Performance Analysis of ODE Solvers on Covid-19 Models with Discontinuities

Humaid Agowun and Paul Muir

July 6, 2022

Abstract

In this report, we consider the numerical solution of two challenging Covid-19 ordinary differential equation (ODE) models that have discontinuities. The discontinuities are associated with modeling the introduction of measures to slow the spread of the virus. One of the models has a time-dependent discontinuity; this means that at a given point in time, a discontinuity is introduced into the model. The second type of discontinuity is a state-dependent discontinuity; in this case, the time at which the discontinuity arises depends on the value of one or more of the solution components, and thus it is not known apriori. These discontinuities make the models quite challenging for standard ODE solvers to solve. Furthermore, the presence of exponentially growing solution components adds to the difficulties faced by standard ODE solvers when they are used to try to solve these types of problems.

In this report, we present an investigation of performance of a collection of ODE solvers (we consider 21 solvers) available in four popular software environments: R, Python, Scilab, and Matlab, when applied to solve these Covid-19 models.

We first focus on straightforward implementations of the models where the user employs the solver to attempt to solve the problems using default settings, e.g., default tolerances, and simple implementations for the discontinuities, i.e., the introduction of 'if' statements into the functions that define the right-hand sides of the ODE systems. *Such implementations of the models and usage of the solvers are typical of what Covid-19 researchers might employ in attempting to solve their models.*

We follow with an investigation of an approach for the solution of the models that uses slightly more advanced treatments of the models; the approach involves making better use of the capabilities of the solvers and better implementations of the models themselves.

We also highlight a number of issues with the way that some of the solvers are implemented in some of the software environments. For example, the treatment of output points, i.e., the points in the domain where solution values should be provided, is an issue for some of the solvers in some of the software environments.

We show that the standard use of ODE solvers available within widely used software environments, applied to simple implementations of these Covid-19 models, will frequently deliver numerical solutions that have no significant digits of accuracy. Furthermore, the solvers give no indication that the returned solutions may be inaccurate. We also show that these straightforward treatments of the models are always less efficient than the slightly more advanced treatments. We show that the slightly more advanced treatments of the models can result in more efficient computations while at the same time providing more accurate approximate solutions.

1 Introduction

In this report, we describe a detailed investigation of the performance of a variety of software packages applied to initial value ordinary differential equation (IVODEs) encountered in Covid-19 models (see, e.g., [1]). Our study considers Covid-19 models with discontinuities associated with the introduction of measures to slow the spread of the virus.

For any mathematical model, the accuracy requirements requested for the numerical solution of the model 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 approximate solution must always be negligible compared to the accuracy of the model itself. Most IVODE solvers allow the user to specify a parameter called a tolerance. The solvers use adaptive algorithms to attempt to compute an approximate solution with a corresponding error estimate that is approximately equal to the tolerance. *Researchers deserve to obtain accurate solutions to the models that they are studying.*

In this report, we will show that the straightforward use of standard IVODE solvers on Covid-19 models with discontinuities can lead to numerical solutions that have large errors, sometimes of the same order of magnitude as the solution itself. We also show that these computations can be substantially less efficient than necessary.

In Section 1.1, we define the SEIR Covid-19 models which we will consider throughout this report. In Section 1.2, we discuss numerical stability issues that arise in problems (such as Covid-19 models) with exponentially growing solutions. In Section 1.3, we provide an overview of the software that we will consider in this report. We also explain the difference between fixed step-size and error-control IVODE solvers. In Section 1.4, we discuss issues with the evaluation of approximate solutions at output points that lead to inefficiencies for some of these solvers. In Section 1.5, we consider the effects of discontinuities on the performance of these solvers.

In Section 2.1, we apply the solvers in a straightforward fashion to a Covid-19 problem with a time-dependent discontinuity and show how, in some cases, this results in numerical solutions that are computed inefficiently and with errors of the same magnitude as the solutions being computed. In Section 2.2, we will use a discontinuity handling approach to accurately solve the time-dependent discontinuity problem. In Section 2.4, we will apply some of the solvers to this model with a range of tolerances to investigate the effects of tolerance on the accuracy and efficiency of the solvers.

In Section 3.1, we apply the solvers to a Covid-19 problem with a state-dependent discontinuity and show how, when using a straightforward implementation of the problem, none of the solvers are able to obtain reasonably accurate solutions. In Section 3.2, we will explain how even the use of very sharp tolerances is not sufficient to improve the accuracy of the computed solutions and show that a more effective way to solve this problem is through the use of a capability provided in some solvers known as event detection, which we

will describe in Section 3.3. We then show the results of using this approach to obtain accurate solutions to the state-dependent discontinuity problem in Section 3.4 and perform a tolerance study on this problem in Section 3.5.

In Section 5, we employ the original Fortran implementation of the Radau solver provided in several of the software environments, in order to further investigate the poor performance of this solver on the state-dependent discontinuity model. We conclude in Section 6 with a summary and a discussion of potential future work.

1.1 Two Covid-19 models with discontinuities.

In this subsection, we describe the models that we are going to consider in this report. They involve a typical SEIR model to which we add discontinuities.

An IVODE problem is defined by the ODEs and initial conditions:

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

Given $f(t, y(t))$ and $y(t_0)$, the goal is to find an approximation to $y(t)$ over a domain $[t_0, t_f]$.

In this report, we consider the Covid-19 model [2]:

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

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

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

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

In this SEIR model, S is the number of susceptible individuals, E is the number of exposed individuals, I is the number of infected individuals and R is the number of recovered individuals. N is the population size. The other parameters in this model are as follows: α^{-1} is the average incubation period, β is the transmission rate, γ is the recovery rate and μ is the birth/death rate. In this report, we assume that all these parameters are known. Our goal is to investigate the performance of IVODE solvers on forms of this problem that have discontinuities. *We will see that, depending on how we solve the ODEs, we can get approximate solutions that are not efficiently computed and that may have significant errors.* This latter issue can have serious consequences as the computed solution will fail to show the actual impact of the virus corresponding to the epidemiology theories behind the mathematical models. These incorrect numerical solutions may lead epidemiologists into reaching incorrect conclusions and thus lead them into questioning the mathematical models themselves when, in fact, it is the solvers that are at fault.

The discontinuities we are going to consider involve the virus transmission parameter, β. Before measures such as social distancing, masking, etc., are implemented, β has a much higher value than after the measures are introduced. For the purpose of this study, we will use a large β value - equal to 0.9 - before the measures are introduced, and a small β value - equal to 0.005 - after they are implemented, corresponding to a highly contagious Covid-19 variant and extreme shut down measures, respectively. These abrupt changes in the parameter β introduce discontinuities into the model, as we will show in Section 1.5. We will consider two types of discontinuities. One depends only on t; the other depends on the value of one of the solution components. We will refer to the former as a time-dependent discontinuity and the latter as a state-dependent discontinuity.

For the time-dependent discontinuity, we will assume that at some point in time, measures are implemented that will lead to a reduction in the parameter β. We would like to solve the problem through this discontinuity, but as we will show, the discontinuity presents a significant issue for the ODE solvers.

For the state-dependent discontinuity, we consider the following situation. If the population of exposed people reaches a certain maximum threshold, measures are introduced, which corresponds to decreasing the value of β, which in turn leads to a decrease in the number of exposed and infected cases. This abrupt decrease in the β value introduces a discontinuity into the model. When the population of exposed people later drops below a certain minimum threshold, the measures are relaxed, which corrresponds to increasing β back to its original value, which introduces another discontinuity. This increase in the β value in turn leads to a exponential growth in the number of exposed and infected cases. We will try to model this problem through multiple instances of shut-downs followed by periods where measures are relaxed. We consider a case where vaccines are not being used. This corresponds to setting β back to its original value when the measures are removed. 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. For this problem, we show that all the solvers, when used in a straigthforward manner, will fail.

The other parameters are assumed to be constant with N = 37,741,000 (the approximate Canadian population size), $\alpha = 1/8$, $\gamma = 0.06$, and $\mu = 0.01/365$. 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 IVODEs that is in a form that can be treated by typical ODE software packages.

1.2 Exponential growth and instability for ODEs

Some of the solution components of the SEIR model exhibit exponential growth over certain time periods. In this section, we discuss exponentially growing solutions and their impact on the accurate computation of a numerical solution. Firstly, we give a quick overview of stability for ODEs. Then we will show that the SEIR model is unstable over certain time intervals and how changing

the model in a way that corresponds to introducing measures such as social distancing can improve the stability of the model because, once this is done, none of the solution components are exponentially increasing. This is important as this means that before measures are implemented, accurate solutions are difficult to obtain but the introduction of the measures results in changes to the model so that the solution components become exponentially decreasing instead of exponentially increasing. This corresponds to an improvement in the stability of the model that can allow the solvers to compute more accurate solutions.

The stability of an ODE is often defined in terms of the impact of small changes to the initial values on the solution to the problem. An ODE is unstable if a small change in the initial values results in a large change in the solution; otherwise, the ODE is said to be stable.

It is straightforward to see that problems with a solution component that exhibits exponential growth are unstable. As mentioned above, this is the case with some of the solution components of a Covid-19 model. The population of infected people, I, grows exponentially as long as no measures are introduced to reduce the spread of the virus.

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

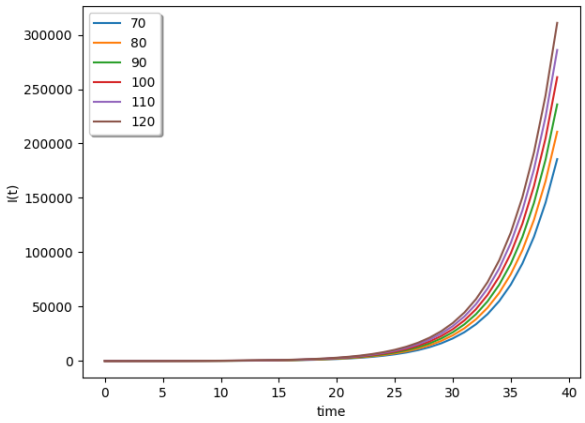

Figure 1: When a solution exhibits exponential growth, relatively small changes in the initial value can eventually lead to much different solution values. Here we consider initial values of $I(t)$ equal to 70, 80, .., 120.

However, when we introduce measures which corresponds to employing a smaller β value, the solution components that were growing exponentially will exhibit slower exponential growth or can even show exponential decay. Slower exponential growth or exponential decay means that the solution will not be as sensitive to small errors in the computation.

Epidemic modeling problems exhibit solutions with this type of behavior. At first, the problem is unstable but as measures are implemented, which lead to exponential decay rather than growth for some of the solution components, the problem becomes stable. We show this in Figure 2 for the model with the time-dependent discontinuity. At first, the $I(t)$ solution components diverge when there is exponential growth, but the introduction of measures such as social distancing leads to exponential decay which makes them converge. Thus the measures not only save lives but also improve the capability of solvers to compute accurate solutions.

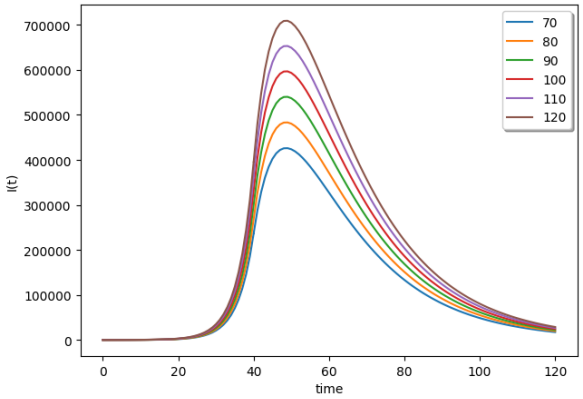

Figure 2: Unstable solutions in the region [0, 40] becomes stable in the region [40, 90] as measures are implemented. Here we consider initial values of $I(t)$ equal to 70, 80, .., 120.

1.3 Brief overview of numerical ODE solvers

We start by explaining how typical solvers attempt to solve an IVODE. Given initial values (at the initial time, t_0), the solver will use an initial step size, h, to compute a solution at time, $t_1 (= t_0 + h)$. The solver will attempt to take a sequence of steps until it reaches the end time. High-quality solvers will also employ an interpolation algorithm, usually locally within each step, to get a continuous numerical solution. We note that a solver is said to have order p if the difference between the true solution and the computed solution is $O(h^p)$.

In this section, we describe adaptive step-size error control for the numerical solution of an IVODE. We then discuss the numerical solvers we are going to use throughout our investigation. We will then provide an additional discussion on the implementation of interpolation to get a continuous numerical solution and how some programming environments have not set up their ODE solvers to use interpolation in an optimal fashion.

1.3.1 Fixed Step Size and Error Control Solvers

In this subsection, we explain the role of the tolerance and the difference between fixed step size and adaptive step-size error control solvers.

The tolerance is a measure of how accurate we want the solution computed by the solvers to be. A key point here is that solvers that can take a tolerance as input must have some way of computing an estimate of the error of the solution that they compute. Then that error estimate can be compared with the user-provided tolerance. Generally, an absolute tolerance means that we want the error estimate to be approximately equal to the tolerance, whereas a relative tolerance means that we want the ratio of the error estimate and the computed solution to be approximately equal to the tolerance. Some solvers will use a blended combination of the user-provided absolute and relative tolerances.

A solver is said to have a fixed step size if the solver begins with an initial step-size and this step-size is used throughout the whole integration. In this case, the solver will step from one point to the next and will not check if the numerical solution it obtains at the end of each step is sufficiently accurate. This fixed step-size is a determing factor in the accuracy of the solution but fixed step-size solvers do not have a way of assessing the accuracy of the solutions that they compute.

An error-controlled solver starts with an initial step size but as it takes a step, it will also compute an error estimate and will repeat the computation with a smaller step-size if the error estimate is larger than the tolerance. It will repeat this process until the error estimate satisfies the given tolerance. Only then will it move to the next step. Thus it reduces the step-size as needed throughout the computation. We note that the error depends on the step-size and that a smaller step-size generally leads to a smaller error. However, a small step-size means that the computation is slower because more steps will be needed. If the error estimate is much smaller than the tolerance on an accepted step, the solver will increase the step-size for the next step. This allows it to make sure that the given tolerance is satisfied over the whole problem interval with as large a step as possible being taken to optimize the efficiency of the computation.

Some researchers may be tempted to write their own solvers, based on a non-error control method like a simple fixed step-size Euler or Runge-Kutta method [3]. We will show, using some fixed step-size solvers, how these solvers simply cannot solve a Covid-19 model with reasonable accuracy. Without error control, these solvers cannot handle the discontinuity and stability issues that are present in these models and they will give erroneous solutions, often without even a warning that the computed solutions should not be trusted.

In this report, we will be referring to numerical solutions that have "reasonable accuracy". By this we mean that when these solutions are plotted, they are visually indistinguishable from a high accuracy solution. This means that the numerical solutions agree with a high accuracy solution to at least two decimal places, which is quite a modest accuracy requirement. We will see however, that for the models we consider in this report, even this modest accuracy requirement cannot be met by many of the solvers when straightforward implementations

are employed.

1.3.2 The ODE Solvers

The ODE solvers are grouped into the following classes: fixed-step Runge-Kutta methods, Runge-Kutta pairs [3], and multi-step methods [3].

A Runge-Kutta method is a one-step method that uses function evaluations, i.e, evaluations of $f(t, y(t))$, within the step. An example is the classical four-stage, fourth-order Runge-Kutta method [3]. Another example is the well-known forward Euler method. A simple solver based on this type of method steps across the time domain with a fixed step-size and has no error control.

A Runge-Kutta pair [3] uses two Runge-Kutta methods of order p and $p+1$ for some integer, p. One of the methods is used to compute a solution and the other method is used to compute an error estimate. A solver that is based on a Runge-Kutta pair resizes the step based on the error estimate, as discussed previously. An example of such a solver is the DOPRI5 solver [3] that uses a fifth-order method for the solution and a fourth-order method for the error estimate.

A multi-step method is a solver that will use a linear combination of solution and function values from the current and previous steps in order to obtain a solution approximation at the end of the current step. An example of such a solver is LSODA [3]. Such solvers compute an error estimate for the numerical solution that they return and use the error-estimate to control the step-size as discussed above. Such solvers typically implement a family of multi-step methods and thus also have the capability to adapt the order of the method they used based on the error estimate.

R packages Scientists who solve ODE models in R commonly use the deSolve package [4], and the *ode*() function within it. This function provides many numerical ODE solvers but we have focused our investigation only on the following popular choices: 'lsoda', 'daspk', 'euler', 'rk4', 'ode45', 'Radau', 'bdf' and 'adams'. The default method is 'lsoda' and the default tolerances are 10^{-6} for both the absolute and relative tolerances. We also note that we did not consider the other integrators in the deSolve package like $rkMethod()$, which provides other Runge-Kutta methods, and the other methods which are available through the *ode*() function itself.

The error control solvers are:

- 'lsoda' which calls the Fortran LSODA routine from ODEPACK [5]. It can automatically detect stiffness and choose between a stiff Backward Differentiation Formula (BDF) [3] and a non-stiff Adams solver [3].
- 'daspk' which calls the Fortran DAE solver of the same name [6].
- 'ode45' which calls an implementation of the Dormand-Prince (4)5 (DOPRI5) Runge-Kutta pair [3], written in C.

- 'Radau' which calls the Fortran solver RADAU5 [7] which implements a Runge-Kutta method of 5^{th} order known as the RADAU IIA method.
- 'bdf' which calls the stiff solver inside the Fortran LSODA package which is based on a family of BDF methods.
- 'adams' which calls the non-stiff solver inside the Fortran LSODA package which is based on a family of Adams methods.

The fixed step-size solvers are:

- 'euler' which calls a simple solver based on the classical Euler method and is implemented in C.
- 'rk4' which calls a simple solver that uses the classical Runge-Kutta method of order 4 and is implemented in C.

We will use these latter two methods to demonstrate what happens when non-error-controlled solvers are applied to the Covid-19 models.

We next consider the R interface for handling output. The *ode*() function is given an array of output points. However, in default mode, there is an issue with the way in which the output points are treated. The array of output points affects the step sequence and efficiency of the solver in a manner which we describe in Section 1.4.

Python packages In Python, researchers can use the scipy.integrate package [8], and will normally use the *solve_ivp*() function due to its newer interface. It lets the user apply the following methods: 'RK23', 'RK45', 'DOP853', 'Radau', 'BDF' and 'LSODA'. The default solver in *solve_ivp*() is 'RK45' and the default tolerance is 10^{-3} for the relative tolerance and 10^{-6} for the absolute tolerance. All of these solvers employ some form of error control. The solvers are:

- 'RK23' which uses an explicit Runge-Kutta pair of order 3(2), the Bogacki-Shampine pair of formulas [9], and is implemented in Python.
- 'RK45' which uses the DOPRI5 pair of formulas mentioned earlier, and is implemented in Python.
- 'DOP853' which uses an explicit Runge-Kutta triple of order 8(5, 3) [10], and is implemented in Python.
- 'Radau' which uses the implicit Radau IIA method of order 5. It is a Python implementation of the RADAU5 Fortran solver.
- 'BDF' which uses BDF methods with the order varying automatically from 1 to 5, and is implemented in Python.
- 'LSODA' which calls the Fortran LSODA routine from ODEPACK.

We note that all solvers in *solve_ivp*() have error control and that only 'LSODA' uses the Fortran package itself; the others are Python implementations.

We next discuss Python's *solve_ivp*() interface. Given only the initial time and the final time, a solver from this method will adaptively step across the domain, returning the output at the end of each successful step. Alternatively, a solver can take a *t_eval* array of specified output points. The solver is allowed to take as big a step as needed and required solution approximations, as specified by *t_eval*, are obtained using interpolation. Thus it does not suffer from the inefficiencies described in Section 1.4. The interface also has a *dense_output* flag. This returns an interpolant for the solution over the entire time range.

Scilab packages In Scilab, researchers solve differential equations using a method from the *ode*() function [11]; the following methods are available: 'lsoda', 'adams', 'stiff', 'rk', 'rkf'. The default integrator is 'lsoda'. Default values for the tolerances are 10^{-5} for the relative tolerance and 10^{-7} for the absolute tolerance for all solvers except 'rkf' for which the relative tolerance is 10^{-3} and the absolute tolerance is 10^{-4}. All of these solvers are error control solvers. The solvers are as follows:

- 'lsoda' which calls the Fortran LSODA routine from ODEPACK.
- 'stiff' which calls the stiff solver inside the Fortran LSODA package which is based on a family of BDF methods.
- 'adams' which calls the non-stiff solver inside the Fortran LSODA package which is based on a family of Adams methods.
- 'rk' which is based on an adaptive Runge-Kutta method of order 4. It uses Richardson extrapolation [12] for the error estimation. This method calls the Fortran program 'rkqc.f' [13].
- 'rkf' which calls the Fortran program written by Shampine and Watts that is based on Fehlberg's Runge-Kutta pair of order 4 and 5 (RKF45) pair [14]. The Fortran program is called 'rkf45.f' [13].

The *ode*() function in Scilab takes as input a vector of output points and the computation uses interpolation or stops the integration at the output points, as described in Section 1.4, based on the method used. For example, Scilab's 'rkf' is an interface to an old software package, 'rkf45.f' which does not have interpolation capabilities and thus in order to obtain solution approximations at the output points, the solver must step to each output point.

Matlab packages In Matlab, researchers can solve differential equations with the *ode suite* [15] of functions. We will consider two of these functions: *ode*45() and *ode*15*s*(). Default values for the tolerances are 10^{-3} for the relative tolerance and 10^{-6} for the absolute tolerance. The solvers are:

- *ode*45() which calls a Matlab implementation of DOPRI5.
- *ode*15*s*() which implements a variable-step, variable-order (VSVO) solver based on the numerical differentiation formulas (NDFs) [15] of orders 1 to 5. Optionally, it can use BDF methods but the authors indicate that these are usually less efficient.

Functions in the *ode suite* take an array of output points as input but the solvers use adaptive step-size control and interpolation to obtain solution approximations at the output points. With such an interface, the solvers do not suffer from the issues discussed in Section 1.4.

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

In R, Python, and Scilab, the 'lsoda' method is a wrapper around the Fortran LSODA code from ODEPACK.

The R 'bdf' method is equivalent to the Scilab 'stiff' method in that they both use the LSODA code from ODEPACK; however, the Python 'BDF' method is a different implementation in Python itself.

The R 'adams' method and the Scilab 'adams' method are the same since they both use the LSODA code from ODEPACK.

The R and Python Runge Kutta 5(4) pairs are both implementations of DOPRI5 but they have different source code as the version in Python is implemented in Python while the R version is implemented in C. The *ode*45() function in Matlab is a Matlab implementation of DOPRI5. The Scilab 'rkf' method does not use the same pair; it uses the Shampine and Watts implementation of the Fehlberg's Runge-Kutta pair, not the Dormand-Prince pair.

