effects on solvers

The main purpose of this paper is to discuss discontinuities and how these affect our models. In this section, we will show what happens when a solver meets discontinuities and how these lead to erroneous solutions.

We need to understand that all the solvers use Mathematical theories based on Taylor Series, one of the core assumption of which is that the functions and all of their relevant derivatives are continuous. If a function is discontinuous, these theories no longer hold and the solvers are no longer guaranteed to converge to the actual solution as the step-size is tend 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 extremely sharp tolerance to step over them and that fixed step solvers simply cannot solve these problems as they do not reduce the step size accordingly.

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 need to be much smaller than the one used over a continuous subinterval. The codes will thus have retake the step with a smaller step size and as long as the error estimate is not small enough, it will need to continue retaking the step. This leads to high numbers of function evaluations near the discontinuity which when we have a complex problem will lead to longer CPU times.

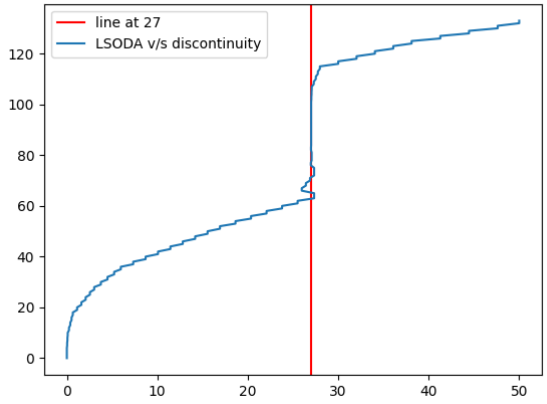

Figure 4: LSODA against a discontinuity

In Figures 4 and 5, we run 'LSODA' and 'DOP853' on a discontinuous problem where the discontinuity is 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.

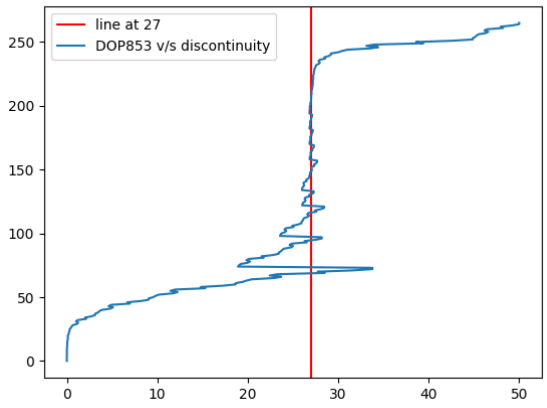

Figure 5: DOP853 against a discontinuity

Following this discussion, we also recommend epidemiologists to 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:

```
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.

VI ========== ADD A SECTION ON DISCONTINUITY DETECTION ALGORITHM. OR REFERENCES. A good advice that I will add in my conclusion is to always use a discontinuity detection algorithm if there is if-statements in the ODE function or the epidemiologist suspects a discontinuity. ========== VI

2 Time Dependent discontinuity problem

In the time dependent disconinuity problem, we change the value of the parameter β from 0.9 to 0.005 at time 27. This introduces a discontinuity in the problem. We will show that this leads to inaccuracies, 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 problems as outlined in Section 1.6.

For this report, we will add a time-dependent discontinuity by using an if statement at time 27, where the parameter β will change from 0.9 to 0.005, indicating that measures are implemented. In pseudo code, this looks like:

```
function model_with_if(t, y)
    (S, E, I, R) = y
    beta = 0.005
    if t < 27:
        beta = 0.9

    // code to get (dSdt, dEdt, dIdt, dRdt)
    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 investigate in Section 2.3.

2.1.1 Time discontinuity in R

From Figure 6, we can see that all the methods except 'euler' and 'rk4' are on the same line. 'rk4' is somehow close to the actual solution but 'euler' was completely wrong. We note that all the other methods had 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 prove in another experiment that this is entirely because of the outputting problem discussed in Section 1.5. In fact, if we use a bigger step-size, 'rk4' gives results which are as bad as 'euler'. Figure 7 shows that experiment with 'rk4' at different initial step-size (space between the output points) plotted against a graphically 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. Analysing the source for 'rk4' and 'euler' shows that it picks the step size using the output points

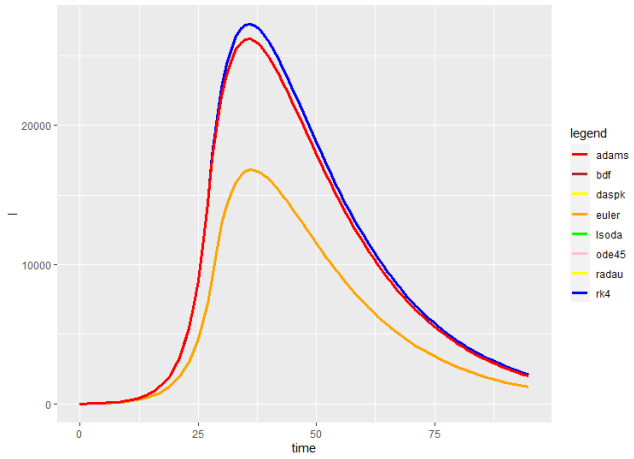

Figure 6: Time Discontinuity in R

requested. Spacing out the output points affect the step-size which affects the accuracy of the fixed step-size solver.

A solution to this problem of wanting to use 'rk4', 'euler' or any user-programmed solver would be to have a small initial step-size but we cannot know beforehand how small is small enough. A better solution would be to not use fixed-step size solvers, be it from the packages or user-implemented ones. Reputable methods with error control should be preferred as we have shown that these solvers can step over one discontinuity by resizing the step repeatedly, as we have seen is needed in Section 1.6.

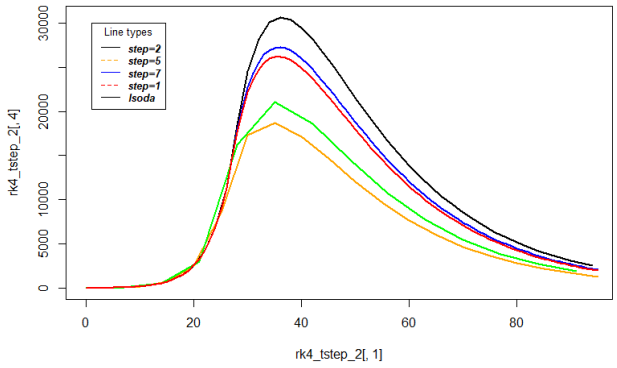

Figure 7: R's rk4 with bigger step-size

2.1.2 Time discontinuity in Python

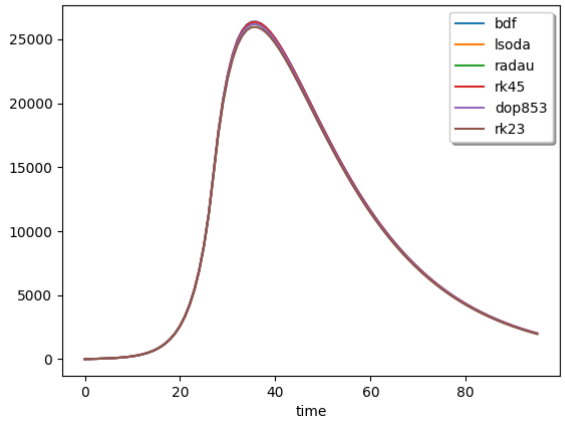

Figure 8: Time Discontinuity in Python

From Figure 8, we can see that all the methods in Python's *solve_ivp*() work correctly. There is some blurring at the peak but all the methods are along the same line. Python only provides error-controlled packages and thus we can see that error-control is all that is needed to step over one discontinuity. This observation also leads us to another conclusion that sharp tolerance with an error-control solution is what is required to step over one discontinuity.

2.1.3 Time discontinuity in Scilab

From Figure 9, in Scilab, all the methods are on the same line except for 'rkf'. This is more interesting as we know that 'rkf' should also have error control. This is explained by noting that 'rkf' has smaller default absolute and relative tolerances in Scilab. We will show during a tolerance analysis in Section 2.3 that with a sharp enough tolerance, it also provides the right solution.

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

This also points to us that an error control solver with a sharp tolerance is able to step over a discontinuity.

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 so that we integrate before and after the discontinuity with different (separate) calls to the solver. Cold starting at a time dependent discontinuity improves the accuracy as we will see in this and the next section. It also improves

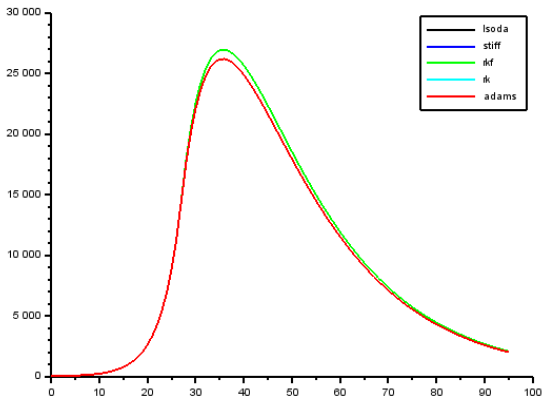

Figure 9: Time Discontinuity in Scilab

the efficiency as less 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 using new unfilled data structures with no values from previous computations corrupting the new integration. It will also use new initial step sizes and for methods of varying order like the BDF and Adams, we start anew with the default order. Each call thus do not overlap and the solver thus integrates a continuous subinterval with each call and will not have to step over a discontinuity.