The Scilab 'rk' method, which is of order 4, and the R 'rk4' method are not the same solvers. The Scilab 'rk' method is adaptive (error-controlled with Richardson extrapolation for the error estimate) whereas the R 'rk4' method is a fixed step-size implementation of the classical 4-stage, 4^{th} order Runge-Kutta method.

The R and Python 'Radau' methods have different source code as Python implements a Python version of RADAU5 while R calls the Fortran version of RADAU5 through a C interface.

1.4 Observations on obtaining solution approximations at output points

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

As mentioned earlier, using a default initial step-size, a solver will take a trial step. This computation gives both a solution approximation at the end of the step and a corresponding error estimate. The solver will then accept or reject the step based on whether the error estimate satisfies the tolerance and will, respectively, adjust the step-size to take the next step or retake the current step with a smaller step-size. This process is repeated until the solver reaches the end of the interval. However, often the users of an ODE solver will require output at specific points and these points may be internal to the steps. The current state-of-the-art approach to get solution approximations at these output points is to construct a high accuracy interpolant on each step and to return the value of the interpolant at the required point. Ideally the interpolant is of order p if the numerical ODE solution is of order p. This way the accuracy of the solution approximation at a point that is interior to a step should be comparable to the accuracy of the solution approximation at the end of the step. However some solvers use a lower order interpolant in order to reduce the computational cost.

Note that the standard ODE solvers only control the error at the end of the step. That is, an error estimate is generated for the solution approximation at the end of the step and the step is accepted if this error estimate satisfies the tolerance. It is hoped that the solution approximations obtained through the use of the interpolant will be of comparable accuracy to the solution approximation at the end of the step. It is typically the case that no error control is actually applied to the continuous solution approximation.

In R and Scilab, the above approach for handling output points is not used in all the solvers. Instead, some solvers in R and Scilab use the output points to dictate the step-size. An issue arises when many output points appear between the steps that would normally be taken by the solver. These solvers will use the difference between the current output point and the next output point to determine the step-size. We note that some R solvers, such as the 'ode45' method, do have interpolants but that their default implementation still treats the output points in a way that can negatively affect the efficiency of the solver.

In such approaches, the output points will limit the step-size that can be taken and will lead to additional function evaluations being performed because the solver needs to compute a solution approximation using the numerical method at each output point. This will lead to a considerable drop in efficiency as we will show later in this report; see for example Tables 9 and 10. These tables show that a problem that can be solved with 150 function evaluations will be solved with 500 function evaluations when there are many output points.

This method of handling output points in which the solver steps to each output point and uses the numerical method itself to compute a solution approximation also means that the accuracy of the solution depends on the space between the output points. Thus, we get the unusual behavior that the accuracy is increased by putting the output points closer together and the accuracy is decreased by putting them further apart. We will point out these inconsistencies as they become relevant later in this report. We also note that spacing the points closer together is not a good way to control the accuracy as it is impossible to know beforehand how close the points should be in order to obtain

a desired accuracy.

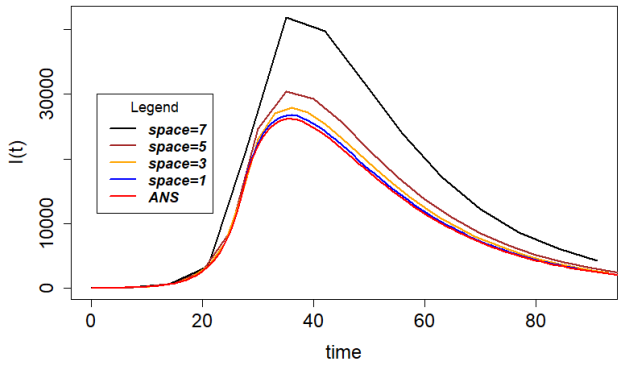

Figure 3: The result of using the R 'ode45' method to solve the same problem with a very coarse tolerance but with different spaces between the output points. Here the spacing between the points is 1, 3, 5 and 7. The corresponding approximate solutions are plotted alongside a highly accurate solution (ANS).

Figure 3 shows an experiment where we solve the time-dependent discontinuity Covid-19 problem using the R 'ode45' method, which is an implementation of DOPRI5 which has error control and an interpolation capability but allows the output points to affect the integration. We set both the absolute and relative tolerance to 0.1 and thus expect low accuracy but very good efficiency. However, the space between the output points becomes the limiting factor for the step-size. When there are many output points, the computed solution has more accuracy than is requested and is computed in a very inefficient manner considering the required tolerance. We record the number of function evaluations in Table 1 and it can be seen that the solver is using many more function evaluations than are needed to satisfy such a coarse tolerance. In Table 1, 'spacing' refers to the distance between the output points and 'nfev' is the number of function evaluations. A spacing of 1 means that the set of output points is $[1, 2, ..., 95]$. A spacing of 3 means that only every third point from the above list of output points is defined, and so on.

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

It is important that users employ the interpolation option for an ODE solver

Table 1: R DOPRI5 output point spacing experiment number of function evaluations.

spacing	nfev
1	572
3	188
5	116
7	80

whenever such an option is readily available so that the solvers can run as efficiently as possible. We also reiterate that the interpolant should have an interpolation error that is at least of order p if the ODE solver gives a solution with an error that is of order p so that the interpolation error is not larger than the error of the numerical solution.

1.5 Discontinuities and their effects on solvers

The main purpose of this report is to discuss how to solve Covid-19 models with discontinuities and how these discontinuities affect the process of computing an accurate numerical solution to the model. In this section, we will show what happens when a solver encounters a discontinuity and how this discontinuity leads to inaccurate solutions.

We first note that one of the key assumptions made in the derivation of the numerical methods upon which ODE solvers are based is that the function $f(t, y(t))$ and a sufficient number of its higher derivatives are continuous. If the right-hand side function is discontinuous, this can have a major (negative) impact on the performance and accuracy of the solver.

We will see that discontinuities will have huge impacts on the accuracy and efficiency of the solvers, and that some solvers, even with error control, will require an extremely sharp tolerance in order to step over a discontinuity in a way that allows them to obtain a reasonably accurate solution approximation. We will also show that fixed-step solvers simply cannot solve these problems accurately.

It is important to note that the step taken by a solver that first meets a discontinuity will almost always fail. This is because in order for the solver to step over a discontinuity, the step size needs to be much smaller than the one that is typically being used before the discontinuity is encountered. The solver will thus have to retake the step with a smaller step size and as long as the error estimate associated with the numerical solution computed on the step is not small enough, it will need to continue reducing the step-size. This leads to a large number of function evaluations near the discontinuity.

In Figures 4 and 5, we run 'LSODA' and 'DOP853' from Python on the time-dependent discontinuity problem where a discontinuity is introduced at t=27 and plot the time at which each function evaluation occurs. We see a

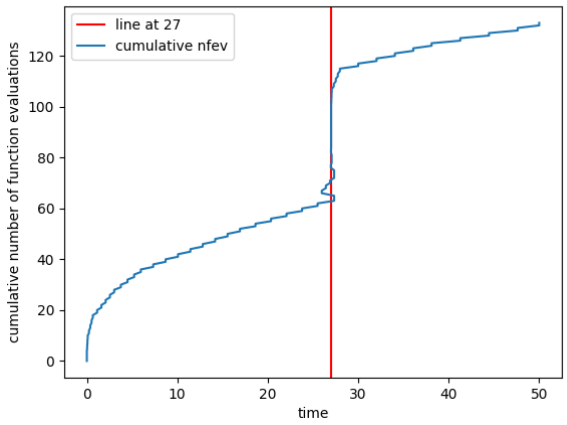

Figure 4: Function evaluations for the Python 'LSODA' method for the time-dependent discontinuity problem with a discontinuity at t=27.

spike in the number of function evaluations at the discontinuity as the solvers repeatedly retake the step with smaller and smaller step-sizes.

Following from the above discussion, we can suggest that, for the case where the location of the time discontinuity is unknown, researchers could carry out a manual discontinuity detection experiment to see if their model has a discontinuity and if so, where it is located. A trivial experiment can be done by collecting data that shows the time at which the solver makes each call to the function that evaluates the right hand side of the ODE. When a plot of the time against the cumulative count of the function calls gives an almost vertical line, this typically indicates that the function was called repeatedly at a specific time and thus that the solver repeatedly changed the step-size in this region in order to attempt to step over a discontinuity. In the remainder of this report, we will outline ways to accurately and efficiently solve problems with such discontinuities.

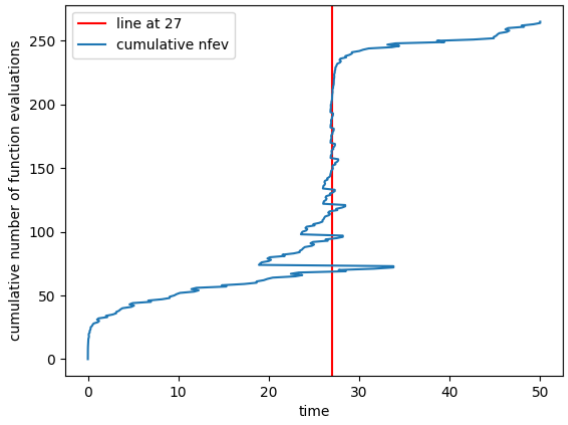

Figure 5: Function evaluations for the Python 'DOP853' method for the time-dependent discontinuity problem with a discontinuity at t=27.

2 Time-dependent discontinuity model

For the time-dependent discontinuity problem, we change the value of the parameter β from 0.9 to 0.005 at t=27. This introduces a discontinuity into the problem. We will show that this discontinuity leads to inaccuracies in the solutions computed by some of the solvers, particularly the fixed-step solvers. We then introduce a form of discontinuity handling, using what are known as cold starts, to show how to obtain an efficient and accurate approach for solving time-dependent discontinuity problems.

2.1 Naive solution of the time-dependent discontinuity model

A naive implementation of the model involves using an 'if' statement inside the right-hand side function, $f(t, y)$, to implement the change in β as measures are implemented.

In pseudo code, this looks like:

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

Also, to stay true to a naive treatment, we will use the default tolerances in this section. Discrepancies across the programming environments that are due to tolerance issues are investigated in Section 2.4. We also note than for the fixed step-size methods in the R environment, the step-size is 1 as the solvers will default to the distance between two consecutive output points, and we choose as our standard output point sequence $t = 1, 2, \ldots, 95$.

2.1.1 Naive solution to the time-dependent discontinuity model in R

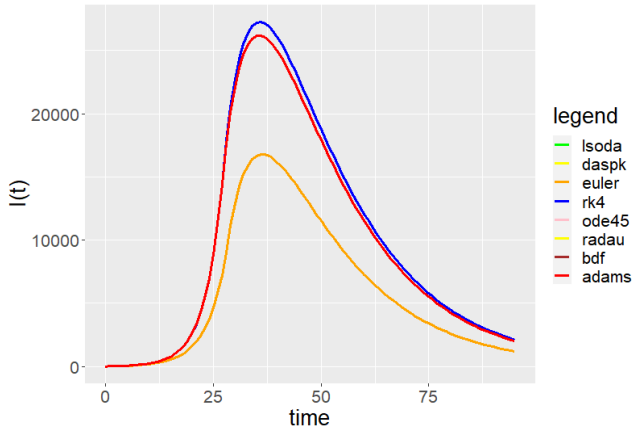

Figure 6: Solutions to the time-dependent discontinuity model using solvers from R.

From Figure 6, we can see that all the methods except 'euler' and 'rk4' compute solutions that agree to "eyeball" accuracy, which typically means that they agree to about two significant digits. The 'rk4' method gives a solution that is somewhat close to the solutions obtained by the other solvers but the solution computed by the 'euler' method is noticibly inaccurate. We note that all the other methods have error control while the 'rk4' and 'euler' methods are fixed step-size solvers.

We also note that the 'rk4' method does better than the 'euler' method for this specific problem as it has a higher order. But, since 'rk4' is using a fixed step-size with no error control, its performance is still better than expected. We show that this is entirely because of the issue associated with how output points are handled, as discussed in Section 1.4. If we use an output point sequence with a larger spacing between the output points, the 'rk4' methods gives results that are of similar accuracy to the results yielded by the 'euler' method. Figure 7 shows an experiment with 'rk4' used with different spacings between output points plotted together with an accurate solution (in red). We can see that as we increase the output point spacing, the solver does not give accurate results. Analyzing the source for 'rk4' and 'euler' shows that these methods select the step size based on the requested output points. Spacing out the output points affects the step-size which affects the accuracy of the fixed step-size solvers.

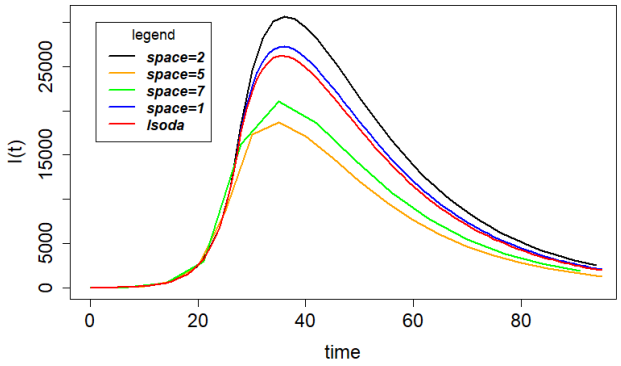

Figure 7: Solutions computed by 'rk4' in R with output point spacings compared with an accurate solution computed by LSODA.

If a user wants to use 'rk4' or 'euler', to get an accurate solution, the user would have to choose a small step-size. However, the user cannot know beforehand how small a step-size is small enough to deliver a desired accuracy. Furthermore, there is the issue that a sufficiently small step-size can vary from one part of the domain to another as the problem difficulty changes. A fixed step-size solver will have to choose the smallest step required anywhere in the domain and this can lead to substantial inefficiency. A better approach is to not use fixed step-size solvers. Reliable methods with error control should be preferred since these solvers can adaptively choose a stepsize sequence that will deliver the desired accuracy.

2.1.2 Naive solution to the time-dependent discontinuity model in Python

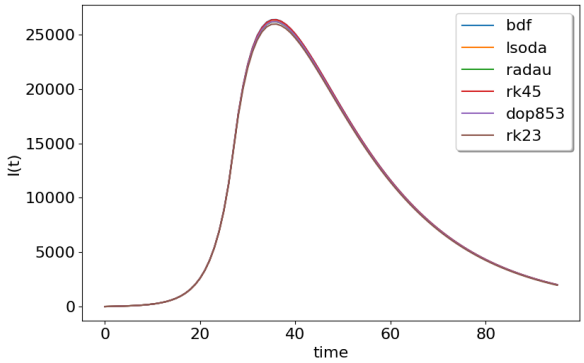

Figure 8: Solutions to the time-dependent discontinuity model using solvers from Python.

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

2.1.3 Naive solution to the time-dependent discontinuity model in Scilab

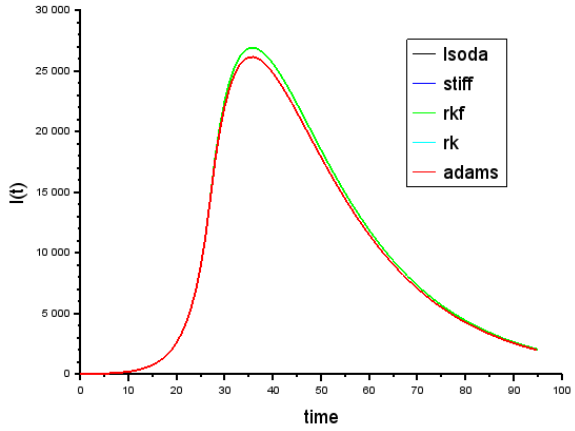

Figure 9: Solutions to the time-dependent discontinuity model using solvers from Scilab.

From Figure 9, we can see that in Scilab, all the methods give similar solutions except for 'rkf'. This is interesting as we know that 'rkf' uses error control. This is explained by noting that 'rkf' uses coarser default absolute and relative tolerances. We will show, through a tolerance analysis in Section 2.4, that with a sharp enough tolerance, 'rkf' also provides a reasonably accurate solution.

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

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

2.1.4 Naive solution to the time-dependent discontinuity model in Matlab

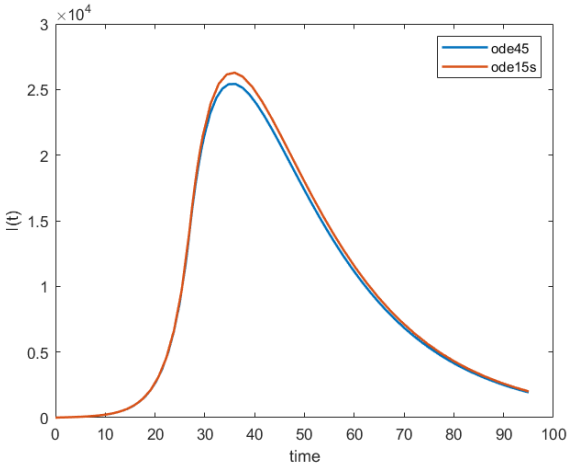

Figure 10: Solutions to the time-dependent discontinuity model using solvers from Matlab.

Figure 10 shows that Matlab's *ode*45 and *ode*15*s* are not in complete agreement. This is unexpected because both are error controlled. We note that the behaviour of *ode*45 is similar to what we have seen for 'rkf' in Scilab but the methods are based on different algorithms. In Matlab, both *ode*45 and *ode*15*s* have the same default tolerances so we can rule out that a tolerance difference is the reason for this behavior. We will see, in Section 2.4, that *ode*45 can give a similar result to *ode*15*s* when the tolerance is sharp enough. From this we can suggest that the issue may be associated with differences in the way that the two solvers apply the absolute and relative tolerances.

2.1.5 Summary of naive approach of solving time-dependent discontinuity problems

Generally, the time-dependent discontinuity problem can be solved accurately by solvers that employ error control with a sufficiently sharp tolerance. However as we will see in the next section, the computations are quite inefficient. (See Section 1.5 for an explanation of why this inefficiency arises.)

2.2 An improved approach for the solution of the time-dependent discontinuity model

A better way to solve the time-dependent discontinuity problem is to make use of cold starts. This means that we integrate up to the time at which the

discontinuity arises and then after the discontinuity we continue the integration with a *separate* call to the solver. Restarting a solver with a cold start at the time of the discontinuity improves the accuracy as we will see in this and the next section. It also improves the efficiency as fewer function calls are required since we do not have the spike in function calls due to the repeated step-size resizing described in Section 1.5.

A cold start means that we restart the solver with method parameters set so that the solver starts the computation with no values from the previous computation. It will also involve using a small initial step size and for methods of varying order like the 'BDF' and 'Adams' methods, they will restart with the default order which is order 1.

To solve the time dependent discontinuity problem, we will integrate from time 0 to the time that measures are implemented, t=27, with one call to the solver and then use the solution values at t=27 as the initial values to make another call that will integrate (restarting with a cold start) from t=27 to t_f. 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 to any problem where it is known when the discontinuity is introduced. **This is a much better approach than introducing a time-dependent 'if' statement into the model.**

2.3 Solving the time-dependent discontinuity model using a cold start

2.3.1 Solving the time-dependent discontinuity model in R using a cold start

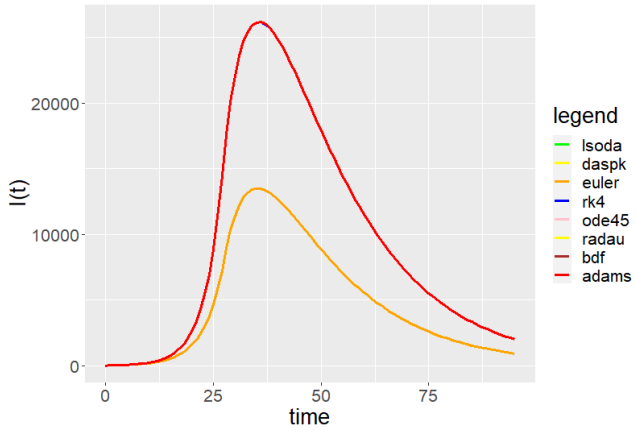

Figure 11: Solutions to the time-dependent discontinuity model using solvers from R and a cold start at t=27.

From Figure 11, we see that the 'euler' method still fails even when the cold start form of discontinuity handling is introduced. This is as expected as this method has no error control and thus it still suffers from accuracy issues and will require smaller steps to achieve even "eyeball" accuracy.

We see that breaking the integration into two parts allows 'rk4' perform better. The method has higher order but this exceptionally good performance is still unexpected. We will show in Figure 12 that the performance of 'rk4' is associated with the method of handling output points as described in Section 1.4.

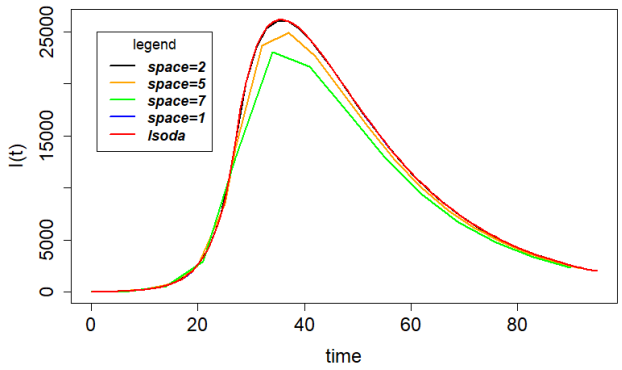

Figure 12: The R version of 'rk4' with larger spacings between the output points and with discontinuity handling.

Thus our recommendation to avoid fixed step size solvers still holds since users will not typically know how small the step size needs to be to obtain sufficient accuracy.

We also note again, that all the error-controlled solvers perform well. We will see, from the efficiency data, that using cold starts results in a more efficient computation. Using cold starts, the error control solvers do not have to step over the discontinuity and we will not have the spike in the number of function evaluations as we discussed in 1.5. Table 2 shows that discontinuity handling gnerally reduces the number of function evaluations.

Table 2: R efficiency data for the time-dependent discontinuity problem - number of function evaluations

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

Our analysis of the efficiency data in Table 2 starts by noting that the non-error controlled solvers in the 'euler' and rk4' methods have essentially the same number of function evaluations in both cases, the additional evaluation being due to evaluating the function twice at time 27. This indicates that they are

just stepping from output point to output point using the same fixed step-size both with and without the discontinuity handling.

Next, we note significant decreases in the number of function evaluations for all the remaining solvers except 'daspk'. These reductions in the number of function evaluations will have a significant impact on the CPU time for the difficult problem. This is entirely explained in Section 1.5 where the error-controlled solvers have to repeatedly resize the step-size as they encounter the discontinuity.

Finally, we explain the almost constant value of the number of function evaluations for the 'daspk' method through the fact that, in the R implementation, it is not using an appropriate interpolation scheme to obtain solution approximations at the output points. Instead it is using the approach described in Section 1.4. In another experiment with a larger spacing between output points, we found that 'daspk' uses 627 function evaluations without discontinuity handling and 522 function evaluations with discontinuity handling; a result that is more consistent with the results from Table 2 for the other error control solvers.

In Section 2.4, we will see that this type of discontinuity handling also allows us to use coarser tolerances, which improves the efficiency of the computation.

2.3.2 Solving the time-dependent discontinuity model in Python using a cold start

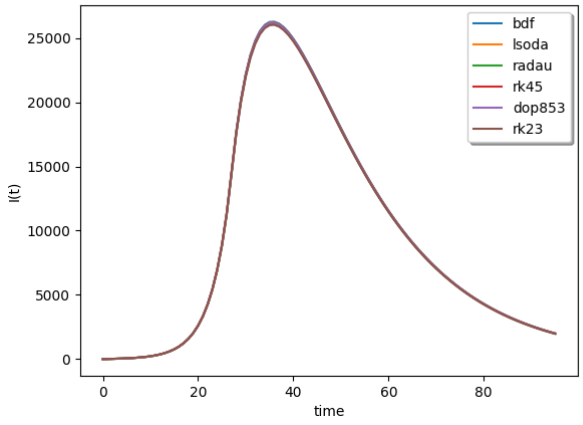

Figure 13: Solutions to the time-dependent discontinuity model using solvers from Python and a cold start at t=27.

The Python solvers did not have significant accuracy issues even without discontinuity handling. This is because all the available methods use error control and the default tolerances are sharp enough. From Figure 13, we can see that

the Python solvers again give sufficiently accurate results. Furthermore, the slight blurring at the peak has disappeared indicating that there is an even better agreement among the solvers. The addition of discontinuity handling also significantly reduces the number of function evaluations. This can be seen in Table 3.

Table 3: Python efficiency data for the time-dependent discontinuity problem - number of function evaluations

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

The Python solvers do not allow the space between the output points to affect the accuracy. They use some form of local interpolation within each step where there are output points.

From Table 3, we see that when discontinuity handling is introduced, the methods use fewer function evaluations. There are some significant improvements for 'BDF', 'DOP853' and 'Radau'. There are slight decreases for 'LSODA' and 'RK23' and only a very small decrease for 'RK45'.

2.3.3 Solving the time-dependent discontinuity model in Scilab using a cold start

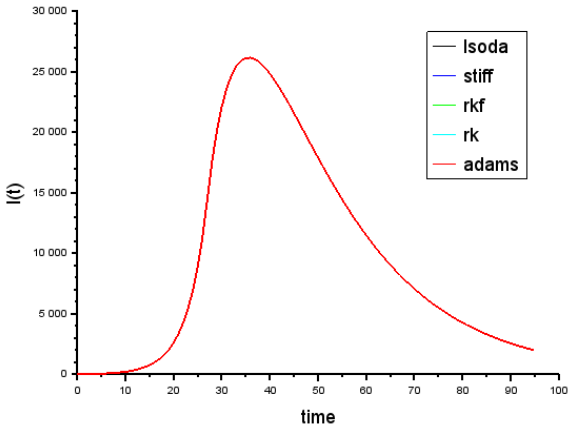

Figure 14: Solutions to the time-dependent discontinuity model using solvers from Scilab and a cold start at t=27.

We can see from Figure 14 that all the methods show good agreement and thus the time-dependent discontinuity model is being solved to a reasonable accuracy. The 'rkf' method is also giving reasonable results. This is despite 'rkf' having a coarser default tolerance.

The addition of discontinuity handling also significantly reduces the number of function evaluations as seen in Table 4.

Table 4: Scilab efficiency data for the time-dependent discontinuity problem - number of function evaluations.

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

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

The unusual result for 'rkf' occurs because 'rkf' is using the method for handling output points as outlined in Section 1.4. The results, when we space

out the output points more, are 335 function evaluations without discontinuity handling and 292 function evaluations with discontinuity handling.

We note that the high number of function evaluations in 'rk' with and without discontinuity handling is because it is using Richardson extrapolation to get an error estimate. Richardson involves using the Runge-Kutta method twice, once to get the solution approximation at the end of the step and once again with half the step-size to do two steps in the same interval to get a more accurate solution to use to obtain an error estimate. Thus in one actual step, there are three 'steps' and this leads to a large number of function evaluations.

2.3.4 Solving the time-dependent discontinuity model in Matlab using a cold start

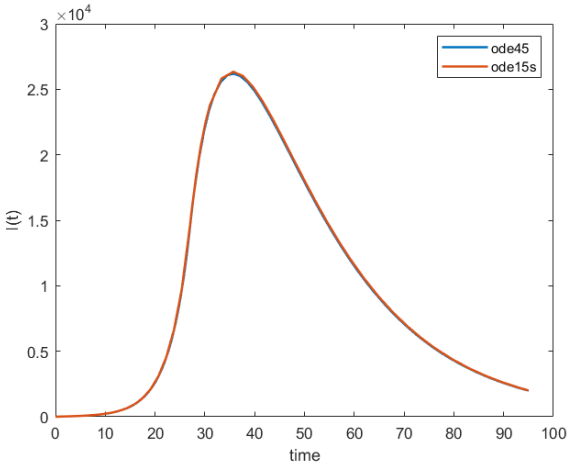

Figure 15: Solutions to the time-dependent discontinuity model using solvers from Matlab and a cold start at t=27.

From Figure 15 we can see that both solvers give similar solutions. We remember that with an 'if' statement inside the function $f(t, y(t))$, the two solvers gave somewhat different solutions. As we will show in Section 2.4, the discontinuity handling allows us to use a coarser tolerance and thus allows *ode*45 to give a reasonably accurate result.

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

Table 5: Matlab efficiency data for the time-dependent discontinuity problem - number of function evaluations

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

From Table 5, we see that *ode*45 uses 11 fewer function evaluations while *ode*15*s* uses 31 fewer function evaluations.

2.4 Efficiency data and tolerance study for the time-dependent discontinuity model

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

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

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

2.4.1 Comparing LSODA across platforms for the time-dependent discontinuity model

Time-dependent discontinuity LSODA tolerance study in R In this section, we run the R LSODA solver with multiple tolerances with and without discontinuity handling. We will set both the relative and absolute tolerances to various values and see how coarse we can set the tolerance while still obtaining reasonably accurate results. We also look at efficiency data to observe the number of function evaluations.

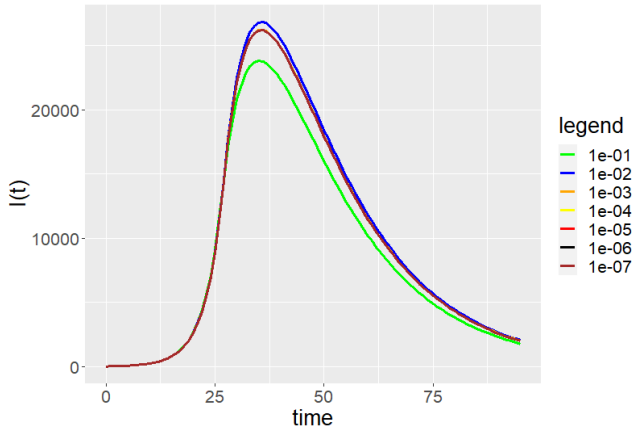

Figure 16: Time-discontinuity model tolerance study on the R version of LSODA without a cold start.

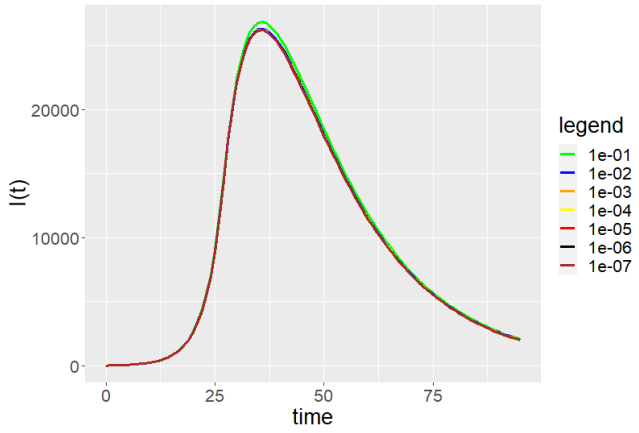

Figure 17: Time-discontinuity model tolerance study on the R version of LSODA with a cold start.

From Figures 16 and 17, we can see that the introduction of discontinuity handling allows the solver to use coarser tolerances and still get a reasonable result; we need a tolerance at least as sharp as 10^{-3} without discontinuity handling but can use a tolerance as coarse as 10^{-2} with it. This supports the observation that the use of discontinuity handling when solving a discontinuous problem is advantageous. Also, using coarser tolerances leads to better efficiency, as we will see in Table 6.

Table 6: The R LSODA time-dependent discontinuity model tolerance study - number of function evaluations

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

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

A similar number of function evaluations for the coarser tolerances should not distract us from the fact that the solver without discontinuity handling at these tolerances gives results that are not as accurate as the results obtained using the solver with discontinuity handling. The small differences of 3 function evaluations for the 0.1 tolerance case and 8 function evaluations in the 0.01 case do not excuse the fact that the solutions obtained when no discontinuity handling is employed are significantly less accurate.

Time-dependent discontinuity LSODA tolerance study in Python In this section, we run the Python version of the LSODA solver with multiple tolerances with and without discontinuity handling. We note that the Python solvers give sufficiently accurate results in both cases apart from some small disagreements in the case where no discontinuity handling is employed but we will see how coarse we can choose the tolerance while still obtaining reasonably accurate results. We set both the relative and absolute tolerances to various values. We also look at efficiency data to see the decreases in the number of function evaluations.

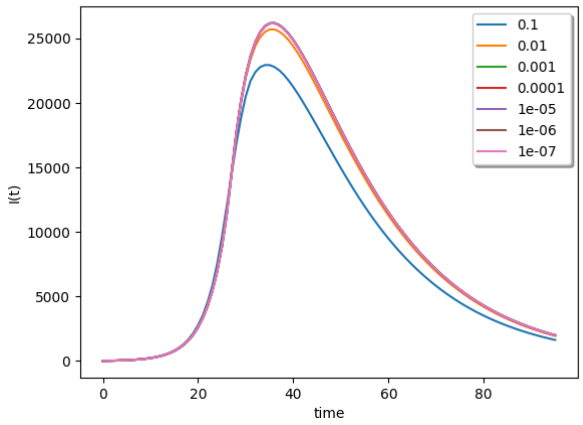

Figure 18: Time-dependent discontinuity model tolerance study on the Python version of LSODA without a cold start.

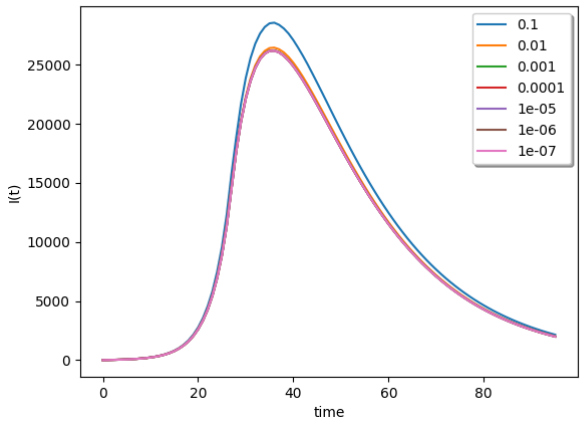

Figure 19: Time-dependent discontinuity model tolerance study on the Python version of LSODA with a cold start.

From Figures 19 and 18, we see that with the use of the discontinuity handling, a tolerance of 10^{-2} is enough to get a reasonably accurate result whereas a tolerance of 10^{-3} is needed otherwise. Also, the use of coarser tolerances leads to better efficiency, as can be seen in Table 7.

Table 7: Python LSODA time-dependent discontinuity model tolerance study - number of function evaluations

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

Again, in Table 7, we see that that at coarse tolerances, the number of function evaluations is roughly the same. This similar number of function evaluations does not excuse the fact that the coarser tolerances are giving inaccurate solutions when discontinuity handling is not employed.

At sharper tolerances, where solutions of reasonable accuracy are obtained in all cases, the number of function evaluations is much smaller with discontinuity handling than without. There are 40 fewer function evaluations at 0.001 and 0.0001 and there are substantially fewer function evaluations for sharper tolerances. We note that if the function for the evaluation of the right-hand side of the ODE was more time-consuming, this reduced number of function evaluations will cause a significant decrease in the CPU times.

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

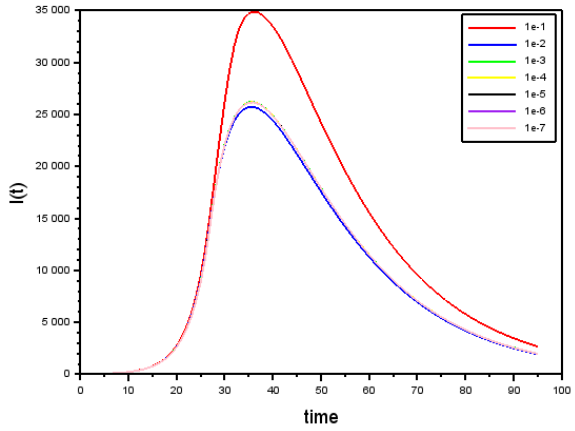

Figure 20: Time-dependent discontinuity model tolerance study on the Scilab version of lsoda without a cold start.

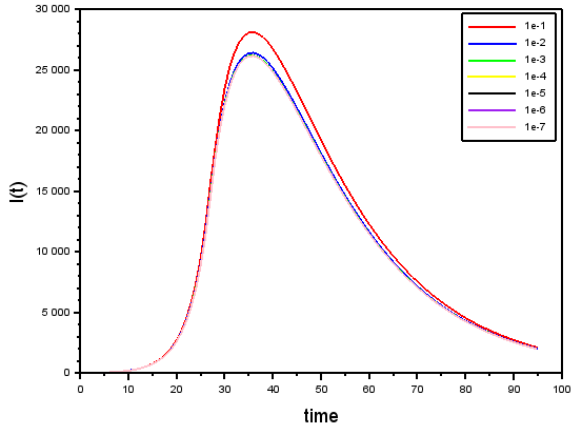

Figure 21: Time-dependent discontinuity model tolerance study on the Scilab version of lsoda with a cold start.

From Figures 20 and 21 we can see that for tolerances from 10^{-1} to 10^{-4}, the Scilab version of LSODA without discontinuity handling does not yield reasonably accurate solutions but we are able to use a tolerance as coarse as 10^{-2} with discontinuity handling.

It is interesting to see how inacurate the solution without discontinuity handling is at a tolerance of 10^{-1}. We also note that this behavior is different from

the R and the Python version LSODA but this may be due to the way Scilab handles the tolerances.

Table 8: Scilab LSODA time-dependent discontinuity model tolerance study - number of function evaluations

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

Again, in Table 8, we see that the number of function evaluations is roughly the same at coarser tolerances but that at sharp tolerances, where both types of computations give reasonably accurate solutions and thus allow for a fair comparison, the solver with discontinuity handling performs better than the solver without discontinuity handling. We can use up to 90 fewer function evaluations through the use of discontinuity handling.

2.4.2 Comparing solvers based on Runge-Kutta pairs across platforms for the time dependent discontinuity problem

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

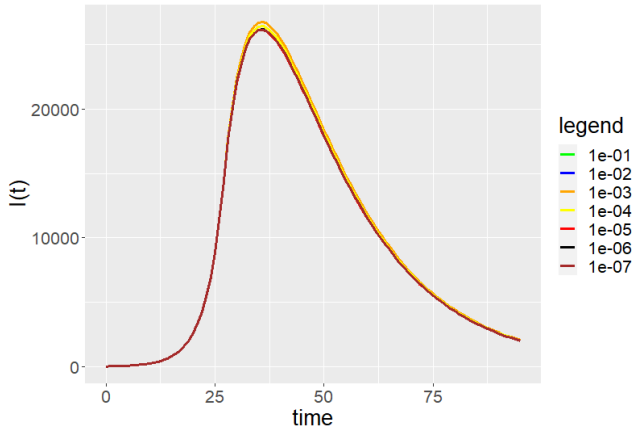

Figure 22: Time-dependent discontinuity model tolerance study on the R version of DOPRI5 without discontinuity handling.

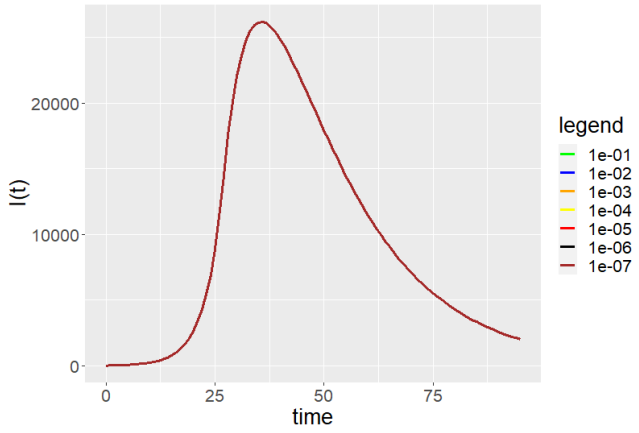

Figure 23: Time-dependent discontinuity model tolerance study on the R version of DOPRI5 with discontinuity handling.

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

However, as we will see in the Python version of DOPRI5, the results from Figures 22 and 23 are suspicious and stem from the fact that R is not using a proper interpolation scheme to produce the results. It is using an algorithm that depends on the selected output points and which affects efficiency and accuracy, as discussed in Section 1.4.

Table 9: The R DOPRI5 time-dependent discontinuity model tolerance study - number of function evaluations

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

Table 9 also confirms our suspicions since, at coarser tolerances, 10^{-1} to 10^{-3}, the number of function evaluations does not change at all. This indicates that something else, not the tolerance nor the discontinuity, is the limiting factor for the number of function evaluations and that this other factor leads to a need for around 572 or 574 function evaluations.

We suspect that the R DOPRI5 version is not using an appropriate interpolation scheme to evaluate the numerical solution and that it is integrating using the output points to determine the step-size. We therefore perform the following experiment where we specify a smaller set of output points with the points further spaced out from each other.

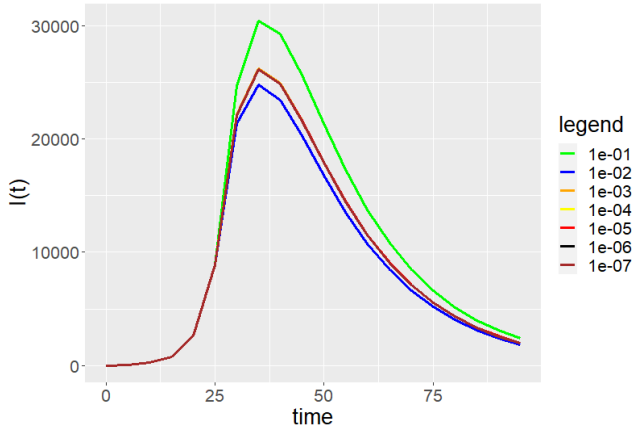

Figure 24: Time-dependent discontinuity model tolerance study on the R version of DOPRI5 without discontinuity handling and with output points more spaced out.

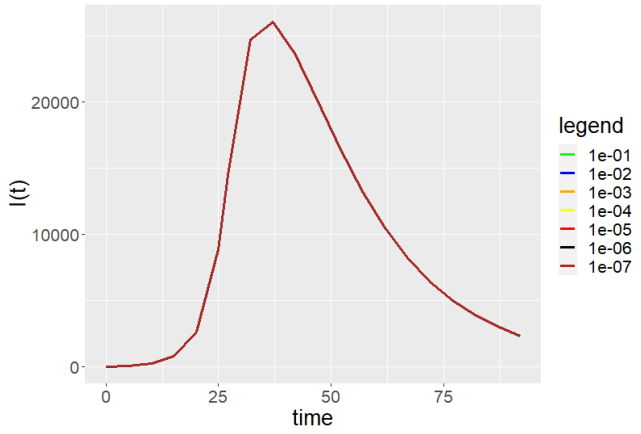

Figure 25: Time-dependent discontinuity model tolerance study on the R version of DOPRI5 with discontinuity handling and with output points more spaced out.

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

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

Table 10: R DOPRI5 time-dependent discontinuity model tolerance study with spaced out output points - number of function evaluations

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

Our analysis of Table 10 begins by noting that the set of output points is no longer a limiting factor. We can see that the number of function evaluations changes with the tolerance and this indicates that the tolerance is controlling the step-size. This confirms our suspicion that the R implementation of DOPRI5 is not using an appropriate scheme for treating the output points. Instead, it is

allowing the output points determine the step-size and thus dictate the efficiency of the solver.

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

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

Time dependent discontinuity model tolerance study on the Python version of DOPRI5 In this section, we run the Python 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 various values and see how coarse we can choose the tolerance while still obtaining reasonably accurate results. We also look at efficiency data to determine the number of function evaluations in each case.

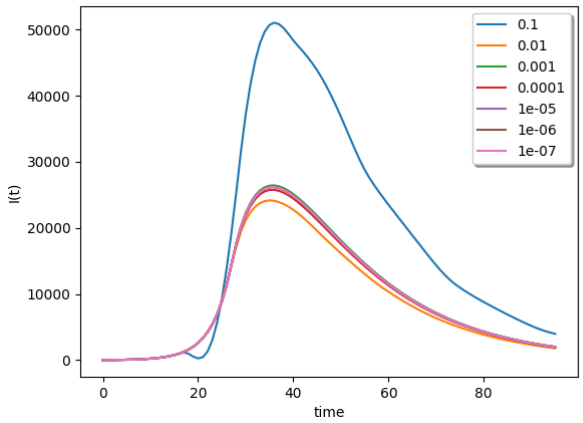

Figure 26: Time-dependent discontinuity model tolerance study on the Python version of DOPRI5 without discontinuity handling.

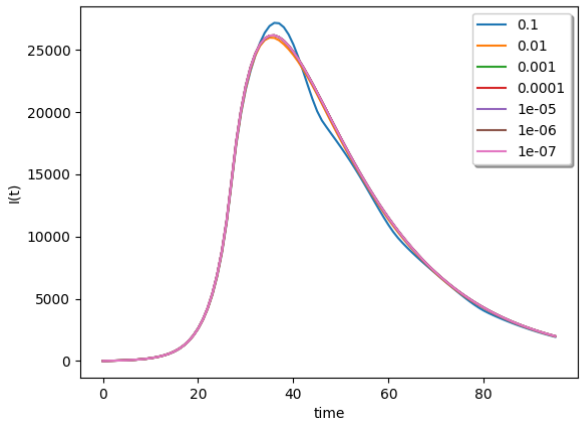

Figure 27: Time-dependent discontinuity model tolerance study on the Python version of DOPRI5 with discontinuity handling.