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

```
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 epidemiologists for any problem where they know when the discontinuity is introduced or any time they feel that a time dependent if-statement should be added to their model.

2.2.1 Solving time discontinuity in R

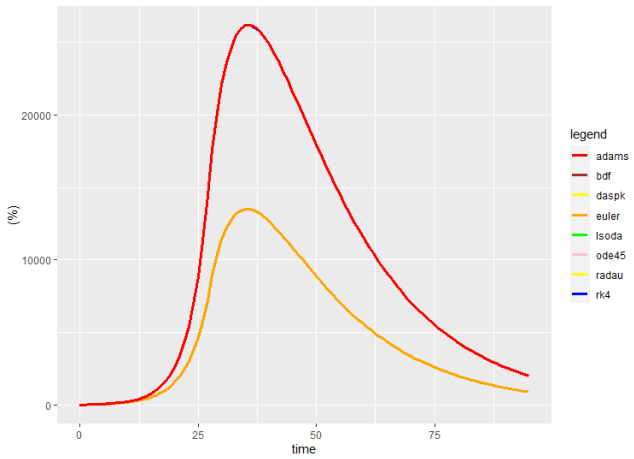

Figure 10: Solving Time Discontinuity in R

From Figure 10, the 'euler' method still fails even with the discontinuity handling. This is as expected as it has no error control and thus it still suffers from the stability issues and will require smaller steps to integrate even continuous problems.

We see that breaking the problem into two 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 well performance is still unexpected. We will show in Figure 11 that this is only due to a very small initial step size and the problem with the old method of outputting points as described in Section 1.5.

Thus our recommendation to avoid fixed step size solvers still holds as for any particular problem, researchers will never know how small the step size should be without knowing the actual solution.

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 evaluation as we discussed in 1.6. Table 2 shows that discontinuity handling reduces the number of function evaluations.

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 the drastic decreases in the number of function evaluations

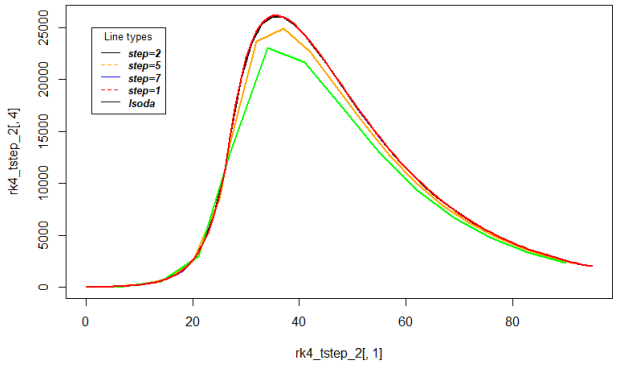

Figure 11: R's rk4 with bigger step-size with discontinuity handling

Table 2: R Time Discontinuity problem efficiency data

method	nfev	event's nfev
euler	96	97
rk4	381	382
lsoda	332	272
ode45	735	599
radau	679	585
bdf	423	263
adams	210	176
daspk	517	521

from all the remaining solvers except 'daspk'. These reductions in the number of function evaluations will have a significant impact on the CPU time if the problem were harder. This is entirely explained in 1.6 where the error controlled solvers have to repeatedly resize the step-size as they encounter a discontinuity. Not having to integrate through a discontinuity means that there is no need to perform the 'crash' into a 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.

2.2.2 Solving time discontinuity in Python

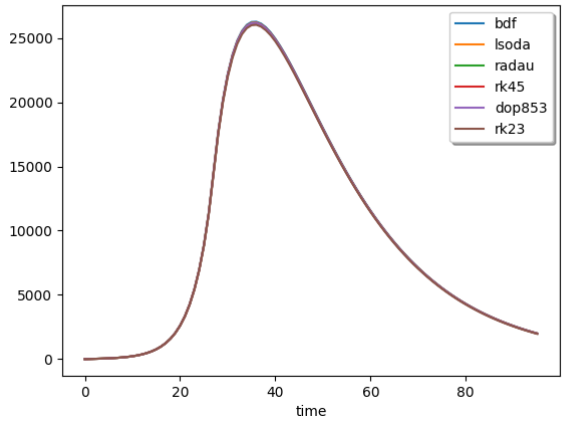

Figure 12: Solving Time Discontinuity in Python

Python did not have a problem even with the if-statement inside. This is because all the available methods use error control in Python and its default tolerances were sharp enough. From Figure 12, we can see that Python again gave the correct results. Furthermore, the slight blurring at the peak has disappeared. The addition of discontinuity handling will also drastically reduce the number of function evaluations as seen in Table 3.

Table 3: Python Time Discontinuity problem efficiency data

method	nfev	event nfev
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, Python does not seem to allow the space between the points affect the accuracy. It seems to be doing some form of local interpolation in a step. *dense_output* is used to do a global interpolation.

From Table 3, we see that across the board, the methods take less 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.

2.2.3 Solving time discontinuity in Scilab

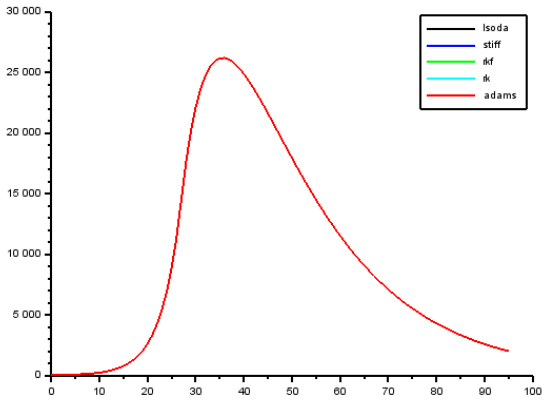

Figure 13: Solving Time Discontinuity in Scilab

We can see from Figure 13 that all the methods lie on a single line and thus the time discontinuity has been solved. The 'rkf' method is also on that line and is correctly solving the solver. This is despite the fact that 'rkf' has a coarser default tolerance. In Section 2.3, we will see that this discontinuity handling also allows us to use coarser tolerances and thus explains why the default tolerance 'rkf' is also solving the problem correctly.

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	nfev	disc. handling nfev
lsoda	346	292
stiff	531	362
rkf	589	590
rk	1649	1473
adams	304	221

From Table **??**, we see that across the board, the methods take less function evaluations. We see substantial decreases in the number of function evaluations for lsoda, stiff, rk and adams.

The odd values of 'rkf' whereby the number of function evaluations does not decrease is because 'rkf' is using the old method for outputting points as outlined in Section 1.5. The results when we space out the points more are 208 without discontinuity handling and 288 with discontinuity handling. We see that the number of function evaluations increased.

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

VI ========== rkf spaced out had number of function evaluations increased?? ========== VI

2.3 Efficiency data and tolerance study for the time discontinuous problem

It is not uncommon for researchers to use the ODE in a loop or within an optimisation algorithm so that they can get models with different parameters. In so doing, some may be tempted to coarsen the tolerances whenever the experiment they are performing is taking too long. In this Section, we investigate how coarse we can set the tolerance while keeping accurate results.

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 the discontinuity handling allow us to use coarser tolerances.

We will also investigate 'rkf' in Scilab as it has a smaller default tolerance and will prove that it can solve the problem without discontinuity handling only for sharper tolerances than its default. We also investigate Runge-Kutta pairs of the same order 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 being a Python implementation and the one in R being a Fortran implementation. We also note that R is not using DOPRI5.f but another Fortran implementation of DOPRI5.

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 having accurate results. We also look at efficiency data to see the decreases in the number of function evaluations, which would lead to significant decreases in computation times.

Time discontinuity LSODA tolerance study in R From Figures 14 and 15, we can see that the addition of discontinuity handling lets us use a coarser tolerance and still get the required answer; we need 10^{-3} and sharper tolerances without discontinuity handling but can use 10^{-2} and sharper with it. This solidifies that a form of discontinuity handling when coding a discontinuous

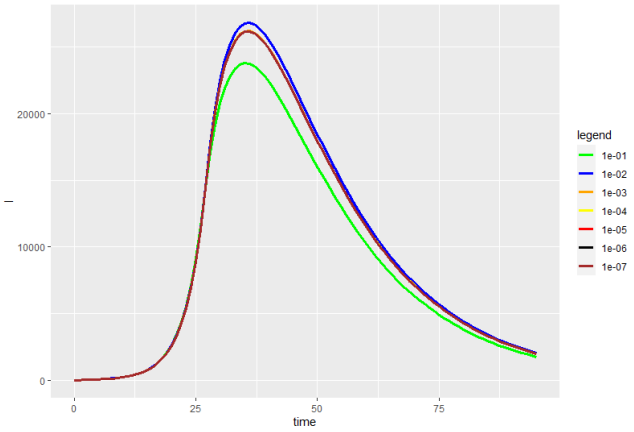

Figure 14: Time discontinuity tolerance study on R's LSODA without a cold start

problem will improve the accuracy of the solution. Also, using coarser tolerances gives us more efficiency, as we will see in Table 5. This allows researchers to coarsen the tolerances of their experiment when the latter is running too slow.

Table 5: R LSODA Time Discontinuity tolerance study

tolerance	nfev	nsteps	disc. handling nfev	disc. handling nsteps
1e-01	197	98	200	99
1e-02	214	104	206	101
1e-03	264	122	212	105
1e-04	264	123	224	111
1e-05	317	145	244	121
1e-06	332	154	272	133
1e-07	393	185	298	146

From Table 5, 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 which took longer to run, we will see clear drops in computation times.

The similar number of function evaluations for the coarser tolerances should not distract from the fact that the code without discontinuity at these tolerances are not as accurate as the code with. 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 that the solutions are wrong.

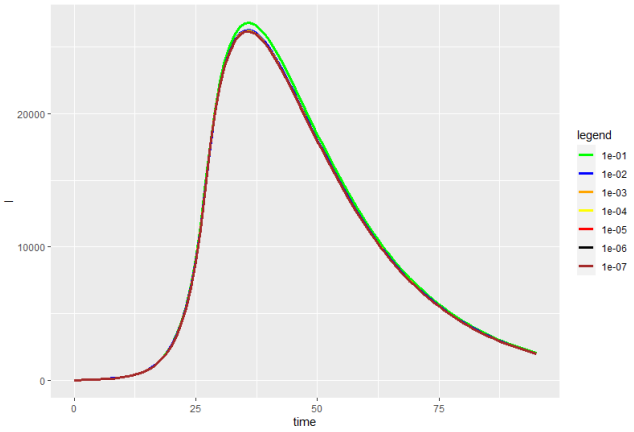

Figure 15: Time discontinuity tolerance study on R's LSODA with a cold start