From Figures 27 and 26, we can see clear differences in the computed solutions at different tolerance values. From studying Python's *solve_ivp* interface and source code, we note that Python is using interpolation to treat the output points.

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

Table 11: Python DOPRI5 time-dependent discontinuity model tolerance study - number of function evaluations

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

From Table 11, we see that at coarser tolerances, the number of function

evaluations is greater with the discontinuity handling than without discontinuity handling but we must point out that DOPRI5 at coarse tolerances gives very inaccurate results; the errors are too large to excuse the small gain in efficiency.

At sharper tolerances where we get reasonably accurate results both with and without discontinuity handling, and thus a fair comparison can be done, we can see that the solver that uses discontinuity handling performs much better. At a tolerance of 10^{-5} or sharper, the decrease in the number of function evaluations is 75 or more.

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

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

We will also see that using discontinuity handling leads to the use of fewer function evaluations which, given a more complex problem, would result in a significant improvement in computation times.

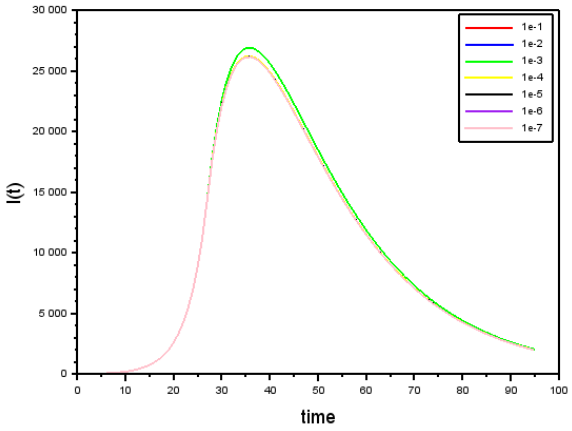

Figure 28: Time discontinuity model tolerance study on the Scilab version of RKF45 without discontinuity handling.

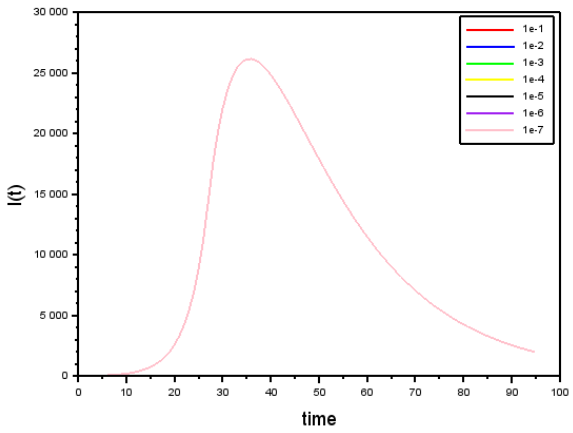

Figure 29: Time discontinuity model tolerance study on the Scilab version of RKF45 with discontinuity handling.

We see from Figure 28 that using 10^{-4} for both the absolute and the relative tolerance gives reasonably accurate answers and that anything coarser leads to somewhat inaccurate solutions. We then recall that the relative tolerance defaults to 10^{-3} and the absolute tolerance defaults to 10^{-4} for 'rkf' which is slightly coarser than what is needed to get a reasonably accurate solution.

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

Table 12: Scilab RKF45 time-dependent discontinuity model tolerance study - number of function evaluations

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

We can see from Table 12 that the Scilab RKF45 method is not using interpolation to treat the output points. We can make this conclusion because at extremely coarse tolerances, it is using the same number of function evaluations

despite the tolerance. There is also no difference with and without discontinuity handling. We also note that a change in the tolerance did not lead to a change in the number of function evaluations and thus something else is determining the number of function evaluations. Doing the same experiment with the points further spaced out shows us that it is the spacing of the output points that is causing the issue. We thus replicate the experiments in the previous sections with the output points more spread out.

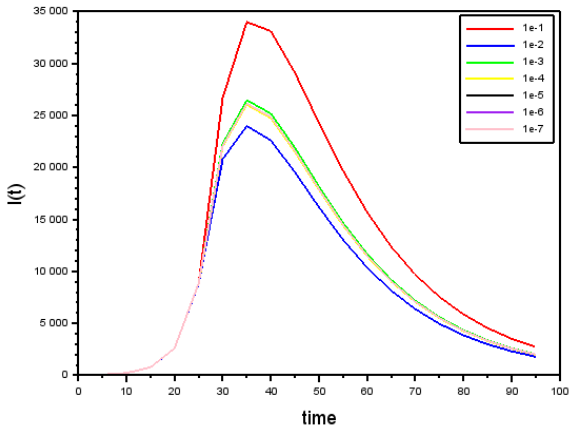

Figure 30: Time discontinuity model tolerance study on the Scilab version of RKF45 without discontinuity handling.

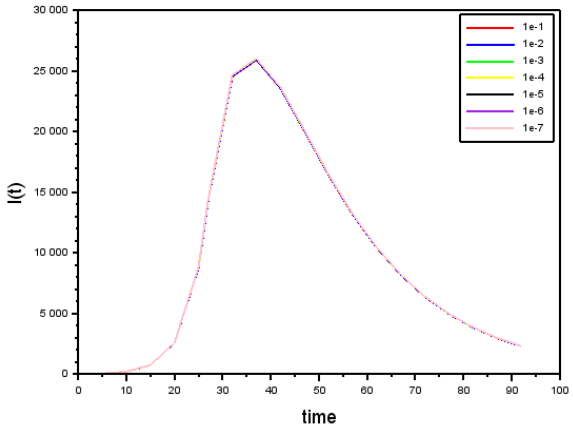

Figure 31: Time discontinuity model tolerance study on the Scilab version of RKF45 with discontinuity handling.

Figures 30 and 31 show a clear indication regarding why discontinuity handling is important. We can see that without it, we need a tolerance of 10^{-3} to get reasonably accurate results but with the discontinuity handling, we can use a tolerance of 10^{-1}. The impact on the number of function evaluations, shown in Table 13, is clear.

Table 13: Scilab RKF45 with spaced out ouptut points, time-dependent discontinuity model, tolerance study - number of function evaluations

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

Table 13 shows that the number of function evaluations, when discontinuity handling is employed, is smaller. We also note that at coarse tolerances, the number of function evaluations is similar but that at those tolerances, the solver without discontinuity handling is not obtaining reasonably accurate results. We can thus conclude that using discontinuity handling lets us use coarser tolerances and leads to a smaller number of function evaluations while improving accuracy.

Time-dependent discontinuity model tolerance study on the Matlab version of DOPRI5 We perform the same experiment using *ode*45 in Matlab. We set both the absolute and relative tolerance to various values and examine how the solver perform. We recall that, using the default tolerance, *ode*45 did not give a reasonably accurate solution. We also recall that *ode*45 did not have a smaller default tolerance than *ode*15*s*. In this section, we show that with a sharper tolerance, *ode*45 is also capable of solving the problem without discontinuity handling but we will see that it is more efficient with discontinuity handling. Discontinuity handling will, again, allow us to use coarser tolerances and still obtain reasonably accurate solutions.

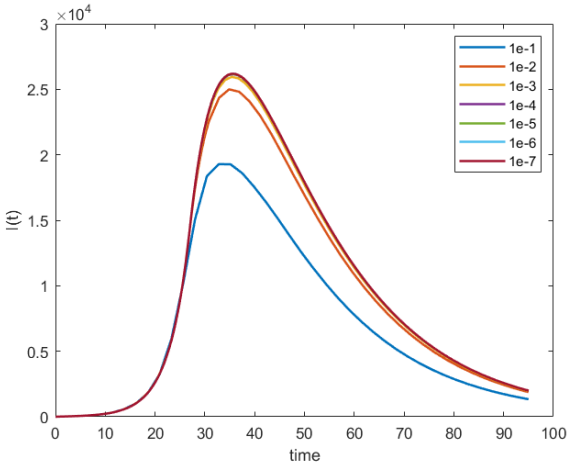

Figure 32: Time discontinuity model tolerance study on the Matlab version of DOPRI5 without discontinuity handling.

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

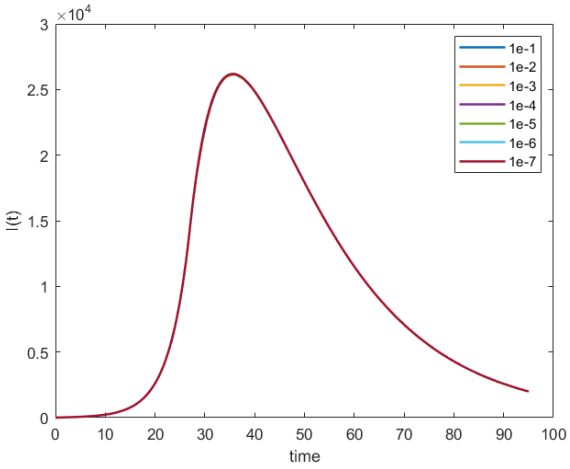

Figure 33: Time discontinuity model tolerance study on the Matlab version of DOPRI5 with discontinuity handling.

From Figures 32 and 33 we see that discontinuity handling allows us to use coarser tolerances while still getting a reasonably accurate solution. We note that we could use a tolerance of 10^{-1} with discontinuity handling but we had to use a tolerance of 10^{-3} to get a reasonably accurate solution when no

discontinuity handling is employed. We will also see that discontinuity handling allows the solver to use fewer function evaluations in Table 14.

Table 14: The Matlab DOPRI5 time-dependent discontinuity model tolerance study - number of function evaluations

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

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

3 State-dependent discontinuity model

In this section, we consider the state-dependent discontinuity problem. We start by noting that this problem cannot be solved with the form of discontinuity handling used in the previous study since we do not know when the discontinuity arises. Also, this problem will be more challenging than the time-dependent discontinuity problem as the parameter β will be changed more than once as we attempt to model the periods of imposition of Covid-19 measures followed by periods where these measures are removed. As in Section 2, changes in the modelling parameter β introduce discontinuities in the function $f(t, y(t))$ and thus the error control solvers will "thrash" when trying to solve the problem (as described in Section 1.5).

This model uses the state variable, E(t), the number of exposed people, to determine when to change the parameter β. When the number of exposed people is greater than 25000, measures will be introduced and β will change from 0.9 to 0.005. When the number of exposed people drops to 10000, the measures will be relaxed and β is set back to 0.9 (corresponding to the case where vaccinations are not available). We will run this model over a longer time period toggling the parameter β back and forth to model the periods of alternating the imposition and relaxing of the measures. This scenario corresponds to the case of an unvaccinated population where the only means of controlling the spread of the virus is through measures such as social isolation, masking, etc. The ability of the virus to infect people is not diminished as time progresses, and when measures to stop the spread of the virus are removed, the infection rate of the virus returns to its original value. More sophisticated models could be considered by treating the β parameter as a function of time. The numerical challenges would be similar.

We start with a simple treatment of the problem with 'if' statements employed inside the function that defines the right-hand side of the ODE system and show how this form of the problem cannot be solved with reasonable accuracy, by any of the solvers, even at sharp tolerances. Finally, we will introduce an approach to efficiently and accurately solve the problem using an approach involving the use of what is known as event detection to handle the discontinuities.

3.1 Simple treatment of Covid-19 state-dependent discontinuity model

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

```
measures_implemented = False
direction = "up"

function model_with_if(_, 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
    // ...
    return (dSdt, dEdt, dIdt, dRdt)
```

3.1.1 Simple solution of the state-dependent discontinuity model in R

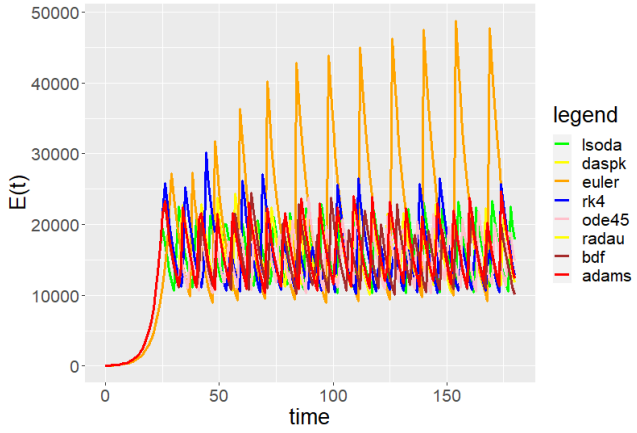

Figure 34: Solutions to the state-dependent discontinuity model in R, based on the simple approach.

In Figure 34, we show the results from the use of a number of solvers in R based on the simple implementation described above, using default tolerances. Figure

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

We note that none of the solvers, even the error-controlled ones, issued a warning about the integration and thus users may be tempted to think that the solver has solved the problem to within reasonable accuracy. Having no warning also tells us that the error estimation and error control algorithms employed by all the solvers did not detect anything abnormal; the solvers return with an indication that the provided solutions are accurate to within the requested tolerance. We would expect the solvers with error control to repeatedly reduce the step-size to satisfy the tolerance and compute solutions that align with each other but Figure 34 shows that this is not the case.

We also note that the result for 'euler' is especially poor as it reaches a maximum of 40000. This is again as expected as 'euler' has no error control; 'rk4', the other fixed step-size method, is also performing poorly; we see the solution it computes reach approximately 30000 in its third peak. This is happening even though the space between the output points is as small as it was when we were investigating the time-dependent discontinuity problem. Because of this, we will not run any spacing of output points experiments in this section.

Another important fact to note is how poorly 'Radau', as shown in Figure 35, performs. This is not an issue with the R programming environment as similar results will be seen in when we consider the Python version of this solver in the next section and the original Fortran version of this solver in Section 5. The solution grows exponentially even after the parameter β is switched to 0.005, which should force the solution to begin to decay. We perform an analysis with the Fortran version of the solver later in this report to show that β is indeed 0.005 while this exponential growth is happening.

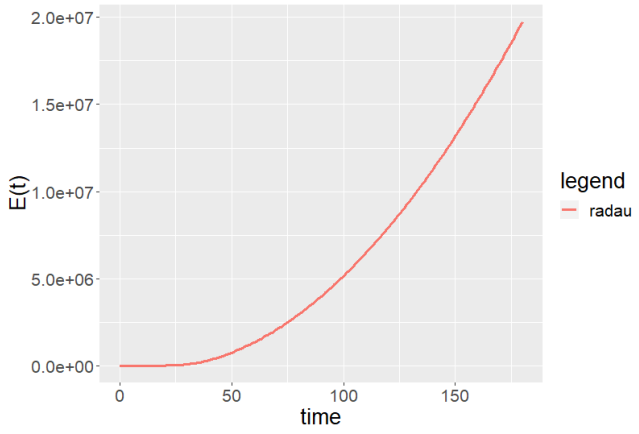

Figure 35: Solution from 'Radau' for the state-dependent discontinuity model in R, based on the simple approach.

We next proceed to show that sharp tolerances are not enough to solve this problem as was the case for the time-dependent discontinuity problem. We repeat our experiments at the sharpest tolerance that could be used prior to some of the solvers failing. This was at 10^{-13} in the R environment. We set both the absolute and relative tolerance to that value and show the results in Figure 36.

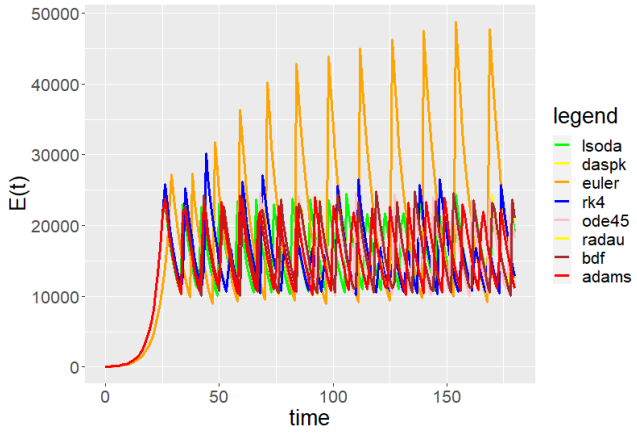

Figure 36: Solutions to the state-dependent discontinuity model in R with a sharp tolerance, using the simple approach.

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

The fixed step-size method 'euler' and 'rk4' results are the same as in Figure 34 since these solvers do not employ a tolerance.

At sharp tolerances 'Radau' no longer computes solutions exhibiting the abnormal behavior we saw previously. From Figure 37, we can see that the solution computed by 'Radau' oscillates approximately between 10000 and 25000. From supplementary experiments, we observe that 'Radau' starts performing at a level that is comparable to the other solvers at a tolerance of 10^{-9} or sharper.

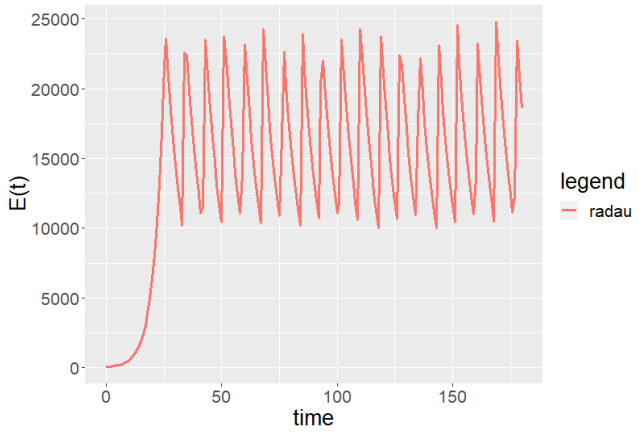

Figure 37: Solution from 'Radau' for the state-dependent discontinuity model in R with a sharp tolerance, using the simple approach.

3.1.2 Simple solution of the state-dependent discontinuity model in Python

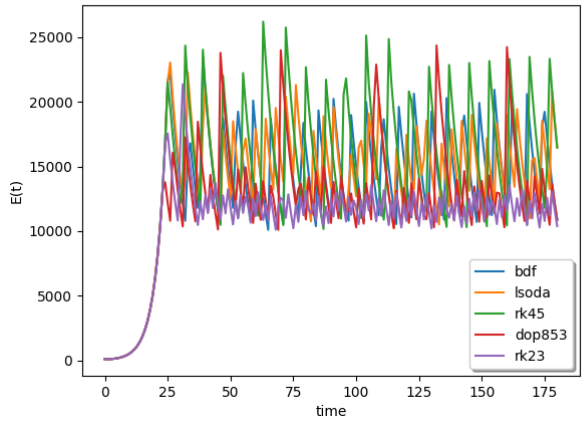

Figure 38: Solutions to the state-dependent discontinuity model in Python, based on the simple approach.

Figure 38 shows what happens when the problem is solved using the simple implementation and default tolerances in Python. We can see that the results are similar to those obtained in R. This happens even though all solvers in Python have error control.

We note that all the solvers except 'RK23' give solutions that at least oscil-

late between 10000 and 25000, though in completely dissimilar patterns. The solutions have peaks and troughs at different times. No warnings were given by the solvers.

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

Again, as shown in Figure 39, 'Radau' computes a solution that has $E(t)$ growing exponentially even though the parameter β is eventually set to 0.005 which should give a solution with an exponential decay in the $E(t)$ component, as we see with all other solvers.

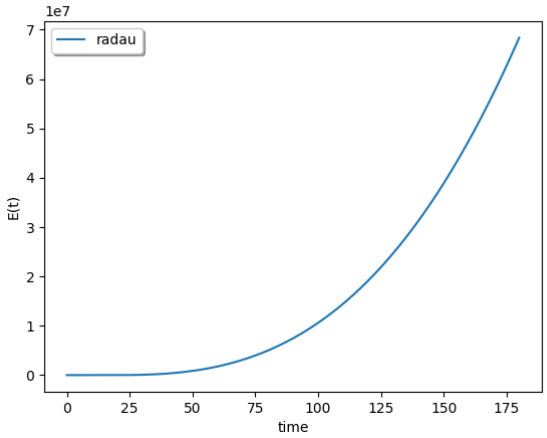

Figure 39: Solution from 'Radau' for the state-dependent discontinuity model in Python, based on the simple approach.

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

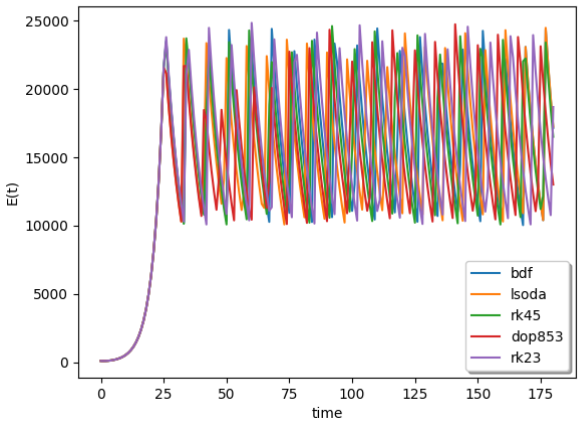

Figure 40: Solutions to the state-dependent discontinuity model in Python with a sharp tolerance, using the simple approach.

Figure 40 shows that the results have improved. However, the solvers give solutions that are not in agreement. We note that none of the solvers are oscillating beyond 25000 as was the case with the fixed-step solvers in R. At sharp tolerances, the solutions are aligned for the first few discontinuities with only some blurring until about t=25 when the solvers give substantially different solutions. Though the pattern is correct, none of the solvers give solutions that are in agreement telling us that none were able to compute a reasonably accurate solution such as the one that we present in Section 3.4. (See the comparison with the final solution in Section 3.1.5 to see that even these sharp tolerance solutions are not accurate enough.)

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

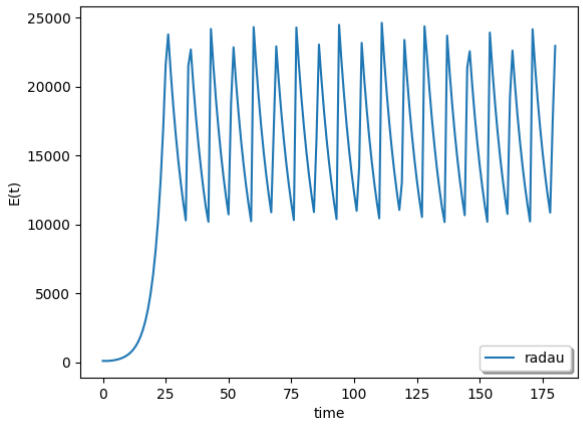

Figure 41: Solution from 'Radau' for the state-dependent discontinuity model in Python with a sharp tolerance, using on the simple approach.

Again, as shown in Figure 41, the 'Radau' solver begins to give reasonable solutions at these sharp tolerances; the solutions follows the pattern we are expecting but as we will show in Section 3.4, they are still not sufficiently accurate. The 'Radau' solver starts performing reasonably well at around a tolerance of 10^{-10}. We also note that the R and Python implementation of 'Radau' are different. The 'Radau' solver in Python is implemented in Python with the NumPy library whereas R calls the Fortran version of the solver. Thus we eliminate the possibility of an issue stemming from the interface from R to Fortran or from Python to NumPy. The problem is simply in how the Radau algorithm interacts with this simple implementation of the state-dependent discontinuity. In our experiments with the Fortran version of Radau, in Section 5, the same behavior is observed.

3.1.3 Simple solution of the state-dependent discontinuity model in Scilab

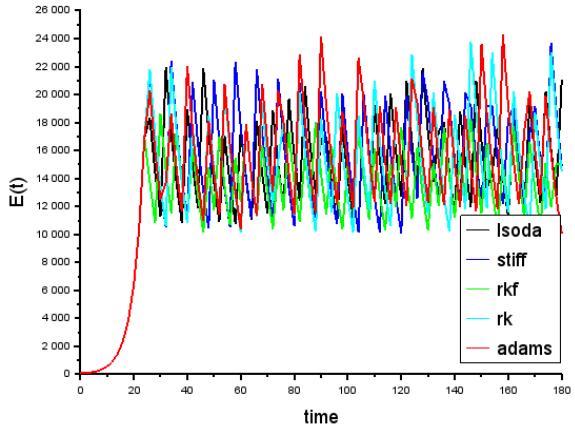

Figure 42: Solutions to the state-dependent discontinuity model in Scilab, based on the simple approach.

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

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

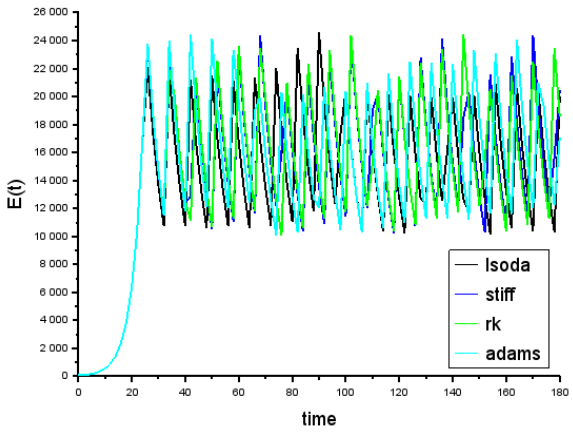

Figure 43: Solutions to the state-dependent discontinuity model in Scilab with a sharp tolerance, using the simple approach.

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

3.1.4 Simple solution of the state-dependent discontinuity model in Matlab

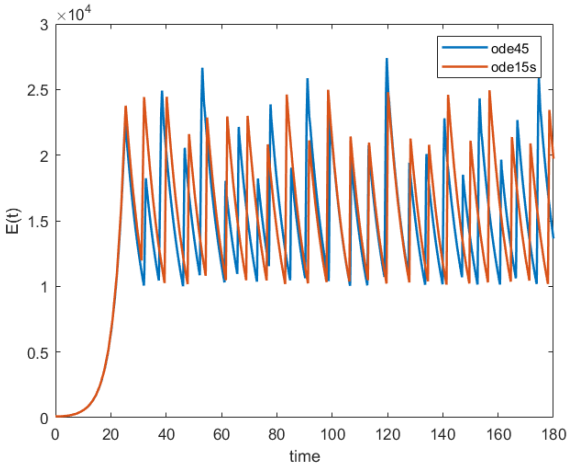

Figure 44: Solutions to the state-dependent discontinuity model in Matlab, based on the simple approach.

In Figure 44, we see in accurate solutions, similar to what we saw in the previous experiments, when the solvers are run with the simple implementation, at the default tolerances. The solvers do not give solutions that consistently reach 25000. We then use a sharper tolerance to see if the solutions are improved.

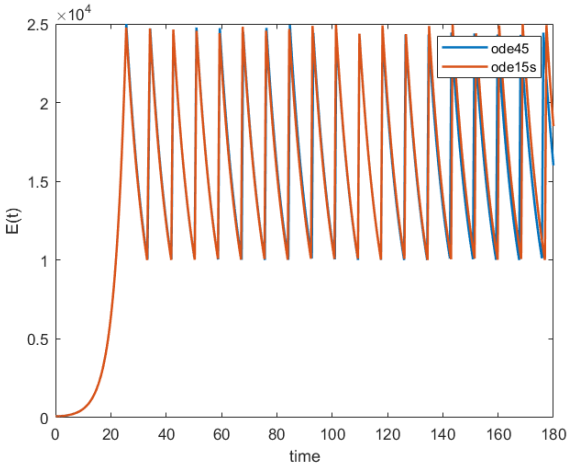

Figure 45: Solutions to the state-dependent discontinuity model in Matlab with a sharp tolerance, using the simple approach.

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

3.1.5 State-dependent discontinuity model - solution comparisons

In all the previous subsections, we have maintained that even the sharp tolerance solutions, though more in agreement, are not accurate. Here, we present a comparison between the solution obtained by LSODA in Python using the simple approach at the default tolerance and at the sharpest tolerance, alongside an accurate solution that we will present shortly which is obtained using event detection. We can see from Figure 46 that the solutions from LSODA both at default and the sharp tolerance obtained using the simple approach do not agree with the more accurate solution.

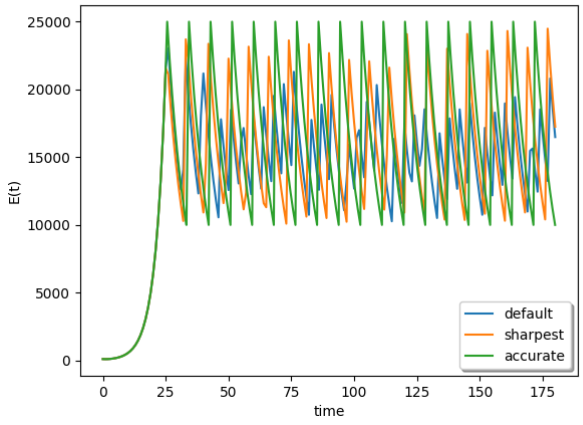

Figure 46: Solutions to the state-dependent discontinuity model from LSODA based on the simple approach using the default tolerance and a sharp tolerance, alongside an accurate solution.

3.2 Why the solvers fail even with sharp tolerances

In this section we discuss why sharp tolerances were not enough to force the solvers to accurately solve the problem in the simple way the the model is implemented, i.e, using global variables and 'if' statements.

Whenever there is a change in the value of β, the step where the discontinuity is first encountered will almost always be a failed step. As discussed in Section

1.5, the step-size required to accurately step through a discontinuity will always have to be much smaller than the step-size on the continuous region to the left of the discontinuity. Thus the first encounter of a solver with any discontinuity will always be in the context of a failed step.

During this failed step, the value of E(t) will cross the threshold. The global variables will thus be toggled. But then, when the solver attempts to retake the step using a smaller step-size, to the left of the discontinuity, it will be using the wrong β value.

This observation is crucial as it allows us to conclude that once a failed step has occurred due to the solver encountering a discontinuity, the function evaluations made to the left of the discontinuity should be based on the previous β value but they are in fact obtained using the new β value. There is no trivial way to implement the model in a way that avoids this behavior in the ODE function, $f(t, y(t))$, since the time at which the discontinuities arise is unknown.

In summary, the issue is that the solvers need to figure out how to step up to the discontinuity such that to the left of the discontinuity, the solver employs function evaluations that use the previous β, and then after the discontinuity, the solver employs function evaluations that use the new β value. This cannot be implemented in a straightforward way using the interfaces available in the programming environments.

In the next few sections, we will present a better approach for treating problems with state-dependent discontinuities that will allow us to get reasonably accurate solutions in an efficient manner.

3.3 Event detection

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

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

To perform event detection, an ODE solver requires two functions from the user: the usual ODE right-hand side function, $f(t, y(t))$, and another function,

the root function (commonly denoted by $g(t, y(t))$), that defines an event.

The root function is a function that, given the value of the solution $y(t)$ at time t to the ODE at the current step will return a value. The event function, $g(t, y(t))$, is said to have a root whenever the value of the root function is zero. The key idea is that each event must be written so that it occurs at the root of a root function.

The solver calls the root function at the end of each successful step and records its value. It will then compare the value of the root function with the corresponding value from the previous step to see if there has been a change of sign. If the value of the root-function has changed sign, the solver will then run a root-finding algorithm on that step to find the point where the root-function equals zero. The solver will then return, allowing us to perform a cold start.

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

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

3.4 Solving the state-dependent discontinuity model using event detection

As mentioned earlier, each change in the value of the parameter β introduces a discontinuity in the function $f(t, y)$. Since none of the solvers are designed to solve discontinuous problems, they return the inaccurate solutions reported in 3.1. We have seen that although sharp tolerances do result in somewhat better solutions being computed, none of the solvers were able to obtain a sufficiently accurate solution. The use of such sharp tolerances leads to inefficiencies as well. We will now present an approach using event detection that is both accurate and efficient.

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

For our specific problem, event detection is used as follows. We start by solving the problem with β=0.9 and with a root function that detects when E(t) is equal to 25000. Once, using the event detection capability of the solver, we reach the time at which $E(t) = 25000$, we do a cold start. We evaluate the

solution computed by the solver at the time of the event and use that solution as the initial value for our next call to the solver. This next call will have $\beta = 0.005$ and a root function that detects a root when $E(t) = 10000$. We again integrate up to that new threshold and cold start when we reach it. The new integration will have β=0.9 and the root function will look for $E(t) = 25000$ as the event. This is repeated until we reach the desired end time. The pseudo-code is as follows:

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

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

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

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

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

// use res as the final solution
```

Some programming environments, such as Python, by default, do not stop the integration when the first event is detected. To do a cold start, we need

the solver to stop at events, and to make this happen, in some programming environments we need to set appropriate input parameters.

3.4.1 Solving the state-dependent discontinuity model in R using event detection

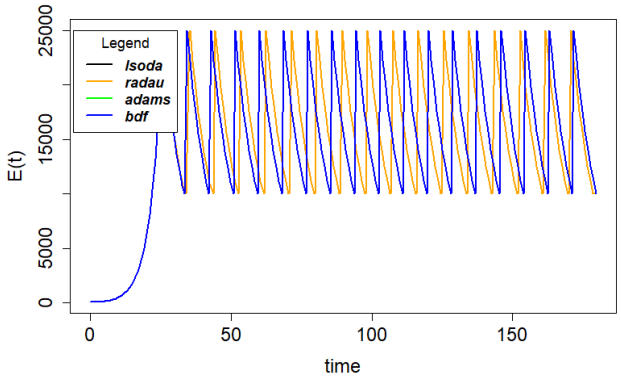

Figure 47: Solving the state-dependent discontinuity model in R with event detection.

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

The case of 'Radau' is interesting as it was giving a poor quality solution at the default tolerances, without event detection but it is now giving at least a solution that is exhibiting the correct pattern. We note that at sharp tolerances 'Radau' with event detection gives results that approach the results from the other solvers, as shown in Figure 48. We will also note the poor performance of Radau in Table 15. We also note that Fortran version of 'Radau' does not have built-in event detection and that the event detection has been added through the C interface, which may explain the disparity in performance that we see for that solver.

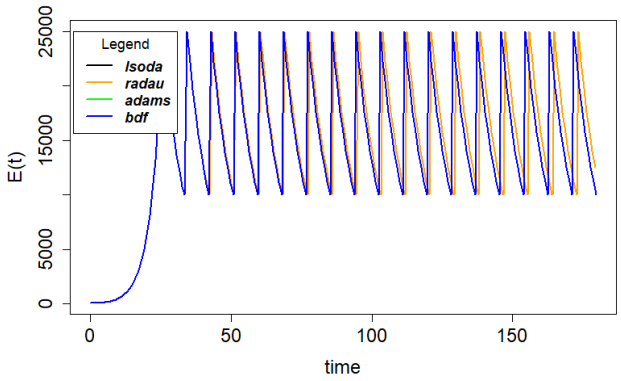

Figure 48: Solving the state-dependent discontinuity model in R with event detection at a sharp tolerance

We will show in Table 15 that introducing event detection also makes the computation significantly more efficient while giving us more accurate results.

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

Table 15: Efficiency data for R state-dependent discontinuity model - number of function evaluations

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

We can see from Table 15 that with event detection we are gaining an improvement of around 1000 function evaluations for 'lsoda', 7000 in 'Radau' (sharp tol comparison), 2000 in 'bdf', and 500 in 'adams' while having more accuracy. This significant decrease in the number of function evaluations will lead to much faster CPU times, especially when the right-hand side function is more complex.

Also, we can see from the table that the solvers use fewer function evaluations compared with event detection than without event detection at the default tolerances. When comparing the values at the sharp tolerances, the use of event detection also led to a decreased number of function evaluations.

3.4.2 Solving the state-dependent discontinuity model in Python using event detection

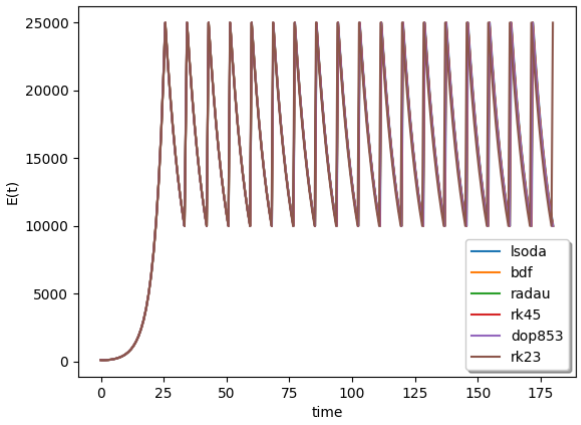

Figure 49: Solving the state-dependent discontinuity model in Python using event detection.

All the solvers in Python have event detection and thus all will be used in this part of the study. In Python, *solve_ivp*() does not stop when an event is detected by default. We thus need to set the terminal flag of the root functions. (Example: $root\_10000.terminal = True$). Again, Figure 49 shows that all the solvers give solutions that are in agreement, suggesting that this is the correct solution. This is different from our results at sharp tolerances when event detection was not employed. We will also see that this is a much more efficient approach across all the solvers. The *solve_ivp*() implementation of 'Radau' is in Python itself and thus it is different from the R implementation. We note that we did not have to provide the Python 'Radau' implementation with a sharp tolerance to make its performance align with the other solvers' performances, suggesting that the issue in R may be due to the C implementation of event detection.

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

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

Table 16: Efficiency data for Python state-dependent discontinuity model - number of function evaluations

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

3.4.3 Solving the state-dependent discontinuity model in Scilab using event detection

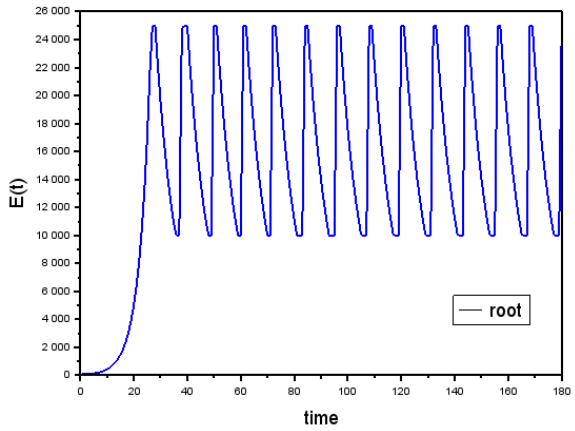

Figure 50: Solving the state-dependent discontinuity model in Scilab with event detection.

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

From Table 17, we can see that 'lsoda', using its event detection mode, uses fewer function evaluations than when it does not use event detection, both at sharp and default tolerances.

Table 17: Efficiency data for Scilab state-dependent discontinuity model - number of function evaluations

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

3.4.4 Solving the state-dependent discontinuity model in Matlab using event detection

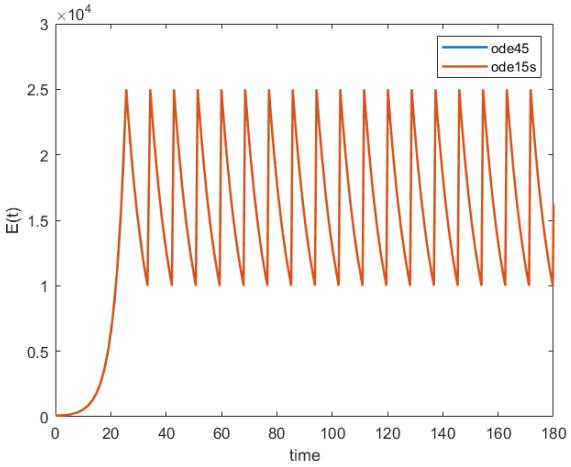

Figure 51: Solving the state-dependent discontinuity model in Matlab with event detection.

Both *ode*45 and *ode*15*s* have an event detection capability. (The root functions need to set an input parameter to indicate that the root is terminal in order to allow a cold start to be performed.) We applied event detection to solve the problem with the solvers in the Matlab environment and the results are shown in Figure 51. We remember that the solutions in Matlab without event detection were surprisingly accurate but were in disagreement with each other at points further in time. We can see that with event detection, the solutions are all in agreement at the default tolerances even at points further in time.

We can see in Table 18 that the computation with event detection uses fewer function evaluations than the computation without event detection at default and sharp tolerances. We see that the computations with sharp tolerances, although they give acceptable solutions, use 20000 more function evaluations for *ode*45 than the computation with event detection and 11000 in the case of *ode*15*s* than the computation with event detection.

Table 18: Efficiency data for Matlab state-dependent discontinuity model - number of function evaluations

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

3.5 Efficiency data and tolerance study for the state-dependent discontinuity model

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

We will perform this analysis on LSODA across R, Python, and Scilab, as they appear to use the same source code, and with R and Python versions of DOPRI5 which do not use the same algorithm but do use the same Runge-Kutta pair and with the Scilab version of RKF45 which does not use the same algorithm, nor the same pair, but does use a Runge-Kutta pair of the same order. We also use *ode*45 in Matlab as it is an implementation of DOPRI5 in Matlab.

3.5.1 Comparing LSODA across platforms for the state-dependent discontinuity model

State-dependent discontinuity LSODA tolerance study in R In this section, we use the R version of LSODA at multiple tolerances. We set both the relative and the absolute tolerance to various values and examine the solutions.

Figure 52 shows that LSODA without event detection applied at different tolerances gives vastly different results. We would expect the solutions at the sharper tolerances to be along very similar curves but that is not the case. This further supports our statement that for any state-dependent discontinuity, we cannot get reasonable results simply by sharpening the tolerance.

From Figure 53, we can see the clear advantage of using event detection. Event detection allows us to use a tolerance of 10^{-3} to get reasonable results while the computation without event detection failed even at a tolerance of 10^{-13}. We also analyze the differences in efficiency between the two modes of operation of LSODA in Table 19.

Table 19 shows a decrease in the number of function evaluations when event detection is employed across all tolerances which will translate into faster CPU times when the right-hand side function is more complex. We note that the comparison is unfair as the computations without event detection do not give a reasonably accurate answer. Furthermore, the latter computations use more

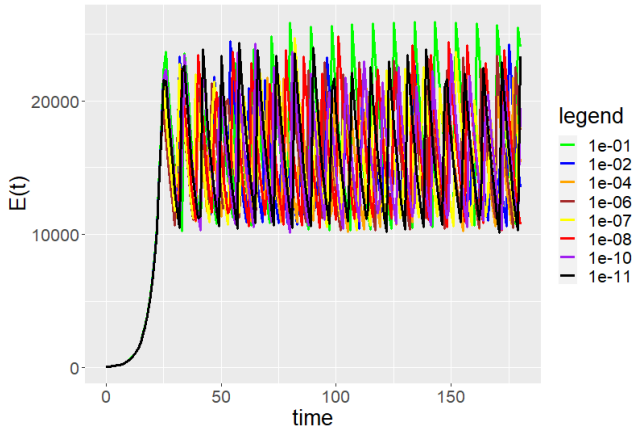

Figure 52: State-dependent discontinuity model tolerance study on the R version of LSODA without event detection.

Table 19: R version of LSODA applied to state-dependent discontinuity model tolerance study - number of function evaluations

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

function evaluations. This supports our conclusion that event detection is the appropriate way to solve state-dependent discontinuity problems when the discontinuity can be characterized in terms of an event.

State-dependent discontinuity model LSODA tolerance study in Python In this section, we use the Python version of LSODA at multiple tolerances to see how it performs. We recall that LSODA without event detection, even at very sharp tolerances, in Python was still not giving accurate results but we will see how the solutions change as the tolerance is sharpened. We will also show that coarse tolerances can be used with the computation that uses event detection.

Again Figure 54 exposes that LSODA applied at different tolerances gives substantially different results. We would expect the computations at the sharper

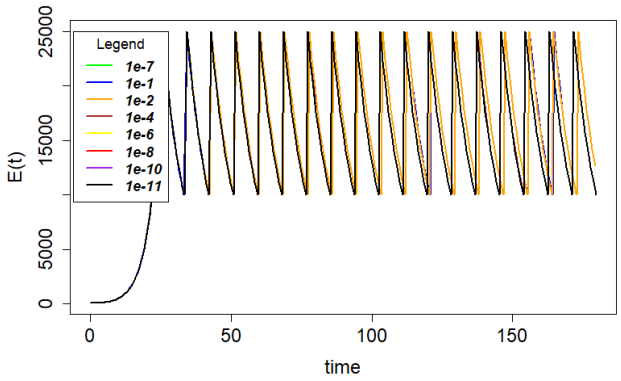

Figure 53: State-dependent discontinuity model tolerance study on the R version of LSODA with event detection.

tolerances to give quite similar results but this is not the case.

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

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

Table 20: Python version of LSODA applied to state-dependent discontinuity model tolerance study - number of function evaluations

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

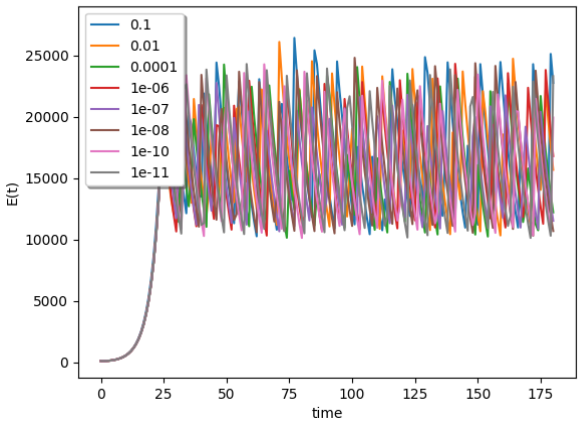

Figure 54: State-dependent discontinuity model tolerance study on the Python version of LSODA without event detection.

State-dependent discontinuity model LSODA tolerance study in Scilab We perform the same experiment in Scilab. We set the absolute and relative tolerance to the same values as in the other experiments and run the solvers. For the different tolerance values, we plot the solutions and examine how the solutions computed without event detection change as the tolerance is sharpened; we also determine how coarse a tolerance we can use with the event detection solver.

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

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

Table 21 shows the number of function evaluations that LSODA uses with and without event detection to solve the state-dependent discontinuity problem at multiple tolerances. We can see that even at coarse tolerances, using event detection allows LSODA use fewer function evaluations while giving more accurate solutions. This reinforces that event detection is the better way to solve state-dependent discontinuity problems.

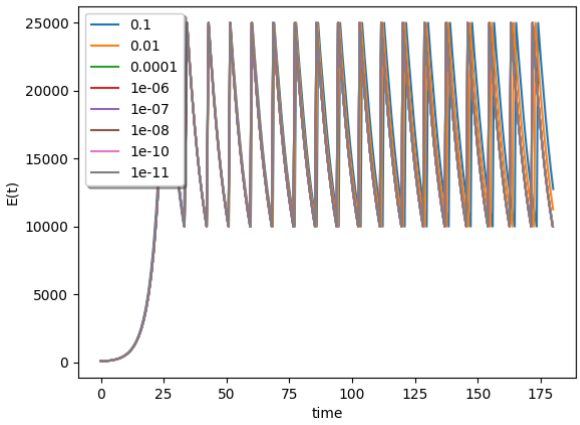

Figure 55: State-dependent discontinuity model tolerance study on the Python version of LSODA with event detection.

3.5.2 Comparing Solvers based on Runge-Kutta pairs across platforms for state-dependent discontinuity model

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

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

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

From Figure 58, we see that DOPRI5 applied with different tolerances gives significantly different solutions. We then report the efficiency data for this case in Table 22. Table 22 shows the number of function evaluations the 'ode45' solver uses. As it does not have event detection unlike the equivalent solvers in Python and Matlab, we cannot compare how the number of function evaluations differs with and without event detection. However, looking at the efficiency data of the Runge-Kutta pairs in the other environment and with R's LSODA solver,

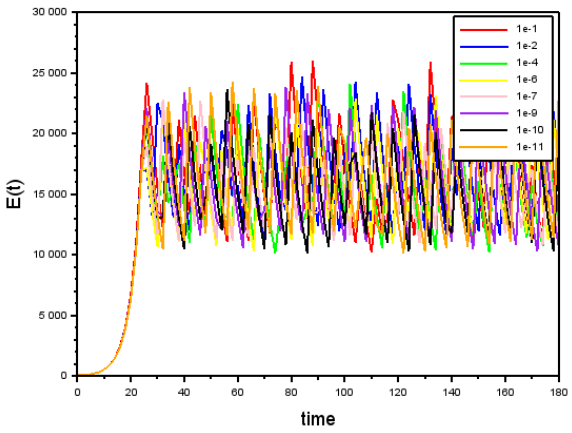

Figure 56: State-dependent discontinuity model tolerance study on the Scilab version of LSODA without event detection.

we can argue that it too will use fewer function evaluations with event detection than without.

Tolerance study on state-dependent discontinuity model using the Python version of DOPRI5 We perform the same experiment in Python. The Python version of DOPRI5 does have an event detection capability. The absolute and relative tolerances are set to a range of values and the solver is run both with and without event detection. We report on how the solver performs as the tolerance is sharpened in the case without event detection. Since the Python version of DOPRI5 has event detection, we will see how coarse the tolerance can be set while still giving us a reasonably accurate solution. We will use results from the Runge-Kutta pair in Python and in Matlab, that both have event detection, to suggest what we can expect the results from the Scilab 'rkf' and the R 'ode45' solvers, which do not have event detection, to be. We note that the solver crashes if we ask for a tolerance of 0.1.

In Figure 59, corresponding to the case with no event detection, we can see that even at sharp tolerances, the solver is not able to compute a reasonably accurate solution. In contrast, in Figure 60, which corresponds to the case where we use event detection, the solver can use very coarse tolerances and still obtain a reasonably accurate solution. We can see that a tolerance of 10^{-4} is sharp enough to solve the given problem accurately; the blurring that occurs is due to the coarser tolerances. We present the efficiency data in Table 23 to show that the solver with event detection is also far more efficient.

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

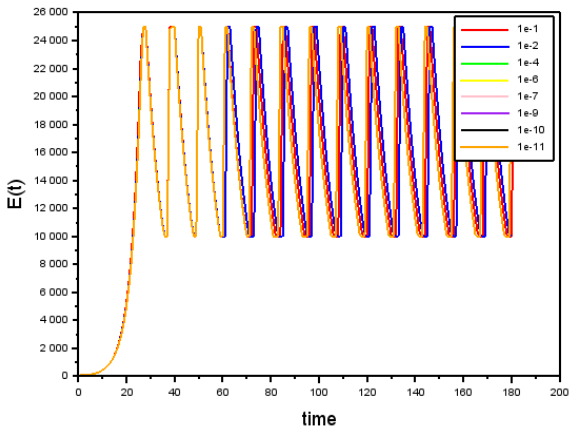

Figure 57: State-dependent discontinuity model tolerance study on the Scilab version of LSODA with event detection.

Tolerance study on state-dependent discontinuity model using the Scilab version of RKF45 Scilab uses RKF45 which is a different Runge-Kutta pair from what is used in DOPRI5 but the pairs have the same order. It does not have event detection but we can still perform the experiment on the solver without event detection. We pick several values for the absolute and relative tolerances and run the solver. In so doing we see how the solver performs as the tolerance is sharpened.

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

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

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

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

Table 25, although being an unfair comparison since the solver without event detection did not give accurate solutions, shows that solving the problem with-

Table 21: Scilab version of LSODA applied to state-dependent discontinuity model tolerance study - number of function evaluations

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

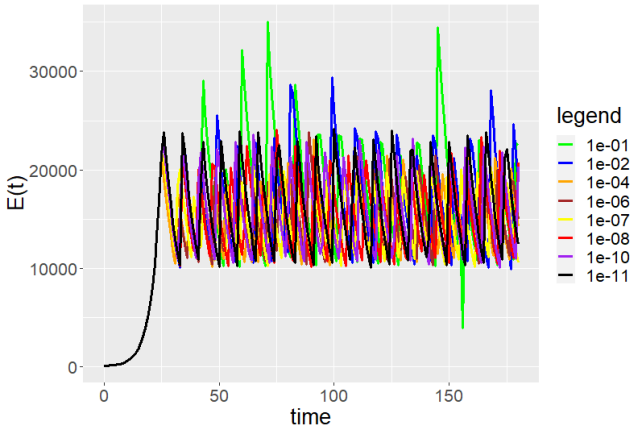

Figure 58: State-dependent discontinuity model tolerance study on the R version of DOPRI5 without event detection.

out event detection is also less efficient. At the tolerance of 0.1, the smaller number of function evaluations for the solver without event detection is not relevant since the solution at a tolerance of 0.1 is very inaccurate. At all the other tolerances, the solver with event detection is both more accurate and more efficient, usually using less than half the number of function evaluations that the solver uses when event detection is not employed.

Table 22: R version of DOPRI5 applied to state-dependent discontinuity model tolerance study - number of function evaluations

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

Table 23: The Python version of DOPRI5 applied to state-dependent discontinuity model tolerance study - number of function evaluations

tolerance	no event detection	with event detection
0.01	1400	664
0.0001	8462	806
1e-06	6248	1232
1e-07	6848	1754
1e-08	7082	2354
1e-10	10262	5066
1e-11	13058	7688

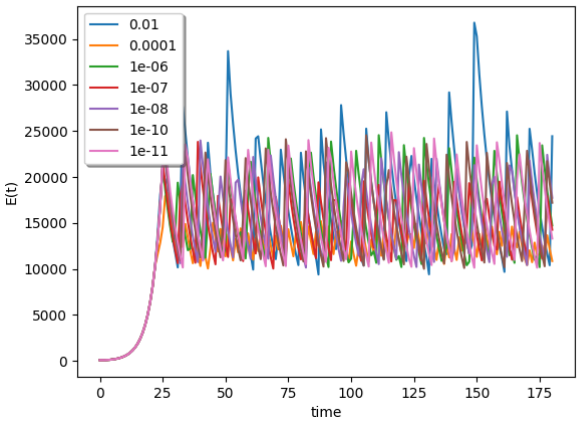

Figure 59: State-dependent discontinuity model tolerance study on the Python version of DOPRI5 without event detection.

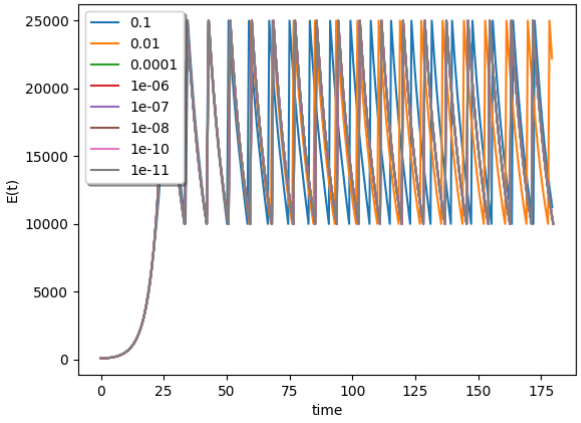

Figure 60: State-dependent discontinuity model tolerance study on the Python version of DOPRI5 with event detection.

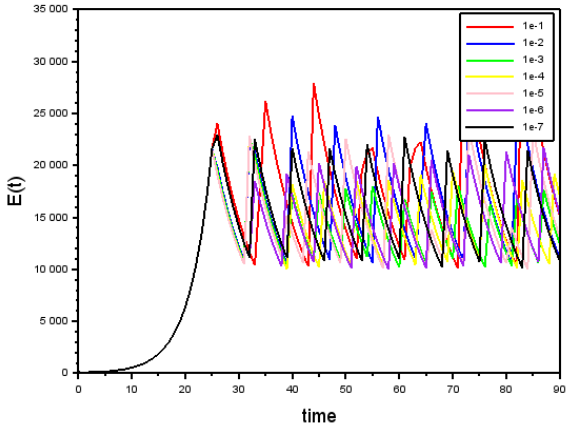

Figure 61: State-dependent discontinuity model tolerance study on the Scilab version of RKF45 without event detection.

Table 24: Scilab version of RKF45 applied to state-dependent discontinuity model tolerance study - number of function evaluations

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

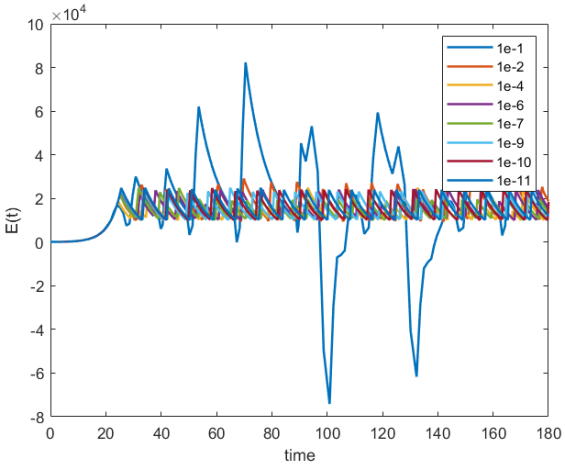

Figure 62: State-dependent discontinuity model tolerance study on the Matlab version of DOPRI5 without event detection.

Table 25: Matlab version of DOPRI5 applied to state-dependent discontinuity model tolerance study - number of function evaluations

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

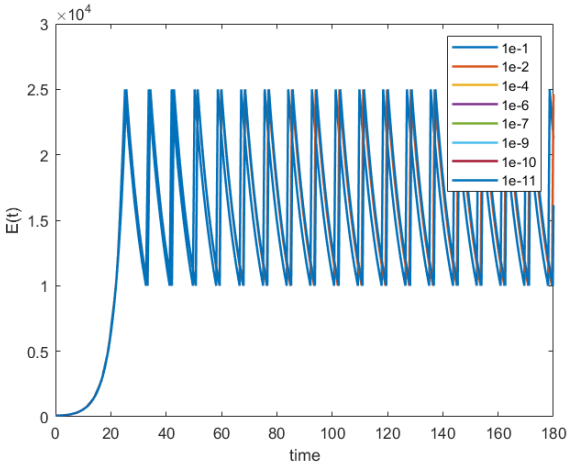

Figure 63: State-dependent discontinuity model tolerance study on the Matlab version of DOPRI5 with event detection.

4 Using continously changing parameters instead of a sharply changing parameter

4.1 Introduction

Instead of using an if-statement to change the value of an ODE model's parameter, some models may use a function that mimicks the changes in parameter. The idea is to use a function such as the exponential, sigmoid, inverse sigmoid, tanh and others to change the value of parameters sharply when required. In the case of the Covid-19 ODE model, an idea to model the change in the β parameter would be to use the inverse sigmoid scaled between 0.9 and 0.005 instead of a simple if-statement.

In this section, we discuss using such sharply changing functions for the model parameters and show how they introduce thrashing, inaccuracy issues and some efficiency issues. We tackle both modeling a time-dependent discontinuity and a state-dependent discontinuity model with these rapidly changing functions and report on ways to improve the accuracy.

4.1.1 Functions Used to model parameters

The inverse sigmoid function An inverse sigmoid function defined as follows:

$$\beta(t) = \frac{0.895e^{-a*(t-t_c)}}{1+e^{-a*(t-t_c)}} + 0.005 \tag{6}$$

is a function that decreases from 0.9 to 0.005 at t_c with a steepness of change of a. This function can be used to model the if-statement when adding Covid-19 measures at time t_c and the steepness of change, a, models how quickly the population adapts to the introduction of these measures.

By varying the value of a, we can change the steepness of discontinuity. Figure 64 shows how different values of a smoothed the discontinuity. We can see that a smaller value of a makes the change of the parameter *beta* from 0.9 to 0.005 slower.

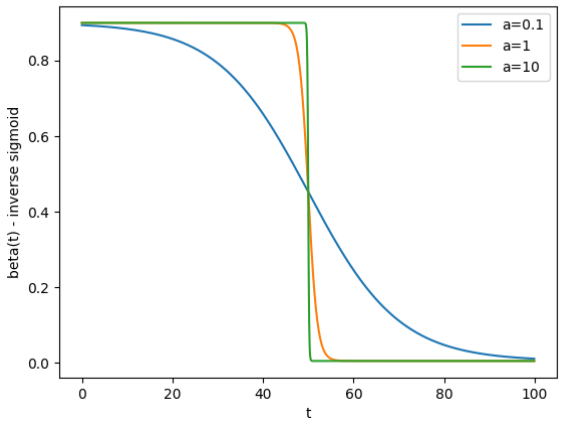

Figure 64: Inverse sigmoid with discontinuity at $t_c = 50$ with steepness of change, a, at 0.1, 1, 10.

The sigmoid function A sigmoid function defined as follows:

$$\beta(t) = \frac{0.895}{(1 + e^{-a(t-t_c)}))} + 0.005 \tag{7}$$

is a function that increases from 0.005 to 0.9 at t_c with a steepness of change of a. This function can be used to model the if-statement when removing Covid-19 measures at time t_c and the steepness of change, a, models how quickly the population adapts to removal of these measures.

By varying the value of a, we can change the steepness of discontinuity. Figure 65 shows how different values of a smoothed the discontinuity. We can see that a smaller value of a makes the change of the parameter *beta* from 0.005 to 0.9 slower.

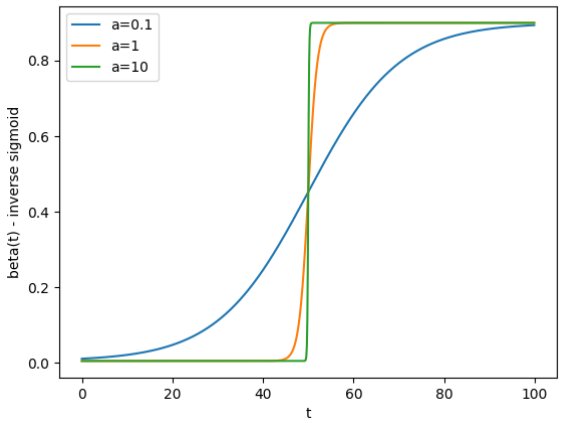

Figure 65: Sigmoid with discontinuity at $t_c = 50$ with steepness of change, a, at 0.1, 1, 10.

4.1.2 Existence of thrashing

In this section, we show that depending on the steepness of change, thrashing also occurs in the ODE model.

Figure 66 and 67 shows the cumulative number of function evaluations at each time when solving the Covid-19 time-dependent discontinuity model using the inverse sigmoid function to model the change in the parameter β at $t = 27$ instead of using an if-statement using the 'LSODA' and the 'DOP853' methods of *solve_ivp* in Python. We can see that for small values of the steepness of change, a, there is no spike in the number of function evaluations at $t = 27$ but as the value of a is increased, a progressively sharper spike can be seen. This indicates some level of thrashing as repeated step-size reductions to cross the discontinuity was required.

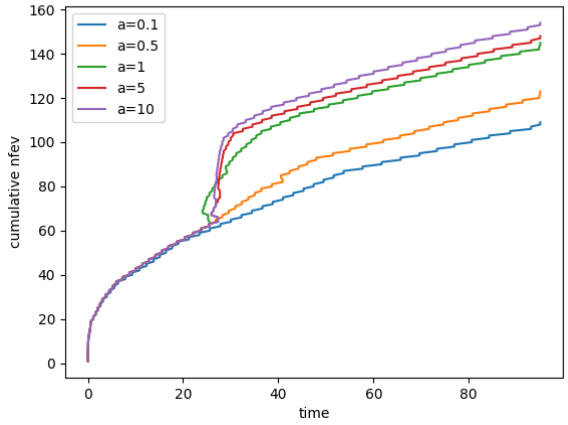

Figure 66: Thrashing using the inverse sigmoid to model the change from 0.9 to 0.005 at t=27 with a = 0.1, 0.5, 1, 5, 10 when solving with the LSODA method.

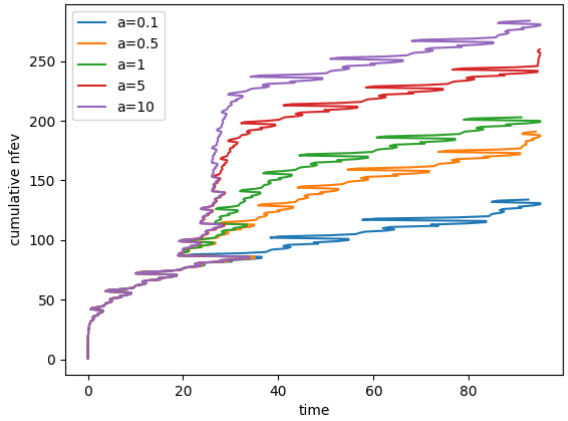

Figure 67: Thrashing using the inverse sigmoid to model the change from 0.9 to 0.005 at t=27 with a = 0.1, 0.5, 1, 5, 10 when solving with the DOP853 method.

In the remainder of this section, we attempt to solve the time-dependent and state-dependent discontinuity problems with steepness of change, a, of 0.1, 1 and 10 to show how using cold starts in the time-dependent discontinuity problem and using event detection in the state-dependent discontinuity problem improves accuracy.

4.2 Time-dependent discontinuity with exponential change

4.2.1 Naive solution of the time-dependent discontinuity with exponential change model

A naive implementation of the model involves using the inverse sigmoid with $t_c = 27$ to model the parameter β inside the right-hand side function, $f(t, y)$, to implement the change in β as measures are implemented and solving the problem in a single call from $t = 0$ to $t = 95$. Also, to stay true to a naive treatment, we will use the default tolerances in this section.

In pseudo code, this looks like:

```
function model_with_if(t, y)
    // ...
    beta = inverse_sigmoid(t, t_c=27)
    // ...
    // return (dSdt, dEdt, dIdt, dRdt)
```

Figures 68, 69 and 70 shows the naive solution to the time-dependent discontinuity problem using a steepness of change of 0.1, 1 and 10.

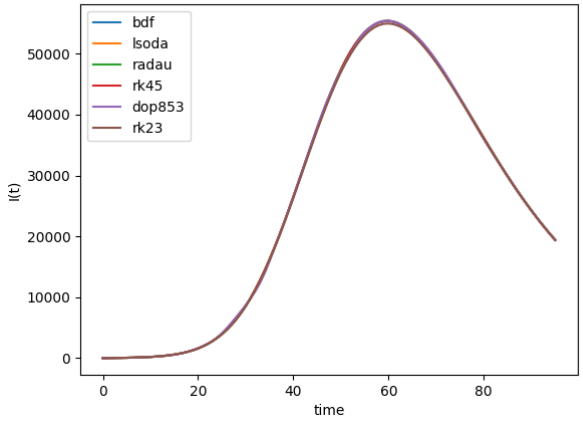

Figure 68: Naive solution to the time-dependent discontinuity problem with a steepness of change of 0.1.

Figure 68 shows the naive solution with a steepness of change of 0.1. We can see that the solvers are aligned with each others.

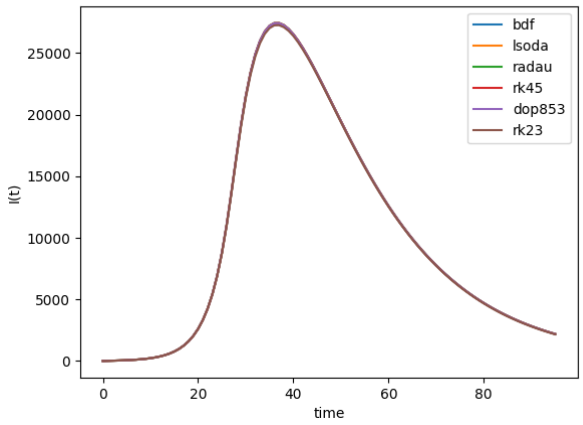

Figure 69: Naive solution to the time-dependent discontinuity problem with a steepness of change of 1.

Figure 69 shows the naive solution with a steepness of change of 1. We can see that the solvers are aligned with each others but there is some slight blurring.

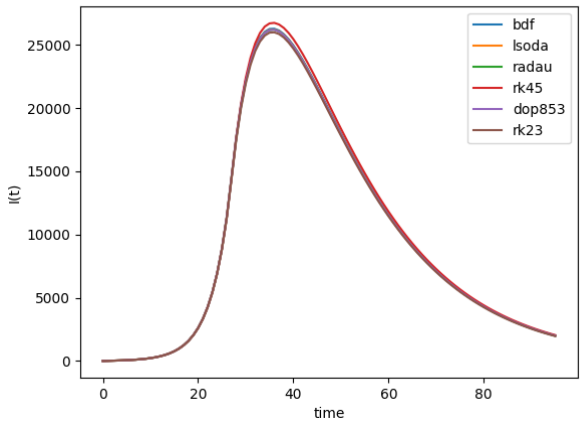

Figure 70: Naive solution to the time-dependent discontinuity problem with a steepness of change of 10.

Figure 70 shows the naive solution with a steepness of change of 10. We can see that the solvers are no longer all aligned with each others.

This behaviour is as expected, as the steepness of change grows larger, the problem becomes more discontinuous, there is more thrashing and thus there is

a drop in accuracy.

In the next section, we present a solution using cold starts and show how it improves the accuracy.

4.2.2 Solving the time-dependent discontinuity with exponential change model using a cold start

As we have seen in the if-statement case, a better way to solve the time-dependent discontinuity problem is to make use of cold starts. This means that we integrate up to the time at which the discontinuity arises and then after the discontinuity we continue the integration with a *separate* call to the solver.

A cold start means that we restart the solver with method parameters set so that the solver starts the computation with no values from the previous computation. It will also involve using a small initial step size and for methods of varying order like the 'BDF' methods, they will restart with the default order which is order 1.

To solve the time dependent discontinuity problem, we will integrate from time 0 to the time that measures are implemented, t=27, with one call to the solver and then use the solution values at t=27 as the initial values to make another call that will integrate (restarting with a cold start) from t=27 to t_f. 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)
```

Figures 71, 72 and 73 shows the discontinuity handling solutions to the time-dependent discontinuity problem using a steepness of change of 0.1, 1 and 10.