Time discontinuity LSODA tolerance study in Python In this Section, we run Python's LSODA solver with multiple tolerances with and without discontinuity handling. We note that Python was working correctly in both cases apart from some blurring but we will see how coarse we can keep the tolerance while still having correct 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.

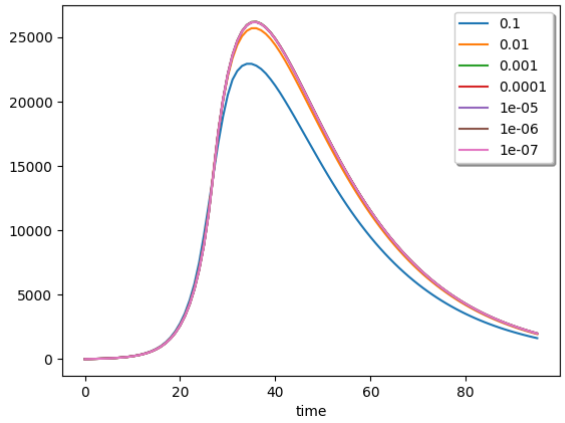

Figure 16: Time discontinuity tolerance study on Python's LSODA without a cold start

From Figures 17 and 16, the addition of the discontinuity handling lets us use a coarser tolerance, as 10^{-2} was enough to get the correct answer with the

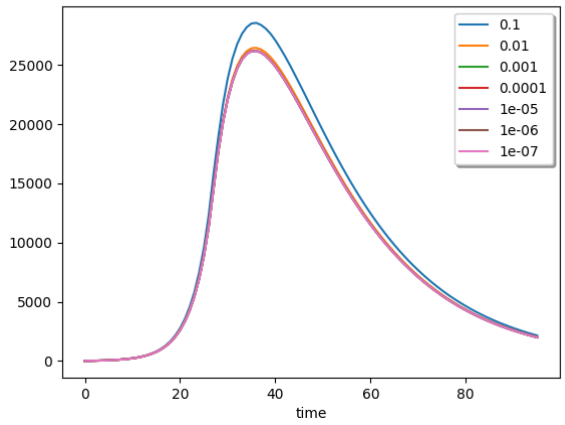

Figure 17: Time discontinuity tolerance study on Python's LSODA with a cold start

discontinuity handling whereas 10^{-3} was needed without. This essentially 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 6.)

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 6: Python LSODA Time Discontinuity tolerance study

tolerance	nfev	disc. handling nfev
0.1	79.0	86.0
0.01	98.0	93.0
0.001	156.0	116.0
0.0001	185.0	146.0
1e-05	259.0	186.0
1e-06	283.0	228.0
1e-07	361.0	272.0

Again, in Table 6, we see that that at coarse tolerances, the number of 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.

At sharp tolerance, where the comparison is fair, the number of function evaluations is much smaller with the discontinuity handling than without; we make 40 less function evaluations at 0.001 and 0.0001 but we do much less

function evaluations for sharper tolerances. We note that if the user-provided function 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 Scilab'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 getting correct results.

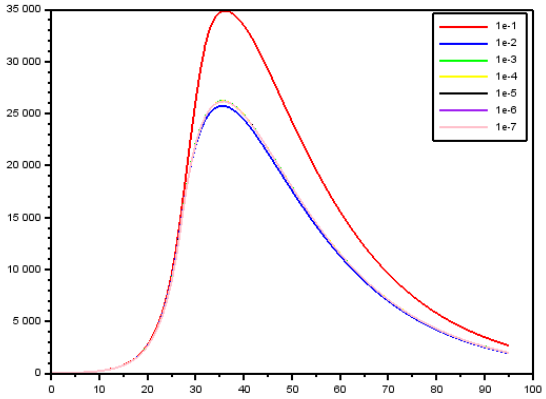

Figure 18: Time discontinuity tolerance study on Scilab's lsoda without a cold start

From Figures 18 and 19 we can see that at 10^{-1} to 10^{-4}, Scilab's LSODA without discontinuity handling fails but we seem to be able to use 10^{-3} 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 behaviour is different from R's and Python's LSODA but this may be due to the way Scilab processes the tolerances before handing it to the source code.

Again, in Table 7, we see that the number of function evaluations is roughly the same at coarser tolerances but that at sharp tolerances, where both give accurate solution and thus allow far comparison, the code with the discontinuity handling performs better than the code without. We can use up to 90 less function evaluations through discontinuity handling.

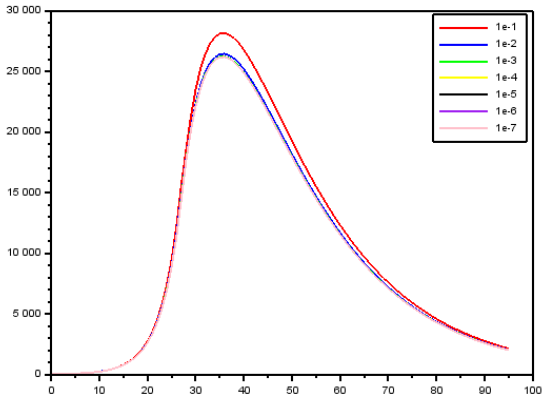

Figure 19: Time discontinuity tolerance study on Scilab's lsoda with a cold start

Table 7: Scilab LSODA Time Discontinuity tolerance study

tolerance	nfev	disc. handling nfev
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.

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

Time discontinuity tolerance study on R's version of DOPRI5 In this Section, we use R's 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 keep the tolerance while still getting correct results. We also look at efficiency data to see the decreases in the number of function evaluations.

From Figures 20 and 21, the addition of discontinuity handling lets us use a smaller tolerance and still get the required answer. Without discontinuity handling, we had to use 10^{-4} for both the absolute and relative tolerance but without, we seem to be able to use 10^{-1}.

However as we will see in the Python's version of DOPRI5, the results from

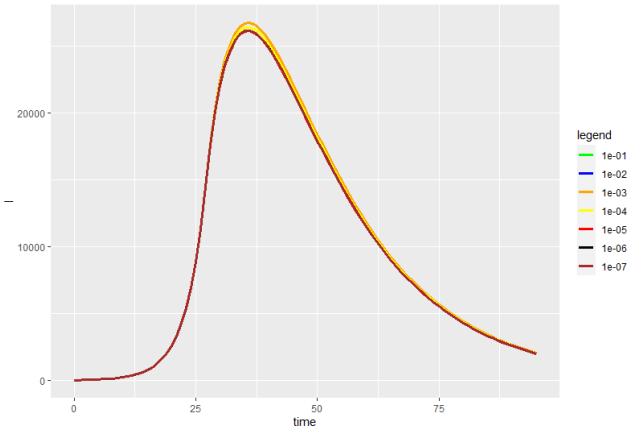

Figure 20: Time Discontinuity tolerance study on R's version of dopri5 without discontinuity handling

20 and 21 are suspicious and stem from the fact that R is not using interpolation to produce the results. It is using an old method that depends on the selected output points which affects efficiency and accuracy.

Table 8: R dopri5 Time Discontinuity tolerance study

tolerance	nfev	nsteps	disc. handling nfev	disc. handling nsteps
1e-01	572	95	574	95
1e-02	572	95	574	95
1e-03	572	95	574	95
1e-04	612	101	574	95
1e-05	692	113	587	97
1e-06	735	120	599	99
1e-07	926	150	702	116

Table 8 also confirms our suspicions as at coarser tolerances, 1e-01 to 1e-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 for the number of function evaluations and that this other factor requires 572 or 574 function evaluations.

We suspect that R's DOPRI5 version is not using interpolation or some other dense output technique to produce its solutions and that it is integrating using the sampling points. 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 other output point 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

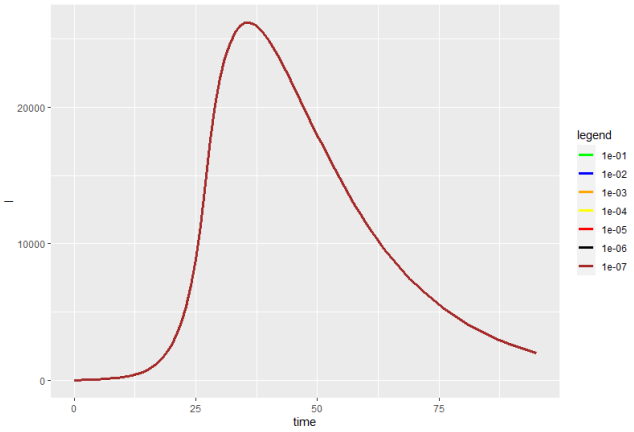

Figure 21: Time Discontinuity tolerance study on R's version of dopri5 with discontinuity handling

with the points are further away from each other and see what happens.

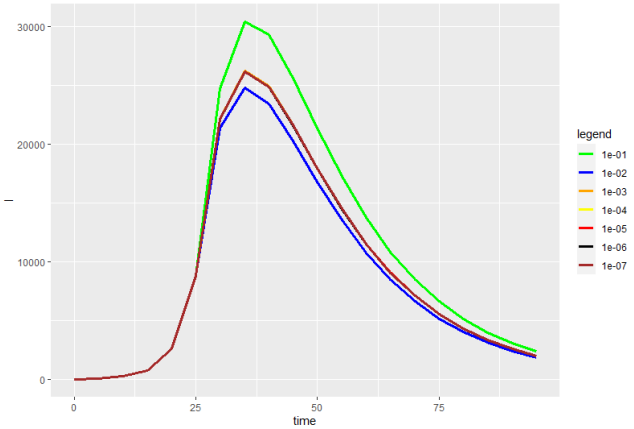

Figure 22: Time Discontinuity tolerance study on R's version of dopri5 without discontinuity handling and output points more space out

From Figures 22 and 23, we can now see a more drastic change in the solution from the codes the output points further spaced out. Also, see table 9 where we will see the number of function evaluations actually change.

Using these two figures, we also see that discontinuity handling is allowing us to use coarser tolerances. We are able to use even 10^{-1} with discontinuity handling while getting an accurate result whereas without it, we need to use 10^{-3} and sharper tolerances to get the required answer.

Our analysis of Table 9 begins by noting that the set of output points is not

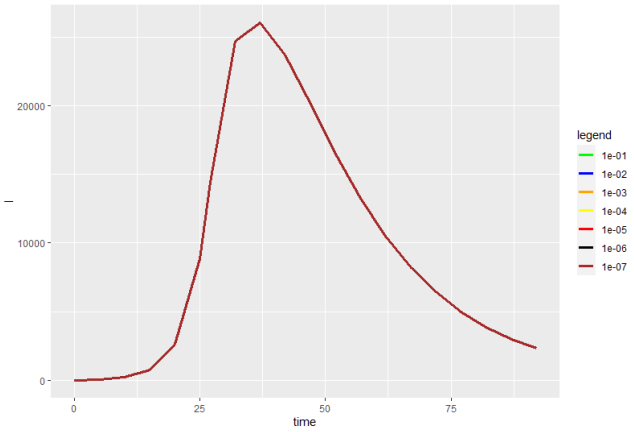

Figure 23: Time Discontinuity tolerance study on R's version of dopri5 with discontinuity handling and output points more space out

Table 9: R DOPRI5 Time Discontinuity tolerance study with spaced out points

tolerance	nfev	nsteps	disc. handling nfev	disc. handling nsteps
1e-01	116	19	112	18
1e-02	142	23	125	20
1e-03	168	27	131	21
1e-04	246	39	162	26
1e-05	352	56	235	38
1e-06	614	97	349	57
1e-07	796	128	542	89

longer a limiting factor. We can see the number of function evaluation change with the tolerance now and this indicates that the tolerance is affecting the step-size.This confirms our suspicions that R's implementation of DOPRI5 is not using a dense output mode or some form of interpolation. Instead it makes a steps to and stops at every required output points. This is the old way of giving function values at specified output points.

Regarding the accuracy of the solver as we coarsen the tolerance we can see from Figures 22 and 23 that even at 10^{-1}, the code with the discontinuity handling is still able to produce accurate solutions whereas it requires 10^{03} for the one without the discontinuity handling.

The new table, Table 9, 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 that the code without discontinuity

handling is not getting the correct answer.

Time discontinuity tolerance study on Python's version of DOPRI5 In this section, we run Python's version of DOPRI5, which is aliased 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 keep the tolerance while still having correct results. We also look at efficiency data to see the decreases in the number of function evaluations.

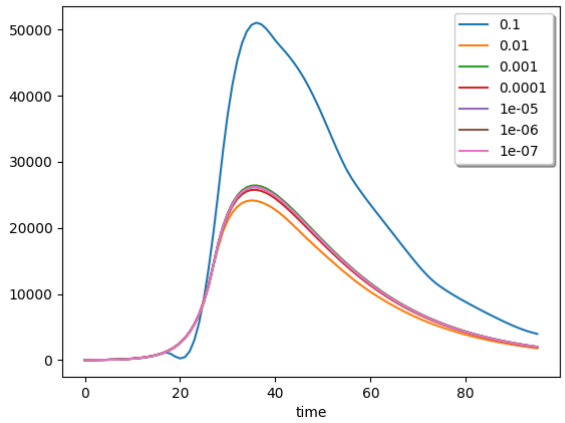

Figure 24: Time Discontinuity tolerance study on Python's version of dopri5 without discontinuity handling

From Figures 25 and 24, we can see clearer differences at the different tolerance values. This is in contrast with the first tolerance study on R's DOPRI5. From studying Python's *solve_ivp* interface and source code, we note that Python is definitely using dense output/interpolation. This concludes the discussion on R's DOPRI5 and we can explain R's DOPRI4 performance entirely because it does not use interpolation by default but instead stops at every output point.

We then compare Python's DOPRI5 with and without discontinuity handling. We can see that the use of discontinuity handling allowed us to use coarser tolerances in Python while keeping accurate results. We see that we need 10^{-5} and sharper to get the correct solutions without discontinuity handling in Python while 10^{-2} was enough with discontinuity handling. We will also see in Table 10 that the discontinuity handling code is much more efficient as well.

From Table 10, we see that at coarser tolerances, the number of function evaluations is lower in Python without the discontinuity handling than wit.

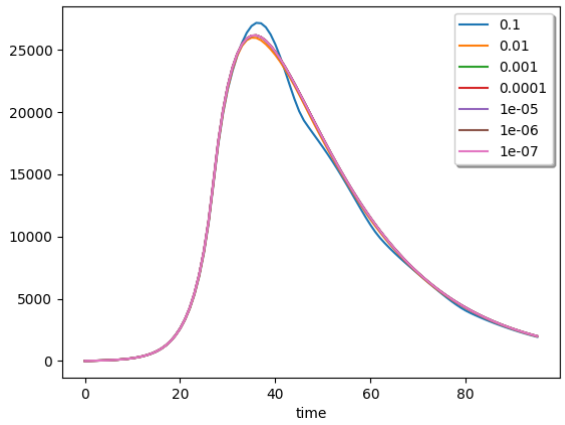

Figure 25: Time Discontinuity tolerance study on Python's version of dopri5 with discontinuity handling

Table 10: Python DOPRI5 Time Discontinuity tolerance study

tolerance	nfev	disc. handling nfev
0.1	68.0	70.0
0.01	86.0	88.0
0.001	146.0	124.0
0.0001	224.0	172.0
1e-05	326.0	250.0
1e-06	488.0	370.0
1e-07	752.0	568.0

But 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 tolerance where we get 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 10^{-7}, the drop in the number of function evaluations is very significant and would lead to much faster execution times whereas at 10^{-5} and sharper, the decrease in the number of function evaluations is 75 or more.

Time discontinuity tolerance study on Scilab's version of RKF45
In this Section we run Scilab's RKF45 aliased as 'rkf' in the *ode*() function with different tolerances. We note that the default tolerance for Scilab's 'rkf' was not enough to solve the problem 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 come up with the correct solutions at sharper tolerances without a discontinuity. Thus the anomaly we saw in section 2.1 was entirely because the solver has a coarser default tolerance in Scilab than the other methods.

We will also see that using the discontinuity handling lets us use less function evaluations which, given a more complex problem, will be a significant improvement in computation times.

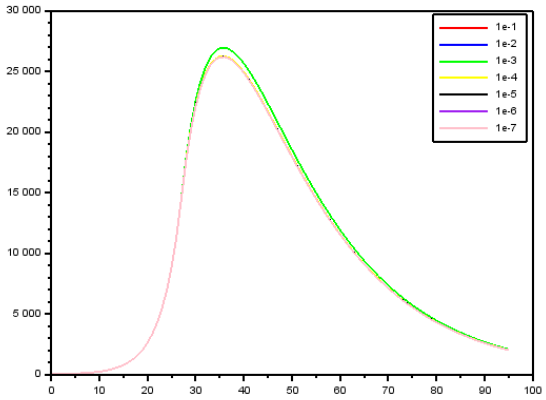

Figure 26: Time Discontinuity tolerance study on Scilab's RKF45 without discontinuity handling

We see from Figure 26 that 10^{-4} for both the absolute and the relative tolerance give 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 was needed to get this correct solution.

Figure 27 is also interesting as it seems to indicate that a 10^{-1} is enough to get the correct solution with discontinuity handling. This is also surprising but is consistent with our observations in R and Python's Runge-Kutta pairs.

We can see from table 11 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 evaluation. There are 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 of function evaluation. Doing the same experiment with the points further spaced out shows us that it was the spacing that was a problem.

We start by replicating the experiments in the previous sections

Figures 28 and 29 shows a clearer distinction in why discontinuity handling is important. We can see that without, we need a tolerance of tolerance of 10^{-3}

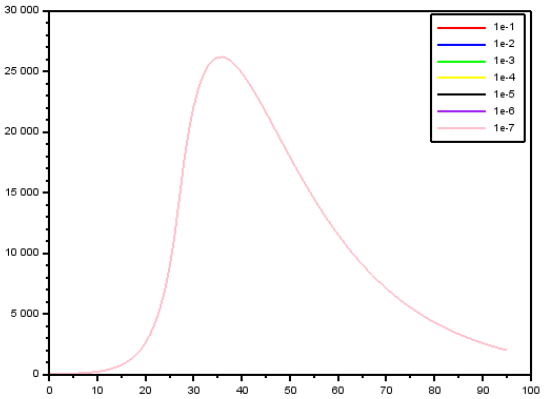

Figure 27: Time Discontinuity tolerance study on Scilab's RKF45 with discontinuity handling

Table 11: Scilab RKF45 Time Discontinuity tolerance study

tolerance	nfev	disc. handling nfev
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.

to get accurate results but with the discontinuity handling, we can use 10^{-1}. We note that such coarse tolerance may mean that we still did not space the points enough but the number of function evaluation study, shown in Table 12, is clear.

Table 12 shows how the number of function evaluation with discontinuity handling is lesser. We also note that at coarse tolerance the number of function evaluation are similar but that at those tolerances, the code without discontinuity handling was not obtaining the correct results. We can thus conclude that using discontinuity handling lets us use lower tolerances and less number of function evaluations while improving the accuracy.

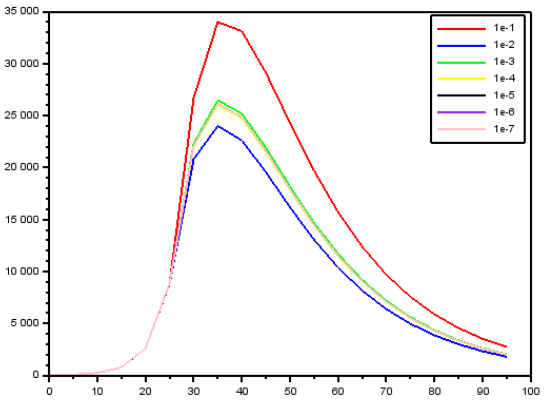

Figure 28: Time Discontinuity tolerance study on Scilab's RKF45 without discontinuity handling

Table 12: Scilab RKF45 Spaced Out Time Discontinuity tolerance study

tolerance	nfev	disc. handling nfev
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.

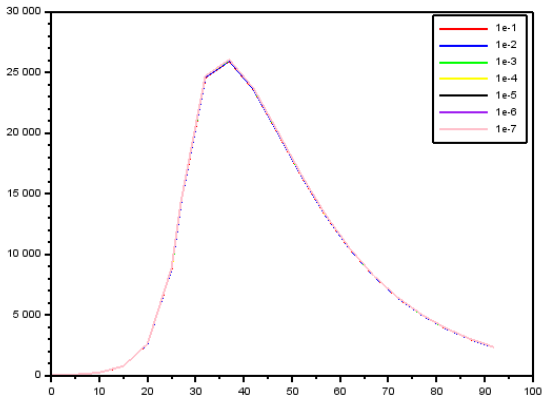

Figure 29: Time Discontinuity tolerance study on Scilab's RKF45 with discontinuity handling

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 using the discontinuity handling used in the previous problem as we do not when the discontinuity will happen. This problem also has more discontinuity than the previous problem and as such is a harder problem that even error-controlled solvers will not be able to solve. We will also show that it cannot be solved simply by sharpening the tolerance and using an error-control solver as it was the case in the previous problem.

As in Section 2, changes in the modelling parameter β introduce discontinuities in the function $f(t, y)$ and thus some solvers will crash when trying to solve them. This problem is harder than the time dependent discontinuity in that there are more discontinuities because the parameter will be toggled more than once as we solve for a longer time period.

This problem will also allow longer term forecasts and will model the 'waves' of the pandemic. It uses a state variable, the number of Exposed people, E, to change the parameter β. When the number of exposed people is greater or equal to 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 government will relax the measures again and β becomes 0.9 again. We run this over a longer time period to see several waves of the virus.

VI ========== We also note that the problem is unstable when β is 0.9 but is stable when *beta* is 0.005. ========== VI

Because we do not know beforehand at what times the discontinuity will occur, this problem cannot be solved by using cold starts as we have done before. We thus start off with a naive treatment of the problem with if-statements inside the function and show how this will never lead to the correct solution. We proceed to show how the problem cannot be solved even at sharp tolerance and finally we will introduce a way to efficiently and accurately solve it using event detection.

For the sections on the state dependent problem, we graph the number of Exposed people, E, rather than the number of infected people, I. This is because we are toggling the parameter β based on the value of E and graphing E will reflect that.

3.1 Naive treatment of Covid-19 state discontinuity model

The naive treatment of this problem is to use global variables or parameters for whether measures are implemented or not and to toggle them 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 which will choose the value of parameter β based on whether measures are implemented. We will show that this naive im-

plementation will lead to huge discrepancies in the solvers' results. The pseudo-code for this problem thus looks as such:

```
measures_implemented = False
direction = "up"
function model_with_if(_, y):
    (S, E, I, R) = y
    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
    // code to get (dSdt, dEdt, dIdt, dRdt)
    return (dSdt, dEdt, dIdt, dRdt)
```

3.1.1 State Discontinuity in R

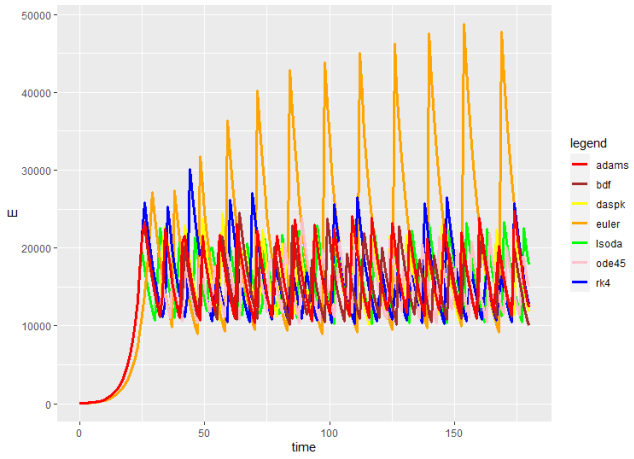

Figure 30: State Discontinuity in R

Figure 30 shows how difficult this problem is with a naive treatment. We note that none of the solutions are aligned and also that none of the solvers

has the correct solution that we will introduce in Section 3.4. We note that all the solvers, even the error controlled ones, did not issue a warning or any flag about the integration and thus researchers may be tempted to think that any of them is accurate. In solving this problem thus, the solvers all had their error estimation algorithms wrongly satisfy the tolerance.

As we are modelling E and not I, we expect that the graph goes 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 that have error control to repeatedly reduce the step-size to satisfy the tolerance and align with each others but Figure 30 shows that these solvers did not perform enough error control; their tolerances were satisfied at wrong answers and thus the error estimation algorithm did not work as expected.

We also note that the result for 'euler' is especially bad as it even reaches 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 it reach 30000 in its third peak. This is all despite the fact that the space between the points is as small as it was when performing the first experiment.

Another important fact to note is how poorly 'radau', as shown in Figure 31, 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. We will discuss our findings on 'radau' in Section 4.

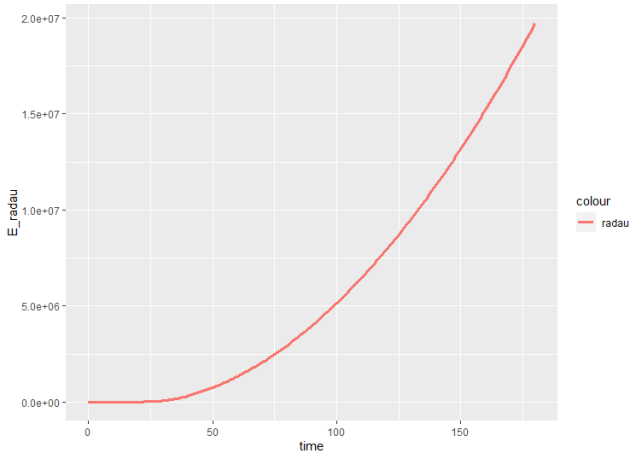

Figure 31: State Discontinuity of Radau in R

We then proceed to show that sharp tolerances and thus more stringent error control are not enough to solve this problem as it was for the time dependent discontinuity problem. We repeated the experiment at the sharpest tolerance usable before some of the solvers cannot integrate which was at 10^{-13}. We set both the absolute and relative tolerance to that number and the results are shown in Figure 32.

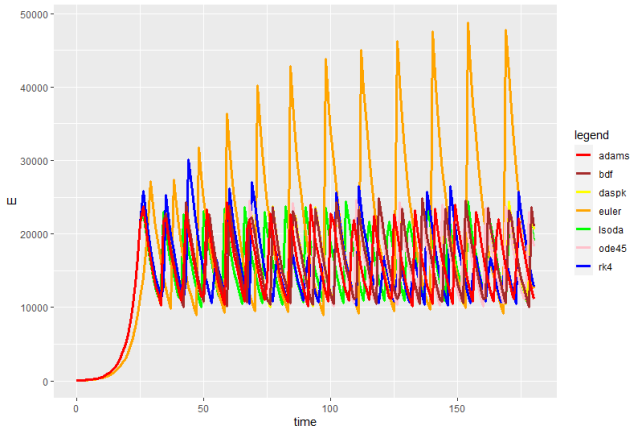

Figure 32: State Discontinuity in R at high tolerances

We can see from Figure 32 that the situation has only marginally improved. None of the solvers are on the same line 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 are along the same line showing that error-control might be able to step over one state dependent discontinuity.

The fixed step-size method 'euler' still gets results that are far off and 'rk4' is far off at the start but follows a decent pattern at the end. This surprisingly good performance of 'rk4' is again due to the fact that it has a small initial step-size as its step-size depends on the set of output points.

We can also point out that at such sharp tolerances, 'radau' is no longer having the abnormal behaviour we saw previously. From Figure 33, we can see that it oscillates between 10000 and 25000 as it should but this is still not quite the correct answer as we will discuss in Section 3.4. From supplementary experiments 'radau' starts performing correctly at about a tolerance of 10^{-9}.

3.1.2 State discontinuity in Python

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

Figure 34 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 despite the fact that all solvers in Python have error control.

We note that all the solvers expect 'RK23' at least oscillate between 10000 and 25000 though in completely random patterns. They have their peak and troughs at different times and no warnings or flags were set showing that the solvers' error estimation algorithm were wrongly satisfied with these results.

The 'RK23' solver, in purple, has a completely different pattern as the other solvers. It never reaches 25000 and only oscillates between around 10000 and

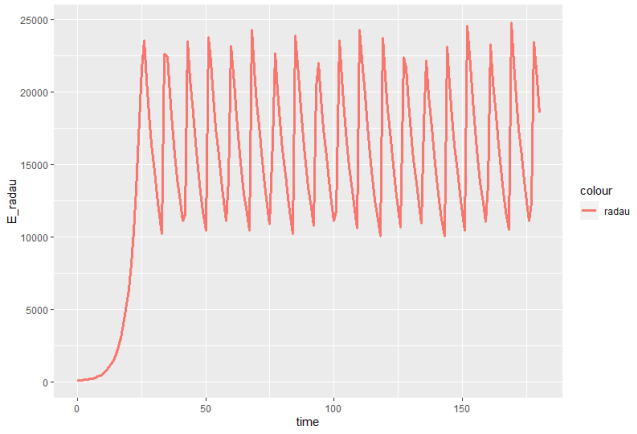

Figure 33: State Discontinuity of Radau in R at high tolerances

15000.

VI ========== I DONT KNOW WHY 'RK23' does this. ========== VI

Again, as shown in Figure 35, 'Radau' messes up completely and grows exponentially even though the parameter β should be set to start an exponential decay as it did 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 correct 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 that value and Figure 36 shows the results from this sharp tolerance experiment.

Figure 36 shows that the solution did improve. Still, the solvers are not all on the same line. 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 time 25 or 30. Though the pattern is correct, none of the solvers are on the same line telling us that none actually got the correct solution.

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 37,'Radau' stops its weird behaviour at this sharp tolerances and follows the pattern we were expecting from the solution but as we will show in Section 3.4, this is still not the correct solution. 'Radau' starts performing well at around a tolerance of 10^{-10}.

VI ========== We should also note that the R and Python implementation of 'Radau' are different. The 'Radau' solvers in Python is implemented in Python with the NumPy library whereas R uses the Fortran code. Thus we eliminate the possibility of a bug

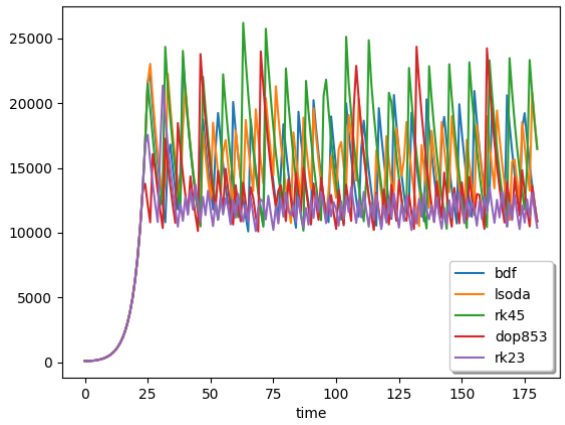

Figure 34: State Discontinuity in Python

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 Fortran, similarly bad results occur with 'Radau'. ========== VI

3.1.3 State discontinuity in Scilab

We perform the experiment with if-statements and global variables in Scilab and the results are as shown in Figures 38 and 39.

Figure 38 shows the same problem in Scilab. None of the solvers are aligned which prompts us to conclude that none of them are getting the correct solution. All of the solvers in Scilab are error controlled 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 their solutions are correct.

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 the next experiment. The sharpest tolerance we can use in Scilab before the other methods crash was 10^{-13} and the results are shown in Figure 39.

Again, in Figure 39 we can see that sharp tolerances are not enough to get the correct answers. The solution 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, they all seem reasonably on the same line but as we go further in time, all of the solutions diverge. We also note that none of them got the correct solution discussed in Section 3.4.

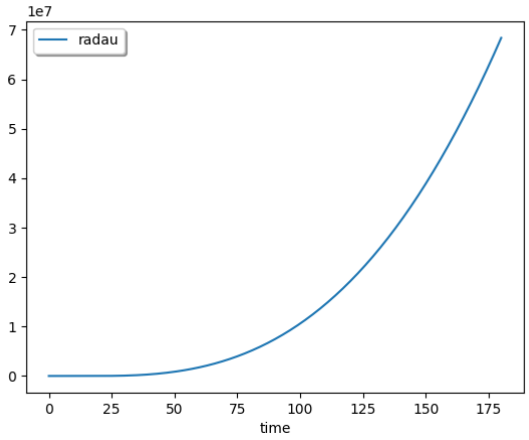

Figure 35: State Discontinuity of Radau in Python

3.2 Why even sharp tolerances failed

In this Section we discuss why sharp tolerances cannot solve the problem in the naive way it is coded, i.e using global variables and if-statements.

This is because whenever there is a change in the value of β, the step that first encounters that change will always fail. As discussed in Section 1.6, the step size at a discontinuity will always have to be much smaller than a step on a continuous region to be able to step over the discontinuity. Thus this first encounter will almost always be through too large a step and will fail. But in performing this step, the value of the E inside the function will cross the threshold, especially if the method is using several stages, the global variables will thus be toggled as the codes inside the if-statement will run. But then when the solver will attempt to retake the step again and again, it will be using the wrong β value.

This observation is crucial as it allows us to conclude that at a discontinuity some of the function evaluations along the step should be using the old β value while the others should be using the new β value. There are no trivial way to code this behaviour in the ODE function, $f(t, y)$, if we do not know the time of the discontinuity.

At extremely sharp tolerances, even beyond 10^{-12}, the first step that encounters the discontinuity can also fail. The solver will still have to retake the step but as shown, it will have to use wrong β as there are no trivial way to code that some stages in that step should use the old one while some should use the new one. In the next sections, we will present the correct way to code problems with state dependent discontinuities so that we get the required behaviour.

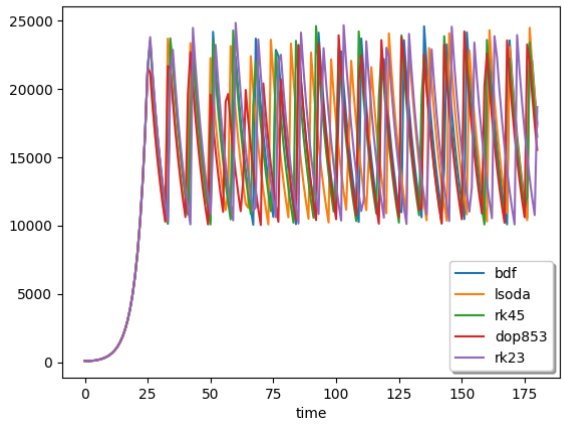

Figure 36: State Discontinuity in Python at sharp tolerances

3.3 Introducing event detection

In the time dependent 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 with just one discontinuity, the problem could have been solved by sharper tolerances also but we showed that this was not the most efficient way to do so. In the previous section we showed why even sharp tolerances will not be able to solve this problem. However, the idea that we developed in Section 2.2 about integrating continuous sub-problems of the ODE separately and combining them into a final solution can still be applied here.

To integrate continuous sub-problems, we only need a way to detect that the thresholds have been met and as soon as we see a threshold, we would 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 in the next Section, we will show how to encode the given thresholds as events to perform a cold start when they are reached.

An event in numerical ODE occurs whenever the solver detects a root of the function it is solving for. The solver will require two functions from the user as a result: the usual ODE right-hand side function, $f(t, y)$ and another function that defines how to look for the roots which we will call the root-function (commonly denoted by $g(t, y)$).

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 solver calls the root-function at each step that it takes and will record its value. It will then compare the value of the root-function with the immediate

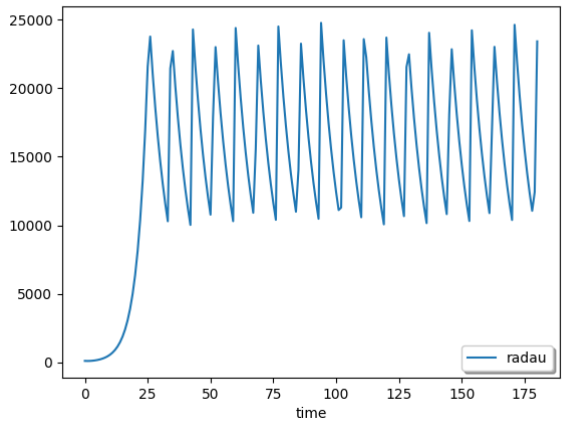

Figure 37: State Discontinuity of Radau in Python at sharp tolerances

previous value and see if there was 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 interval until it finds the exact point where the root-function returns zero. Most solvers will return the solution from that root and stop the integration, allowing us to cold start.

To use event detection thus entails defining a function that takes the value of the ODE solution at the current point and return a real number which is zero whenever we want it to detect an event. If we want to detect whenever x is 100, it is sufficient to return (x - 100) from the root-function. The next section will elaborate on how to use event detection for the state dependent discontinuity problem.

3.4 Solving State discontinuity 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 previously reported erroneous solution. We have seen that though sharp tolerances give better solutions, none of the solvers were along the same line and so the results were still wrong. Such sharp tolerances come with inefficiencies as well. We will now present a solution using event detection that is both accurate and efficient.

The solution is to use the thresholds that we have defined in our model as events and integrate only up to the first threshold we meet with each call to the solver. We can then cold start from there and repeat the process with another right hand side function denoting the new model and with another root-function which encodes the next threshold we are looking for. As this new call meets its threshold, we repeat the process using the first ODE and root functions. We

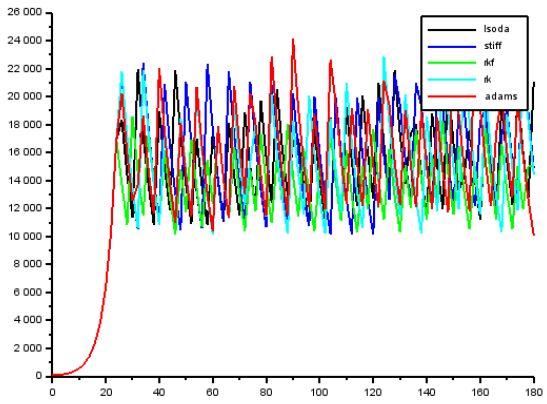

Figure 38: State Discontinuity in Scilab

repeat this until we reach the end of the time interval. This solve continuous sub-problems one at a time and these sub-problems can then combined into a final solution. This strategy thus both avoid discontinuities and allows us to 'step' over the discontinuity with the function calls before it using the old parameters and the function calls after it using the new parameters.

For our specific problem, event detection is used as such: We start by solving the problem with β at 0.9 and with a root function that detects 25000. Once we detect 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 at 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.

The pseudo-code is as such:

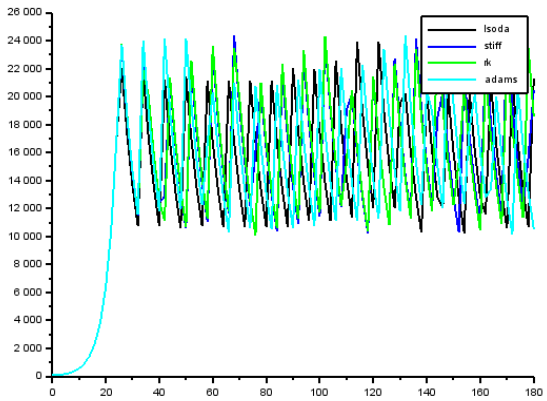

Figure 39: State Discontinuity in Scilab with sharp tolerances