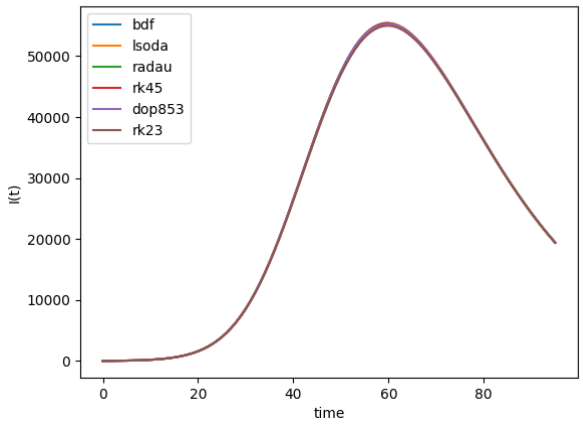

Figure 71: Solutions to the time-dependent discontinuity model using the inverse sigmoid with a steepness of change of 0.1 using solvers from Python and a cold start at t=27.

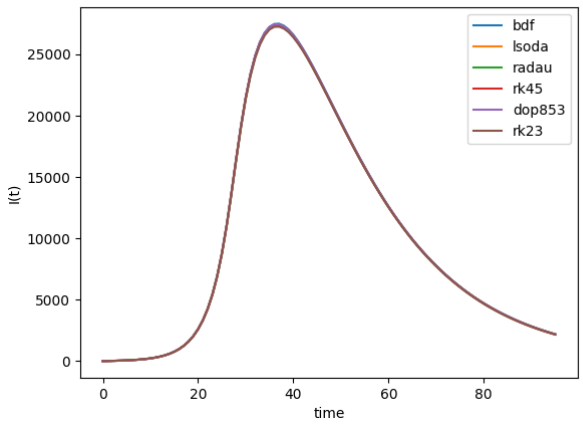

Figure 72: Solutions to the time-dependent discontinuity model using the inverse sigmoid with a steepness of change of 1 using solvers from Python and a cold start at t=27.

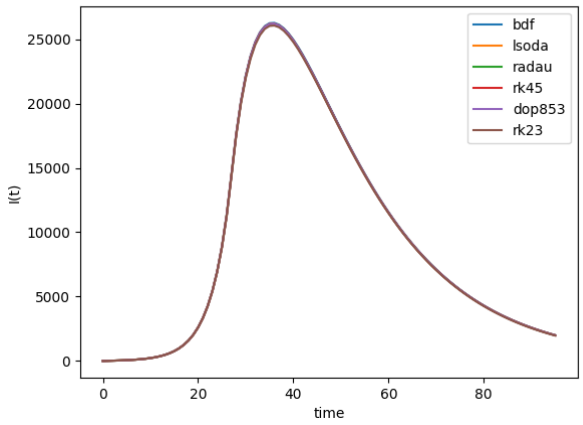

Figure 73: Solutions to the time-dependent discontinuity model using the inverse sigmoid with a steepness of change of 10 using solvers from Python and a cold start at t=27.

Using the cold state at $t = 27$ allowed the solvers to produce solutions that are aligned at all three different steepness of change. Thus the cold start improved accuracy.

However the efficiency of the solvers is not improved by much.

Table 26: Python efficiency data for the time-dependent discontinuity problem with a steepness of change of 0.1 - number of function evaluations

method	no discontinuity handling	with discontinuity handling
lsoda	109	116
rk45	116	130
bdf	127	140
radau	182	201
dop853	134	181
rk23	113	121

Table 27: Python efficiency data for the time-dependent discontinuity problem with a steepness of change of 1 - number of function evaluations

method	no discontinuity handling	with discontinuity handling
lsoda	145	150
rk45	122	136
bdf	166	173
radau	241	247
dop853	203	211
rk23	131	139

Table 28: Python efficiency data for the time-dependent discontinuity problem with a steepness of change of 10 - number of function evaluations

method	no discontinuity handling	with discontinuity handling
lsoda	154	164
rk45	140	148
bdf	186	174
radau	288	287
dop853	284	262
rk23	134	142

4.2.3 LSODA time-dependent discontinuity with exponential change problem tolerance study

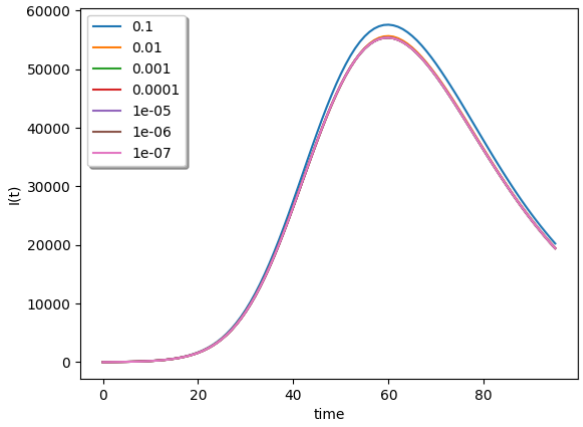

Figure 74: Time-dependent discontinuity model tolerance study on the Python version of LSODA without a cold start with a steepness of change of 0.1.

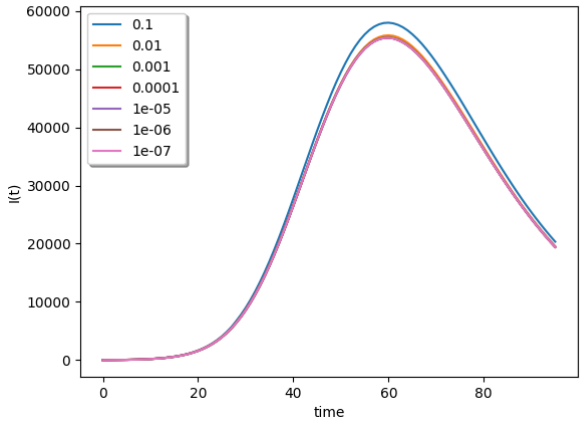

Figure 75: Time-dependent discontinuity model tolerance study on the Python version of LSODA with a cold start with a steepness of change of 0.1.

Table 29: Python LSODA time-dependent discontinuity model with exponential change with a steepness of change of 0.1 tolerance study - number of function evaluations

tolerance	no discontinuity handling	with discontinuity handling
0.1	78.0	87.0
0.01	81.0	93.0
0.001	99.0	108.0
0.0001	127.0	136.0
1e-05	161.0	176.0
1e-06	199.0	228.0
1e-07	235.0	266.0

steepness of change of 0.1

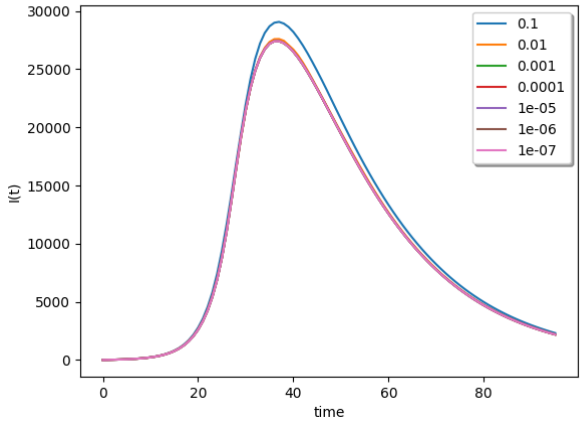

Figure 76: Time-dependent discontinuity model tolerance study on the Python version of LSODA without a cold start with a steepness of change of 1.

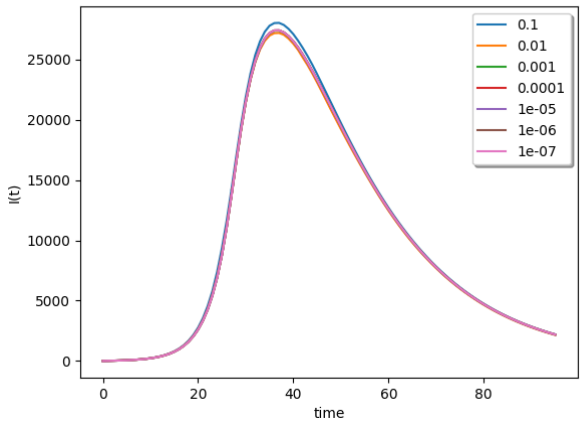

Figure 77: Time-dependent discontinuity model tolerance study on the Python version of LSODA with a cold start with a steepness of change of 1.

Table 30: Python LSODA time-dependent discontinuity model with exponential change with a steepness of change of 1 tolerance study - number of function evaluations

tolerance	no discontinuity handling	with discontinuity handling
0.1	78.0	88.0
0.01	89.0	99.0
0.001	135.0	140.0
0.0001	173.0	182.0
1e-05	215.0	234.0
1e-06	261.0	286.0
1e-07	333.0	370.0

steepness of change of 1

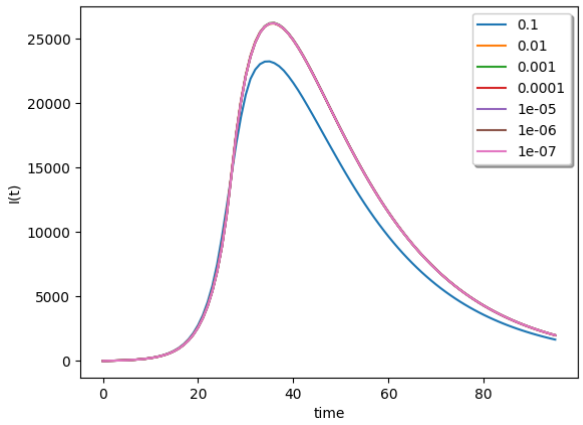

Figure 78: Time-dependent discontinuity model tolerance study on the Python version of LSODA without a cold start with a steepness of change of 10.

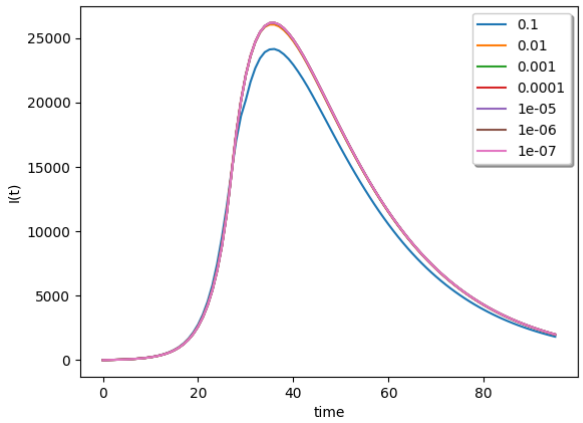

Figure 79: Time-dependent discontinuity model tolerance study on the Python version of LSODA with a cold start with a steepness of change of 10.

Table 31: Python LSODA time-dependent discontinuity model with exponential change with a steepness of change of 1 tolerance study - number of function evaluations

tolerance	no discontinuity handling	with discontinuity handling
0.1	77.0	86.0
0.01	98.0	99.0
0.001	140.0	151.0
0.0001	191.0	205.0
1e-05	264.0	263.0
1e-06	320.0	335.0
1e-07	386.0	415.0

steepness of change of 10

4.2.4 RK45 time-dependent discontinuity problem tolerance study

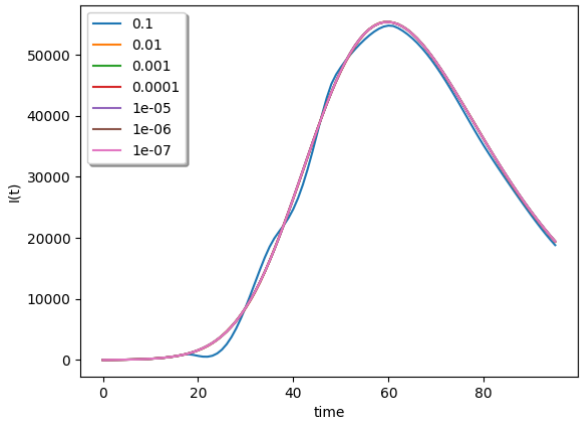

Figure 80: Time-dependent discontinuity model tolerance study on the Python version of RK45 without a cold start with a steepness of change of 0.1.

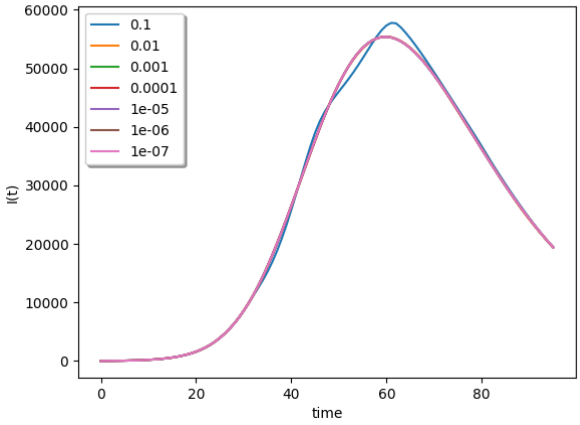

Figure 81: Time-dependent discontinuity model tolerance study on the Python version of RK45 with a cold start with a steepness of change of 0.1.

Table 32: Python RK45 time-dependent discontinuity model with exponential change with a steepness of change of 0.1 tolerance study - number of function evaluations

tolerance	no discontinuity handling	with discontinuity handling
0.1	62.0	70.0
0.01	80.0	94.0
0.001	110.0	130.0
0.0001	158.0	172.0
1e-05	230.0	244.0
1e-06	350.0	370.0
1e-07	536.0	550.0

steepness of change of 0.1

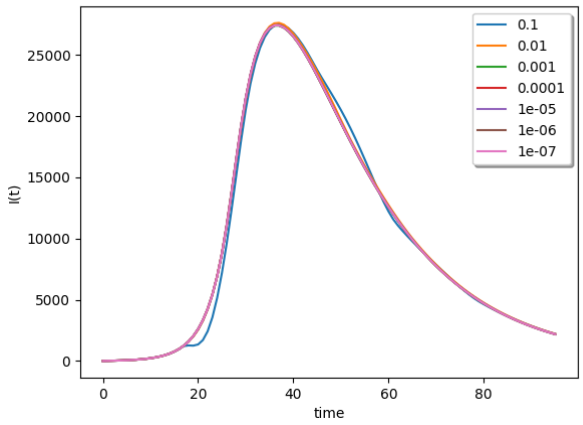

Figure 82: Time-dependent discontinuity model tolerance study on the Python version of RK45 without a cold start with a steepness of change of 1.

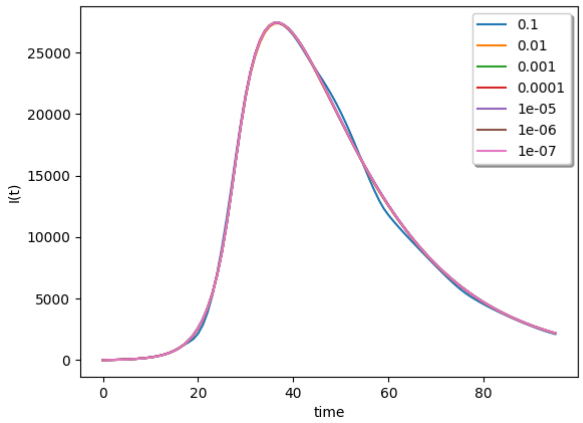

Figure 83: Time-dependent discontinuity model tolerance study on the Python version of RK45 with a cold start with a steepness of change of 1.

Table 33: Python RK45 time-dependent discontinuity model with exponential change with a steepness of change of 1 tolerance study - number of function evaluations

tolerance	no discontinuity handling	with discontinuity handling
0.1	68.0	76.0
0.01	86.0	100.0
0.001	116.0	130.0
0.0001	164.0	178.0
1e-05	248.0	262.0
1e-06	374.0	382.0
1e-07	572.0	586.0

steepness of change of 1

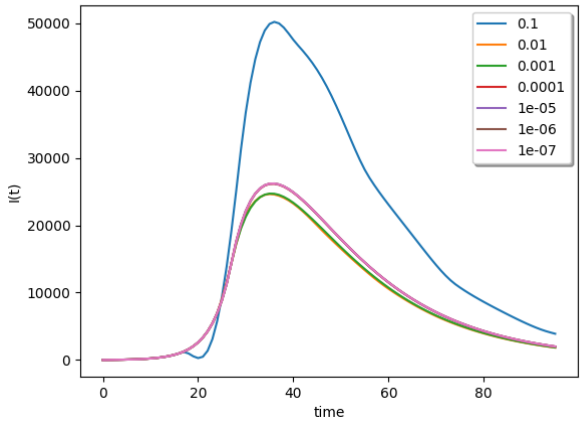

Figure 84: Time-dependent discontinuity model tolerance study on the Python version of RK45 without a cold start with a steepness of change of 10.

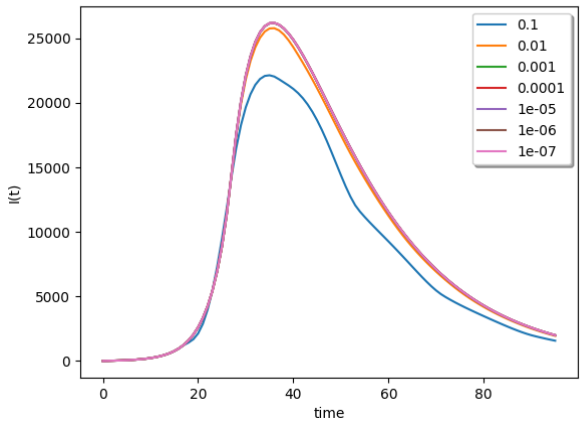

Figure 85: Time-dependent discontinuity model tolerance study on the Python version of RK45 with a cold start with a steepness of change of 10.

Table 34: Python RK45 time-dependent discontinuity model with exponential change with a steepness of change of 1 tolerance study - number of function evaluations

tolerance	no discontinuity handling	with discontinuity handling
0.1	68.0	76.0
0.01	86.0	106.0
0.001	116.0	124.0
0.0001	182.0	190.0
1e-05	284.0	292.0
1e-06	410.0	424.0
1e-07	644.0	652.0

steepness of change of 10

4.3 State-dependent discontinuity with exponential change

In this section, we consider an extension of the state-dependent discontinuity problem where we use the sigmoid and inverse sigmoid function to change the parameter β instead of if-statements. We again attempt a long term forecast where measures are introduced and relaxed based on E(t), the number of exposed individuals at time, t.

As in Section 2, changes in the modelling parameter β introduce discontinuities in the function $f(t, y(t))$ and thus the error control solvers will "thrash" when trying to solve the problem (as described in Section 1.5).

When there are no measures and E(t) crosses 25000, we assume that measures are introduced which will reduce the value of the parameter β from 0.9 to 0.005. When there are no measures and E(t) crosses 10000, we assume that measures are relaxed which increases the value of the parameter β from 0.005 back to 0.9.

We start with a simple treatment of the problem with 'if' statements employed inside the function that defines the right-hand side of the ODE system which choses between the sigmoid and the inverse sigmoid centered at the time of the last change based on whether measures are introduced or not and show how this form of the problem cannot be solved with reasonable accuracy, by any of the solvers, even at sharp tolerances. Finally, we will introduce an approach to efficiently and accurately solve the problem using an approach involving the use of what is known as event detection to handle the discontinuities.

4.3.1 Naive solution of the state-dependent discontinuity with exponential change model

A simple treatment of this problem is to use global variables for tracking when measures are implemented and relaxed and to toggle these global variables as we reach the required thresholds. We use this approach because we need to know if the number of exposed people is going up or down to know whether we need to check for the maximum or the minimum threshold. We then have an 'if' statement that will choose between the sigmoid and inverse sigmoid function centered at the time of the last change, also a global variable, for the parameter β based on whether measures are being implemented or not. The pseudo-code for this algorithm is as follows:

```
measures_implemented = False
direction = "up"
time_last_changed = 0

function model_with_if(t, y):
    // ...
    global measures_implemented, direction, time_last_changed
    if (direction == "up"):
        if (E > 25000):
            measures_implemented = True
            direction = "down"
            time_last_changed = t
    else:
        if (E < 10000):
            measures_implemented = False
            direction = "up"
            time_last_changed = t

    if measures_implemented:
        beta = inverse_sigmoid(t, t_c=time_last_changed)
    else:
        beta = sigmoid(t, t_c=time_last_changed)
    // ...
    return (dSdt, dEdt, dIdt, dRdt)
```

Figures 86, 88 and 90 shows the naive solution to the state-dependent discontinuity problem using a steepness of change of 0.1, 1 and 10.

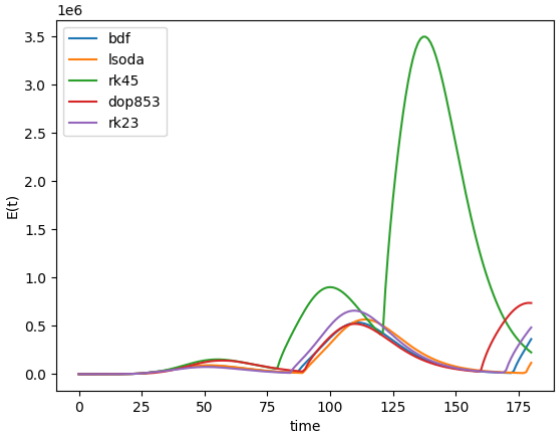

Figure 86: Naive solution to the state-dependent discontinuity problem with a steepness of change of 0.1.