```
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 = initial_values
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
```

Some programming environments, such as Python, do not stop the integration as the first event is detected by default. To do a cold start, we need them to stop at events and thus when programming, we need to set the appropriate flags. (In Python, it is done by setting the terminal flag of the root functions. Example: $root\_10000.terminal = True$)

Each call to the solvers will have a continuous problem and thus will satisfy the requirements for their underlying theories. We will show that this is enough to get high accuracy efficient forecasts.

3.4.1 Solving state discontinuity in R

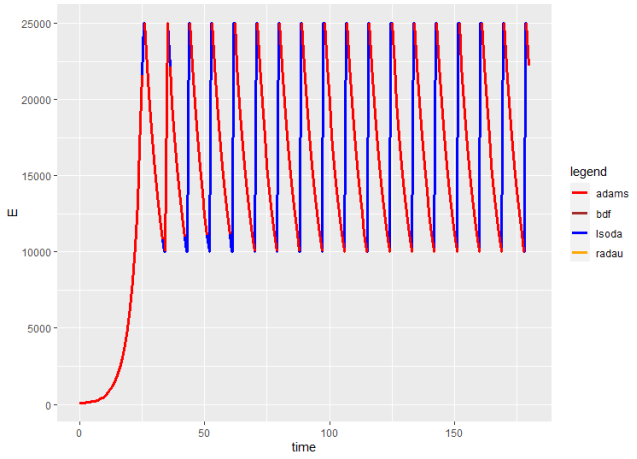

Figure 40: Solving state Discontinuity in R

We note that only a few solvers have event detection capabilities in R. These are: adams, bdf, lsoda, radau. From Figure 40, we can see that all the solvers give solutions that are on the same line. This is in contrast with what happened previously when we were integrating a discontinuous problem, even at sharp tolerances. We will show in Table 13 that introducing event detection also made the solvers significantly more efficient.

We note that it is unfair to compare the efficiency at the default tolerances to the event detection's efficiency as their results were completely wrong. We can only compare the values with the sharp tolerance values.

VI ========== Would you prefer that I add the default tolerance values. Some solvers used less function evaluation even with event detection than even the default tolerance experiment??? ========== VI

We can see from Table 13 that even with sharp tolerances, we had inaccurate results and this required significantly more function evaluations. We note that we are gaining around 3000 function evaluations in 'lsoda', 20000 in 'radau', 8000 in 'bdf' and 2500 in 'adams' while having more accuracy. This significant

Table 13: R State Discontinuity problem

method	nfev	nfev sharp	event nfev
lsoda	2135	4658	1226
daspk	5143	15185	NaN
euler	181	181	NaN
rk4	721	721	NaN
ode45	2027	18246	NaN
radau	1002	21835	2232
bdf	3300	9803	1657
adams	1368	3467	802

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 a more complex function.

3.4.2 Solving state discontinuity in Python

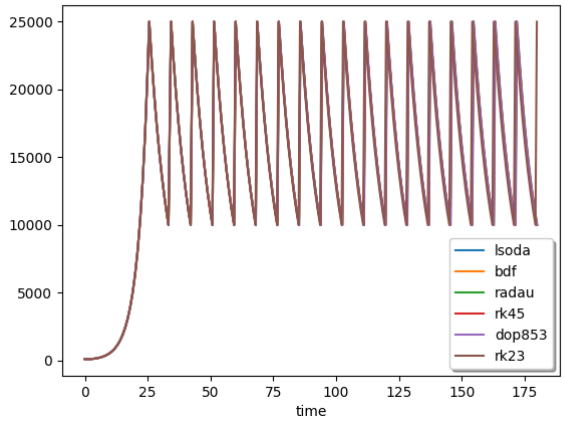

Figure 41: Solving state Discontinuity in Python

All the solvers in Python have event detection. Again, Figure 41 shows that all the solvers give solutions that are on the same line, prompting that this is the correct solution. This is different from our results when integrating with a single *solve_ivp*() call. We will also see that this is much more efficient across all the solvers.

As in the case with R, we cannot compare the default tolerance efficiency data to the event detection efficiency data as the former clearly had worse

results. So, in table 14, we compare the sharp tolerance efficiency data with the data from the event detection.

VI ========== Would you prefer that I add the default tolerance values. Some solvers used less function evaluation even with event detection than even the default tolerance experiment??? ========== VI

Table 14: Python State Discontinuity problem

method	nfev	nfev sharp	event nfev
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 14 shows that the number of function evaluations using event detection is far less; 'lsoda' used around 3000 less function evaluations, 'bdf' used 11000 less, radau used 74000 less, rk45 used 17000 less, 'dop853' used 20000 less and 'rk23' used 246000 less function evaluations. The reduction in CPU times from this will be significant across all the solvers, especially with a more complex right hand side function.

3.4.3 Solving state discontinuity in Scilab

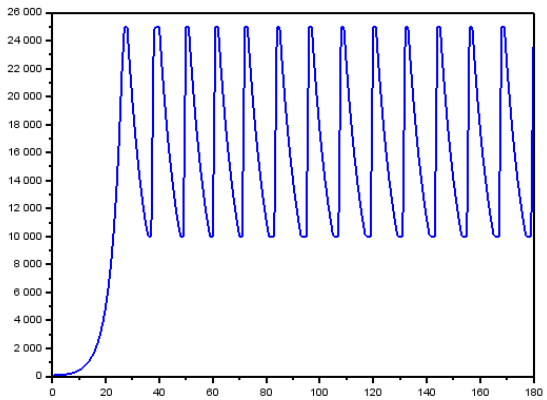

Figure 42: Solving state Discontinuity in Scilab

There is only one solver with root functionality in Scilab which is the 'lsodar', the root-finding version of 'lsoda'. We do not get to compare it with other solvers but judging from the solutions we got from Python and R, it seems that it gave a correct solution as well. It is oscillating in the correct pattern and goes sharply between 10000 and 25000.

Table 15: ScilabState Discontinuity problem

method	nfev	nfev sharp	event nfev
lsoda	2794.	4636.	1327.
stiff	3476.	9797.	—
rkf	3004.	—	—
rk	12131.	45940.	—
adams	2979.	5694.	—

VI ========== do you want me to add daskr ========== VI

3.5 Efficiency data and tolerance Study for the state discontinuous problem

We can see that default tolerances for the model without event detection does not give correct answers. In this section, we will investigate how sharpening the tolerance improves the results in the case of the non-event detection experiment while we coarsen the tolerance with the event detection code to show how coarse a tolerance we can use while getting good 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's and Python's version of DOPRI5 which does not use the same code but do use the same Runge-Kutta pair and Scilab's version of RKF45 which is not the same code, nor the same pair but is a Runge-Kutta pair of the same order.

3.5.1 Comparing LSODA across platforms for state discontinuous problem

In this section, we use R's version of LSODA at multiple tolerances to see how it acts at multiple tolerances. We set both the relative and the absolute tolerance to a particular value and analyse the solution.

We know that without event detection, LSODA does not work even at very sharp tolerances but we report on any strange behaviour from the code. We also would like to know how coarse we can set the tolerance to still have the event detection code work.

state discontinuity LSODA tolerance study in R Figure 43 shows a very strange behaviour. The same code with the same problem at different

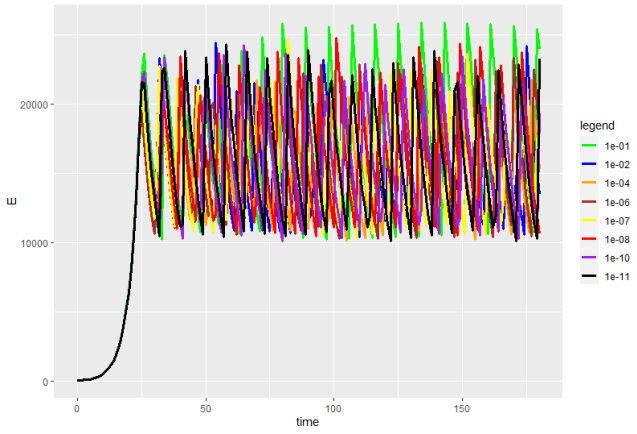

Figure 43: State discontinuity tolerance study on R's LSODA without event detection

tolerances is giving vastly different results. We would expect the solutions at the sharper tolerances to be along very similar curves but that is not the case. We conclude that LSODA even at sharp tolerances is still resizing the step size. It is also suffering from the fact that the first step that encounters a discontinuity fails while still switching the global variables. This further proves our statement that for any state-dependent discontinuity, we cannot get the correct results simply by sharpening the tolerance.

From Figures 44 and 43, we can see the clear advantage of using event detection. Event detection even allows us to use very coarse tolerances while solving the problem correctly. Event detection allows us to use 10^{-3} and sharper to get the correct results while the code without event detection still failed at 10^{-13}. We will also analyse the differences in efficiency between the two codes in Table $reftab : tolerance_state_discontinuity_lsoda_R$.

Table 16: R lsoda State Discontinuity tolerance study

tolerance	nfev	nsteps	event nfev	event nsteps
1e-01	675	274	552	233
1e-02	1856	727	512	236
1e-04	1863	706	736	350
1e-06	2135	810	1226	559
1e-07	2676	1005	1832	830
1e-08	2730	1025	2032	911
1e-10	3337	1392	2546	1212
1e-11	3603	1566	2997	1470

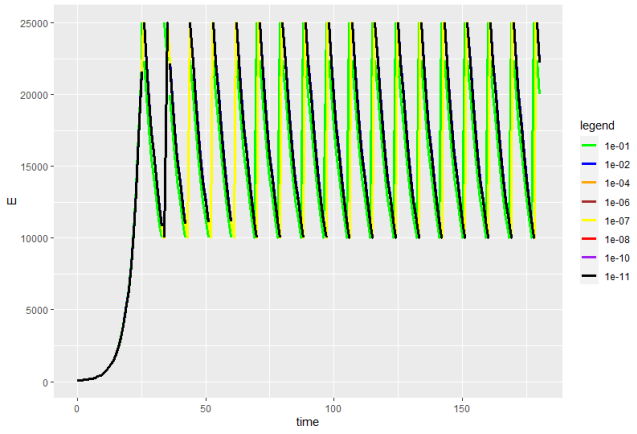

Figure 44: State discontinuity tolerance study on R's LSODA with event detection

Table 16 shows a decrease in the number of function evaluations 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 code without event detection did not give a correct answer. However, it gave this wrong answer while still using more function evaluations which adds on to our conclusion that event detection is the correct way to solve state dependent discontinuity problems.

discontinuity LSODA tolerance study in Python In this section, we use Python's version of LSODA at multiple tolerances to see how it performs. We set both the absolute and relative tolerance to a particular value.

We note that LSODA without event detection even at very sharp tolerances in Python was still failing 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 code with event detection as each call to the solver is executing on continuous sub-intervals.

Again Figure 45 exposes the strange behaviour of LSODA whereby the same code with the same problem at different tolerances give different results. We would expect the code at the sharper tolerances to give very similar curves but clearly, LSODA even at sharp tolerances is still resizing the step size. 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 by sharpening the tolerance.

From Figures 46 and 45, we can see that the addition of the event detection lets us use a smaller tolerance. We also note that the code with event detection blur as we go further in time. This is because the coarser tolerances are not giving the actual solution. In Python, it is at 10^{-4} and sharper that we are

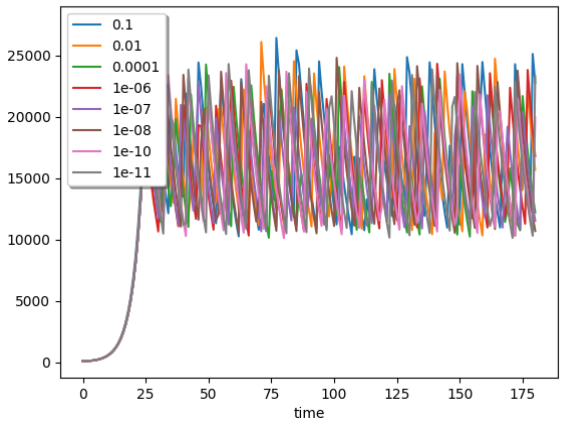

Figure 45: State discontinuity tolerance study on Python's LSODA without event detection

getting correct results.

We then analyse the efficiency in Table 17. We must note that this analysis is unfair as the code without event detection does not actually solve the problem. Still, we will see that the event detection code uses less function evaluations while getting the better answer.

Table 17: Python LSODA State Discontinuity tolerance study

tolerance	nfev	event nfev
0.1	1207.0	425.0
0.01	1627.0	454.0
0.0001	1968.0	689.0
1e-06	2122.0	1305.0
1e-07	2684.0	1807.0
1e-08	2730.0	2099.0
1e-10	3337.0	2639.0
1e-11	3603.0	3098.0

Again we see in table 17, that the code without event detection give us wrong answers while taking more function evaluations. A problem with a more complex right hand side function will take far less computational time with event detection than without.

discontinuity LSODA tolerance study in Scilab We perform the same experiment in Scilab. We set the absolute and relative tolerance to the

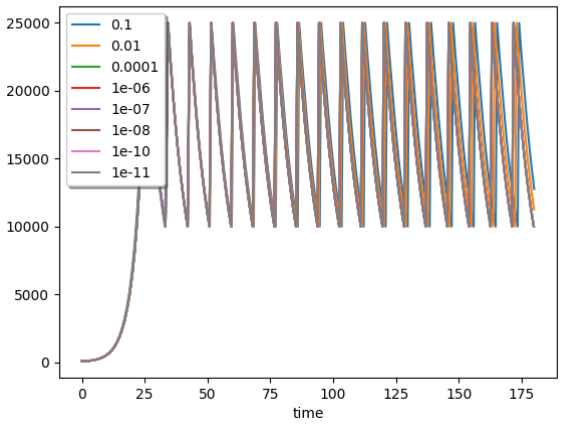

Figure 46: State discontinuity tolerance study on Python's LSODA with event detection

same particular value and run the solvers. For the different tolerance values, we plot the solutions and analyse how the solutions without event detection change as the tolerance is sharpened and how coarse a tolerance we can use with the event detection codes.

Again, Figure 47 exposes the behaviour whereby the same code with the same problem at different tolerances give different results. We would expect the code at the sharper tolerances to give very similar curves but clearly, LSODA even at sharp tolerances is still resizing the step size. This confirms that even at high tolerances, the step-size when first encountering the discontinuity is still too big.

From Figures 48 and 47, the addition of the event detection lets us use a smaller tolerance. We also see that the code with event detection allows us to use a tolerance of 10^{-3} and still get the correct answer whereas without event detection, even a tolerance of 10^{-12} was not enough.

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

In this section, we use Runge-Kutta pairs of the same order; DOPRI5 in R aliased as 'ode45', DOPRI5 in Python aliased as 'RK45' and RKF45 in Scilab aliased as 'rkf'.

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 code as the tolerance is sharpened to discuss why the problem is not solvable simply by sharpening the tolerance. We also coarsen the tolerance for the code with event detection where that is possible to see how

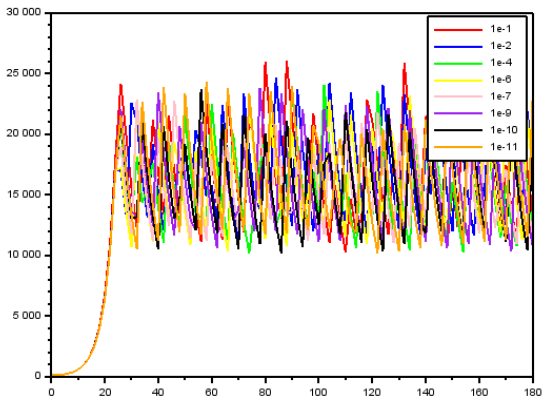

Figure 47: State discontinuity tolerance study on Scilab's LSODA without event detection

coarse the tolerance can be while still obtaining sufficient accuracy.

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

Form Figure 49, we see the behaviour in LSODA whereby the same solver on the same problem with different tolerances still give different solutions. This tells us that the step is still being resized even at sharp tolerances and thus the first step that first encounters a discontinuity is still too big. The solver cannot step over the discontinuity and will have to repeatedly retake the step. But the global variables will be switched from the first step that encountered the discontinuity and thus the problem cannot be solved. This strengthens our conclusion that problem with similar state discontinuities cannot be solved without event detection.

We then report on the efficiency data with only the code without event detection in Table 19.

tolerance study on state discontinuity using Python's version of dopri5 We perform the same experiment in Python. The absolute and relative tolerances is set to a particular value 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 and as Python's version of DOPRI5 has event detection, we will see how coarse the tolerance can be set while still giving us a correct solution.

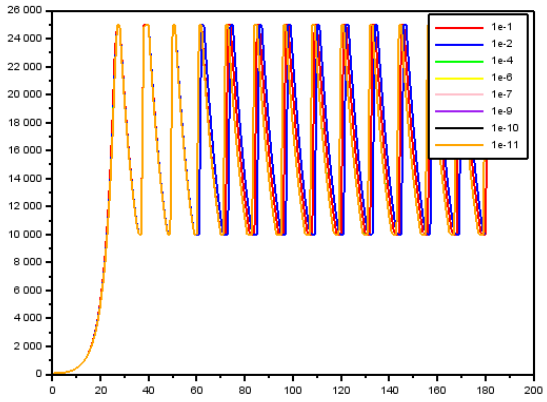

Figure 48: State discontinuity tolerance study on Scilab's LSODA with event detection

We must note that the solver crashes if we ask for a tolerance of 0.1 as it requires smaller steps in the case without event detection. The problem is too discontinuous and the solution is completely wrong as shown in Figure 52. The code stops at time 149 and cannot complete the whole time interval while getting a completely erroneous solution.

In Figure 50, we can see that even at sharp tolerances, the solver is still resizing the step-size. This indicates that even at sharp tolerances, the step along the continuous intervals are still bigger than at the discontinuity and the solver cannot step over it without resizing the step-size repeatedly. The global variables are switched at the first encounter with the discontinuity and thus the problem cannot be solved simply by increasing the tolerance.

In contrast, when using event detection, the code can use very coarse tolerances. We can see that 10^{-4} is sharp enough to solve the given problem, the blurring occurring due to the coarser tolerances. We then present the efficiency data in Table 20 to show how the code with event detection is also far more efficient.

We can see in Table 20 that across all the different tolerances except 0.1, the code with event detection require less function evaluations, around several thousands less for the sharper tolerances. We note that at a tolerance of 0.1, the code is using less function evaluations without event detection but at this tolerance, the code without event detection also gave a completely erroneous solution where it did not follow the oscillating between 10000 and 25000 pattern.

discontinuity RKF45 tolerance study in Scilab Scilab uses RKF45 which is a different Runge-Kutta pair as DOPRI5 but have the same order. It does not have event detection but we still perform the experiment on the code

Table 18: Scilab LSODA State Discontinuity tolerance study

tolerance	nfev	event nfev
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.

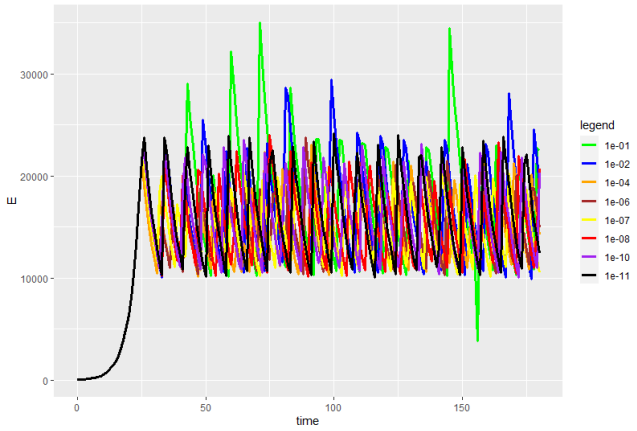

Figure 49: State discontinuity tolerance study on R's dopri5 without event detection

without event detection. We pick several values for the the absolute and relative tolerances and run the solvers. In so doing we see how the code performs as the tolerance is sharpened.

Scilab's '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 on the same line. Figure 53 shows that the step is still being resized and thus the problem cannot be solved by simply using higher tolerances. We can then conclude that event detection is required.

Table 19: R rk45 State Discontinuity tolerance study

tolerance	nfev	nsteps
1e-01	1082	180
1e-02	1142	189
1e-04	2014	323
1e-06	2027	328
1e-07	2193	355
1e-08	2919	471
1e-10	5194	855
1e-11	7690	1271

Table 20: Python rk45 State Discontinuity tolerance study

tolerance	nfev	event's nfev
0.1	536.0	664.0
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

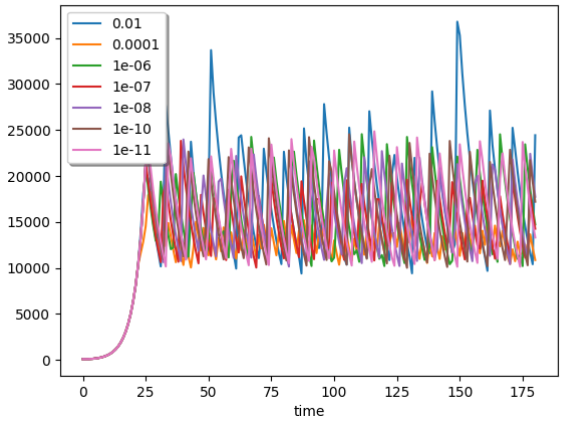

Figure 50: State discontinuity tolerance study on Python's DOPRI5 without event detection

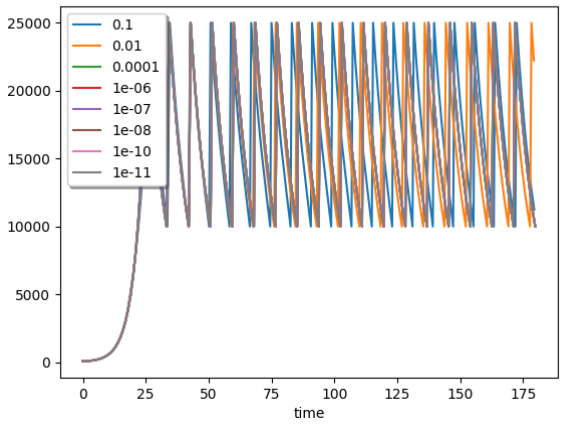

Figure 51: State discontinuity tolerance study on Python's DOPRI5 with event detection

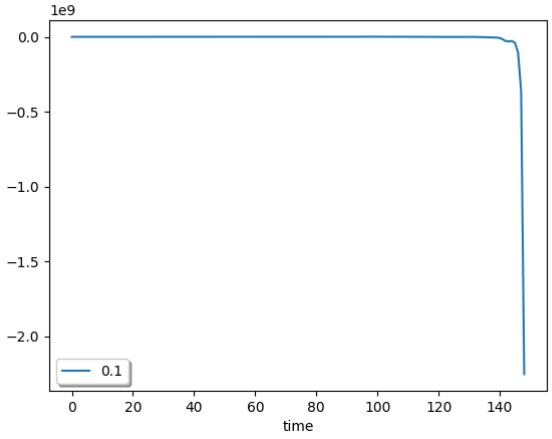

Figure 52: Python's DOPRI5 without event detection at 0.1 tolerances

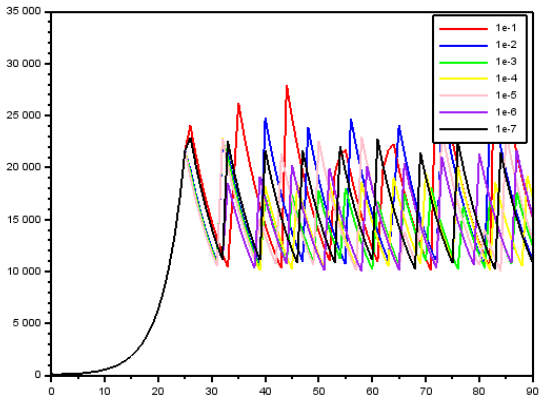

Figure 53: State discontinuity tolerance study on Scilab's RKF45 without event detection

Table 21: Scilab RKF45 State Discontinuity tolerance study

tolerance	nfev
0.1	547.
0.01	732.
0.001	1294.
1e-4	1956.
1e-5	2364.
1e-6	2662.
1e-7	2802.

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

4.2 may have a section on dopri5.f and rkf45.f

5 Summary, Conclusions, and Future Work

5.1 Summary

In this report, we analysed the difficulties in modelling discontinuous epidemiology models. We reported on the stability and discontinuity problems associated with these models. We showed how stability affects out 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 trivial 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.

In 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 though error-controlled can solve the problem, the use of discontinuity handling in the form of cold starts lead to more efficient solutions that allow us to use coarser tolerances.

In the state dependent discontinuity problem, we have shown that even error control solvers cannot step over multiple discontinuities. We have shown that if the discontinuity is state dependent, we cannot model it trivially using the model function $f(t, y)$ alone as we need a way for the solver to use a previous model function before the discontinuity and a new model function after it. 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.

5.2 Conclusion

We begin our conclusions by making recommendations to epidemiologists. We recommend that epidemiologists avoid using fixed step-size solvers. They must also avoid coding their own solvers, especially if they are not implementing a fixed step-size solver and prefer the high quality software packages available in their respective programming environments.

We recommend using a form of discontinuity detection whenever there is an if-statement in their right hand side function. The trivial experiment detailed in Section 1.6 should be a start.

When they have a problem which has a time dependent discontinuity and know when the discontinuity occur, they should use the form of discontinuity handling presented in this report. Using cold starts allow the researcher to integrate continuous subinterval of the problems in separate calls leading to efficient and accurate solutions.

When they have a problem which has a state dependent discontinuity, they should map out the thresholds at which these discontinuity 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.

5.3 Future Work

In Section 3.1, we see that Radau exhibits a strange behaviour when solving the naive state dependent problem. A further analysis needs to be done on the algorithm itself as two different implementation of the algorithm gave similarly bad solutions.

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

VI ========== We also propose a research on finding and categorising discontinuity detection algorithms and how they can be implemented in IVODE, BVODE and PDE software efficiently. ========== VI

VI ========== LOOK INTO DDE (delay differential equations) solvers to solve state dependent problems. ========== VI

6 Bibliography

7 Ebola Paper

In Chritian L. Althaus's paper, 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 needs 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 at the real-life recorded data points. We run different ODE models by changing the parameters until we minimise these errors. The model with that minimum error is the 'best fit' model and the parameters that were used by this model are our estimates. These estimates are then used to understand the spread of the virus.

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

The following is the pseudo-code for our attempt at replicating the experiment:

```
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 were reported in the paper is shown in Figure 54. Our figures are as shown in Figures 55, 56 and 57

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 54: Original figure in Ebola paper

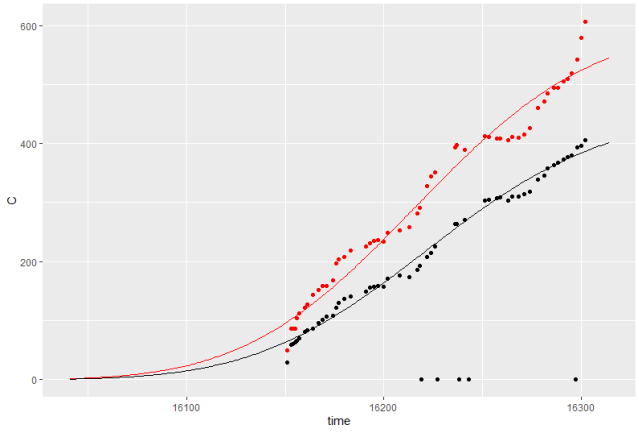

Figure 55: Our Guinea Figure

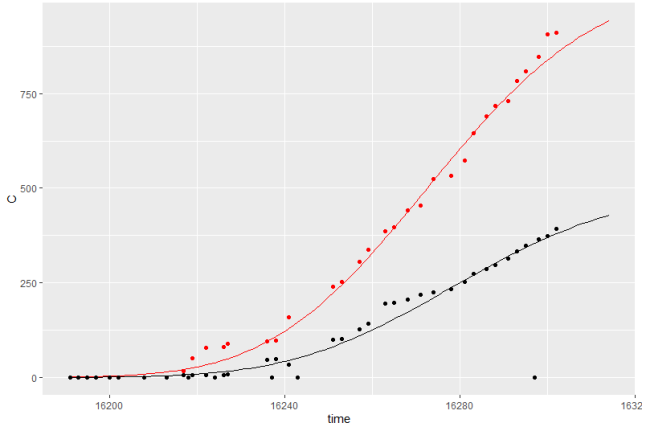

Figure 56: Our Sierra Leone Figure

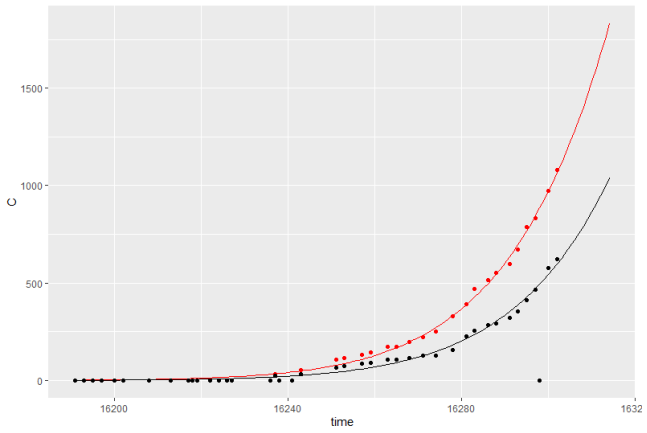

Figure 57: Our Liberia Figure