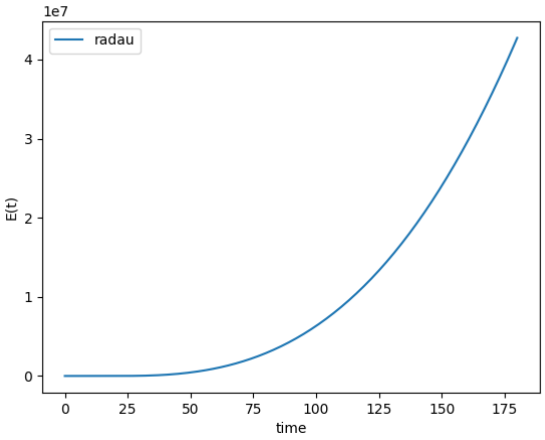

Figure 87: Naive solution computed by 'Radau' to the state-dependent discontinuity problem with a steepness of change of 0.1.

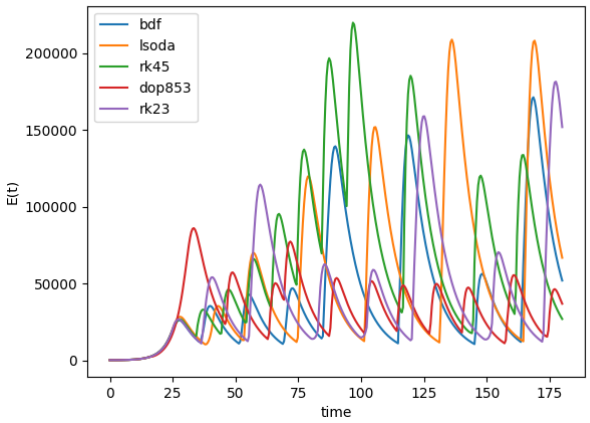

Figure 88: Naive solution to the state-dependent discontinuity problem with a steepness of change of 1.

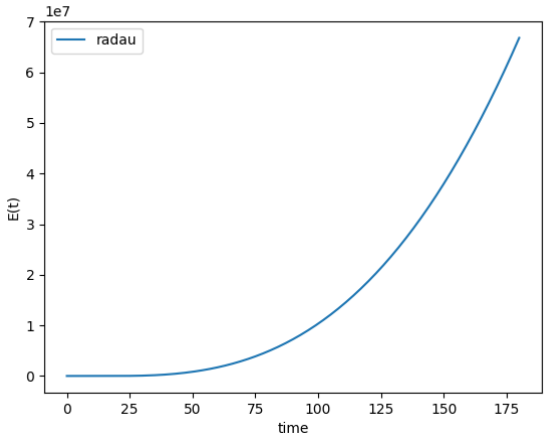

Figure 89: Naive solution computed by 'Radau' to the state-dependent discontinuity problem with a steepness of change of 1.

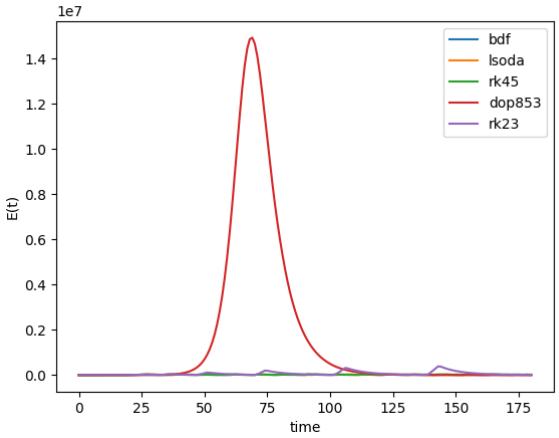

Figure 90: Naive solution to the state-dependent discontinuity problem with a steepness of change of 10.

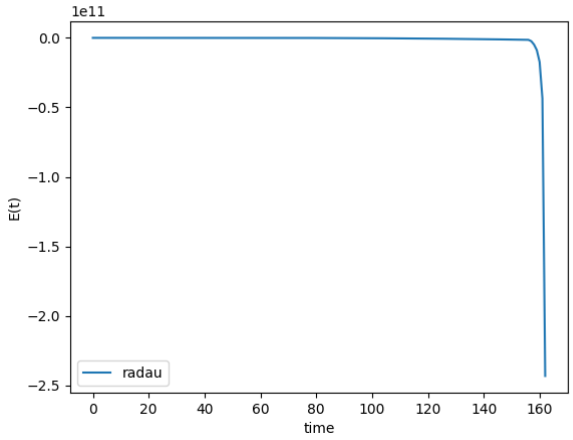

Figure 91: Naive solution computed by 'Radau' to the state-dependent discontinuity problem with a steepness of change of 10.

4.3.2 Naive sharp tolerance solution of the state-dependent discontinuity with exponential change model

Figures 92, 93 and 94 shows the naive solution to the state-dependent discontinuity problem using a steepness of change of 0.1, 1 and 10 at an absolute tolerance of 10^{-12} and a relative tolerance of 10^{-12}.

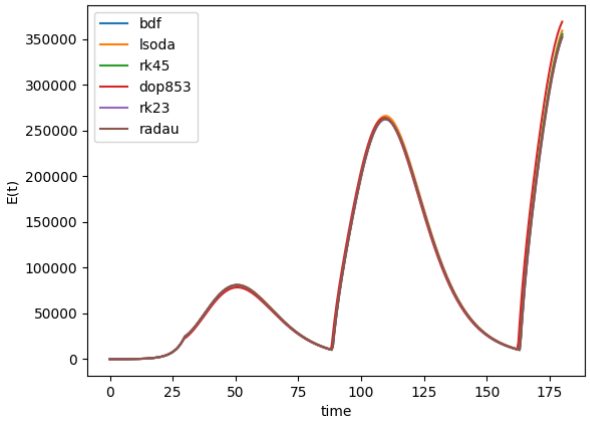

Figure 92: Naive sharp tolerance solution to the state-dependent discontinuity problem with a steepness of change of 0.1.

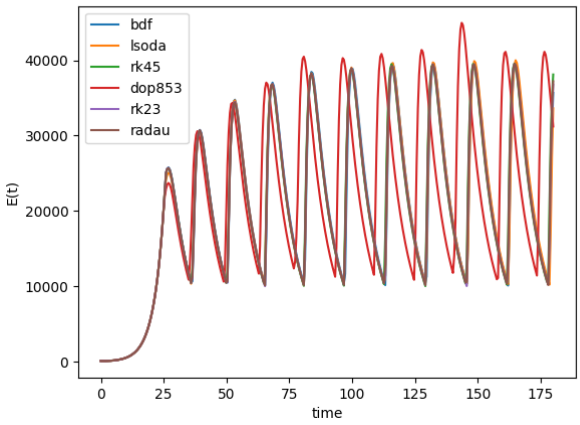

Figure 93: Naive sharp tolerance solution to the state-dependent discontinuity problem with a steepness of change of 1.

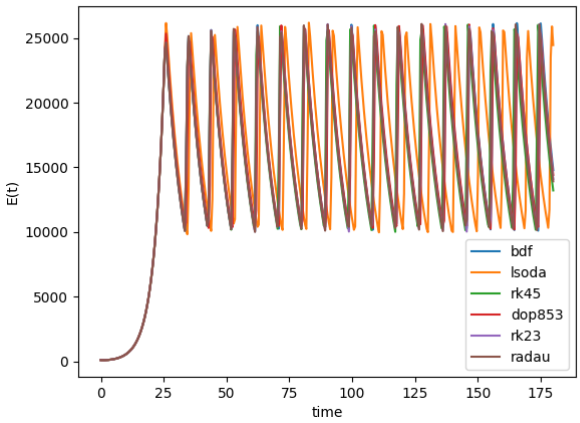

Figure 94: Naive sharp tolerance solution to the state-dependent discontinuity problem with a steepness of change of 10.

4.3.3 Solving the state-dependent discontinuity with exponential change model using event detection

We can see that with this version of the state-dependent discontinuity problem, using a sharp tolerance allows us to get solutions that are aligned with each others. We then use event detection to try to see if we can get more accurate solutions.

Again we use the idea of defining the thresholds as events. When an event is detected, that is a threshold is crossed, we cold start the solver with a new right hand side function using either the sigmoid or inverse sigmoid function centered at the time of the last event for the value of the β parameter and a new root function.

For our specific problem, event detection is used as follows. We start by solving the problem with β using sigmoid function growing from 0.005 to 0.9 centered at $t - 0$ and with a root function that detects when E(t) is equal to 25000. Once, using the event detection capability of the solver, we reach the time at which $E(t) = 25000$, we do a cold start. We evaluate the solution computed by the solver at the time of the event and use that solution as the initial value for our next call to the solver. This next call will have β using the inverse sigmoid function decreasing from 0.9 to 0.005 centered at the time of the $E(t) = 25000$ event just recorded and a root function that detects a root when $E(t) = 10000$. We again integrate up to that new threshold and cold start when we reach it. The new integration will have β using the sigmoid function growing from 0.005 to 0.9 centered at the time of the $E(t) = 10000$ event just recorded and the root function will look for $E(t) = 25000$ as the event. This is repeated until we reach the desired end time. The pseudo-code is as follows:

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

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

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

Figures 95, 96 and 97 shows the solution using event detection to the state-dependent discontinuity problem using a steepness of change of 0.1, 1 and 10.

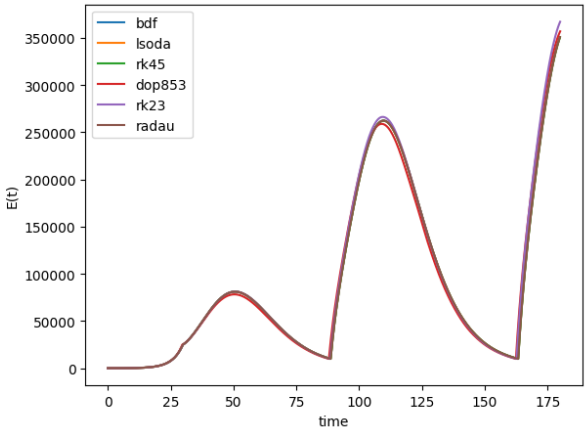

Figure 95: Solution using event detection to the state-dependent discontinuity problem with a steepness of change of 0.1.

Table 35: Efficiency data for Python state-dependent discontinuity model with exponential change with a steepness of change of 0.1 - number of function evaluations

method	no event	no event with sharp tol.	with event detection
lsoda	296	1389	240
bdf	393	5663	297
radau	204	48529	397
rk45	272	10136	282
dop853	1094	4817	432
rk23	293	168026	237

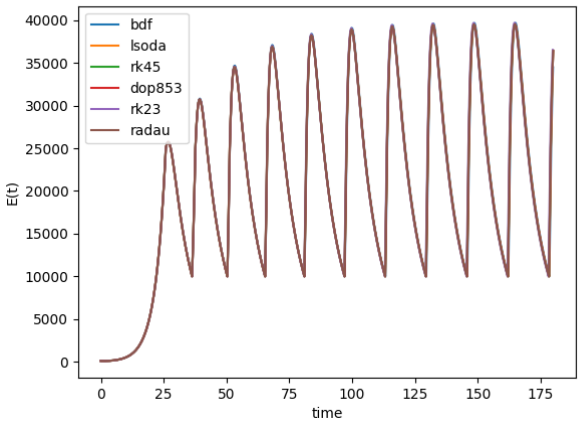

Figure 96: Solution using event detection to the state-dependent discontinuity problem with a steepness of change of 1.

Table 36: Efficiency data for Python state-dependent discontinuity model with exponential change with a steepness of change of 1 - number of function evaluations

method	no event	no event with sharp tol.	with event detection
lsoda	589	4829	548
bdf	811	14642	758
radau	211	96933	1076
rk45	566	15800	584
dop853	2648	14558	1184
rk23	653	306788	527

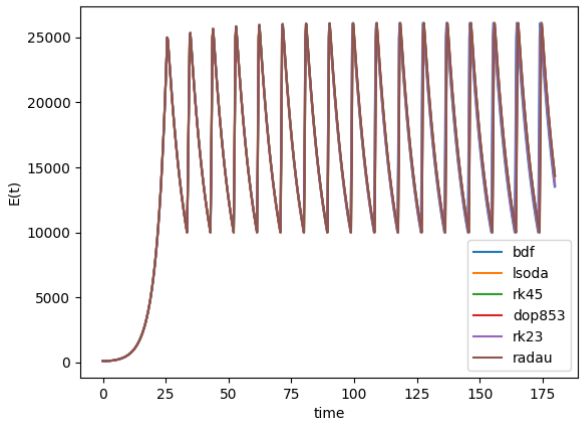

Figure 97: Solution using event detection to the state-dependent discontinuity problem with a steepness of change of 10.

Table 37: Efficiency data for Python state-dependent discontinuity model with exponential change with a steepness of change of 10 - number of function evaluations

method	no event	no event with sharp tol.	with event detection
lsoda	1503	11518	1098
bdf	1235	27795	1192
radau	1810	164687	1739
rk45	2384	25994	842
dop853	992	27092	2063
rk23	572	440066	785

4.3.4 LSODA state-dependent discontinuity problem tolerance study

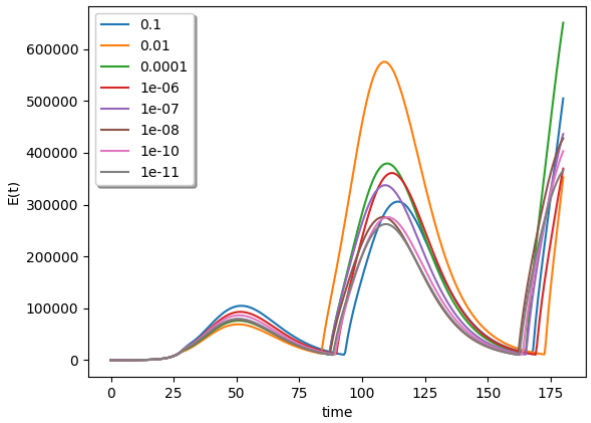

Figure 98: State-dependent discontinuity model tolerance study on the Python version of LSODA without event detection with a steepness of change of 0.1.

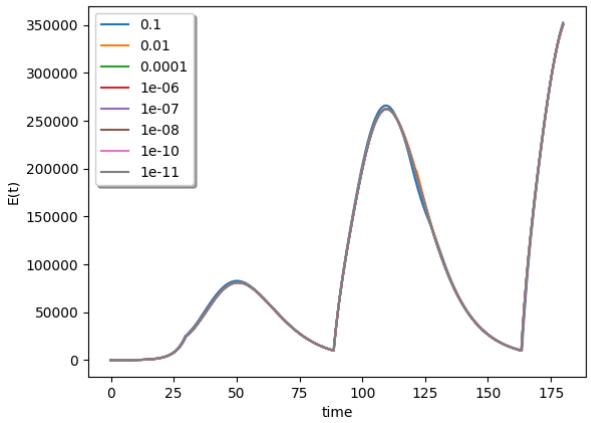

Figure 99: State-dependent discontinuity model tolerance study on the Python version of LSODA with event detection with a steepness of change of 0.1.

Table 38: Python LSODA state-dependent discontinuity model with exponential change with a steepness of change of 0.1 tolerance study - number of function evaluations

tolerance	no event detection	with event detection
0.1	200	182
0.01	246	192
0.0001	345	300
1e-06	515	478
1e-07	621	620
1e-08	782	738
1e-10	993	980
1e-11	1200	1127

steepness of change of 0.1

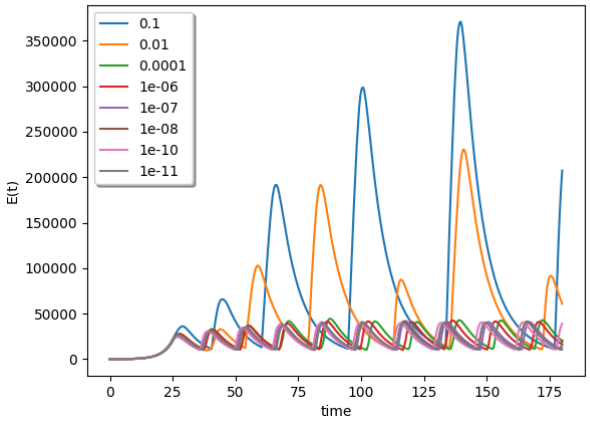

Figure 100: State-dependent discontinuity model tolerance study on the Python version of LSODA without event detection with a steepness of change of 1.

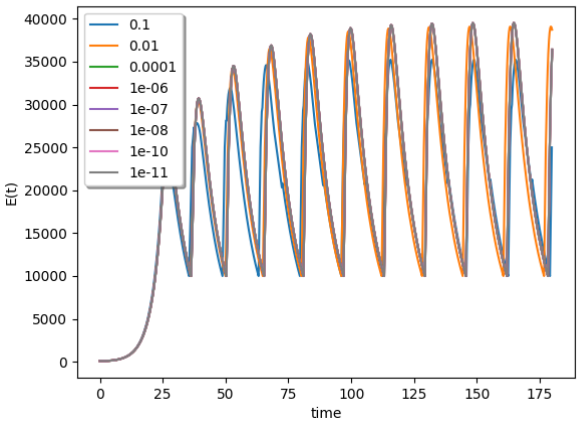

Figure 101: State-dependent discontinuity model tolerance study on the Python version of LSODA with event detection with a steepness of change of 1.

Table 39: Python LSODA state-dependent discontinuity model with exponential change with a steepness of change of 1 tolerance study - number of function evaluations

tolerance	no event detection	with event detection
0.1	339	353
0.01	437	431
0.0001	1133	786
1e-06	1793	1406
1e-07	2170	1748
1e-08	2640	2242
1e-10	3559	3271
1e-11	4059	3707

steepness of change of 1

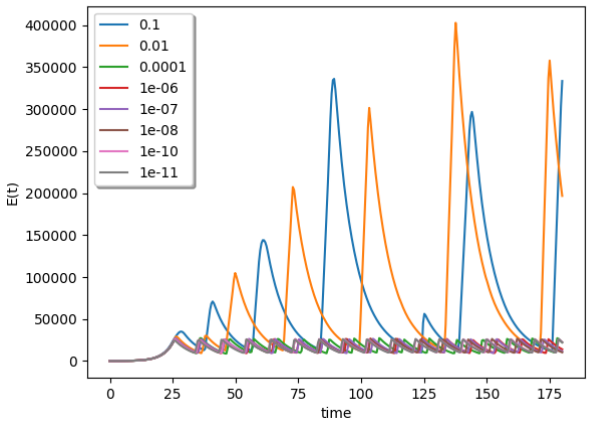

Figure 102: State-dependent discontinuity model tolerance study on the Python version of LSODA without event detection with a steepness of change of 10.

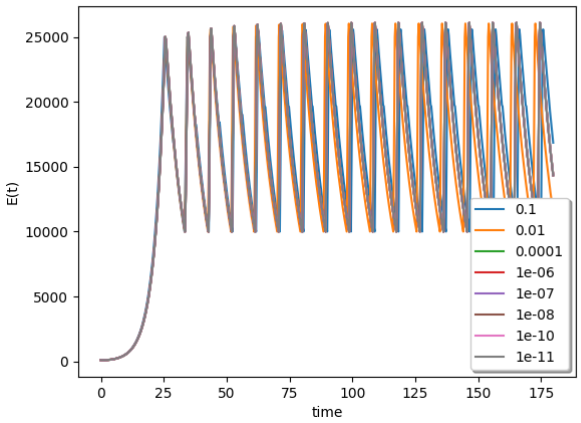

Figure 103: State-dependent discontinuity model tolerance study on the Python version of LSODA with event detection with a steepness of change of 10.

Table 40: Python LSODA state-dependent discontinuity model with exponential change with a steepness of change of 10 tolerance study - number of function evaluations

tolerance	no event detection	with event detection
0.1	365	763
0.01	504	878
0.0001	2338	1746
1e-06	4004	3106
1e-07	5224	4004
1e-08	6158	4984
1e-10	8565	7367
1e-11	10068	7885

steepness of change of 10

4.3.5 RK45 state-dependent discontinuity problem tolerance study

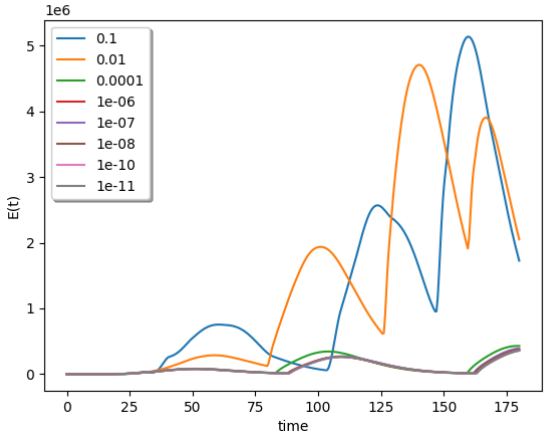

Figure 104: State-dependent discontinuity model tolerance study on the Python version of RK45 without event detection with a steepness of change of 0.1.

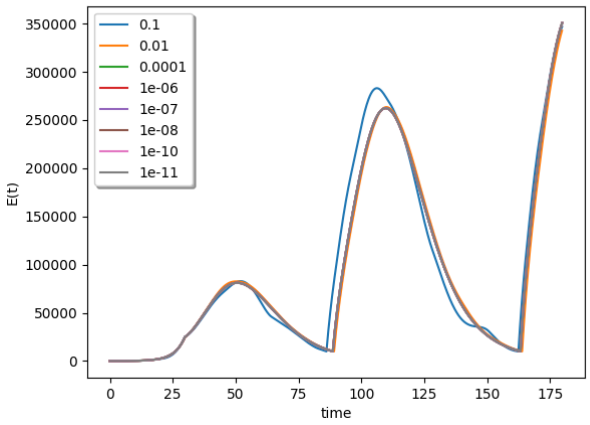

Figure 105: State-dependent discontinuity model tolerance study on the Python version of RK45 with event detection with a steepness of change of 0.1.

Table 41: Python RK45 state-dependent discontinuity model with exponential change with a steepness of change of 0.1 tolerance study - number of function evaluations

tolerance	no event detection	with event detection
0.1	164	180
0.01	224	222
0.0001	458	360
1e-06	1094	726
1e-07	1490	1062
1e-08	2066	1608
1e-10	4430	3834
1e-11	6644	6006

steepness of change of 0.1

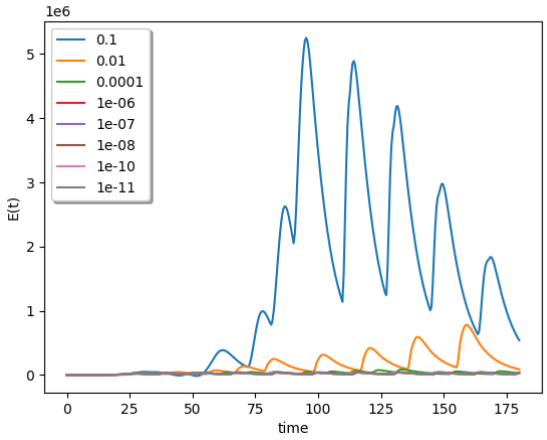

Figure 106: State-dependent discontinuity model tolerance study on the Python version of RK45 without event detection with a steepness of change of 1.

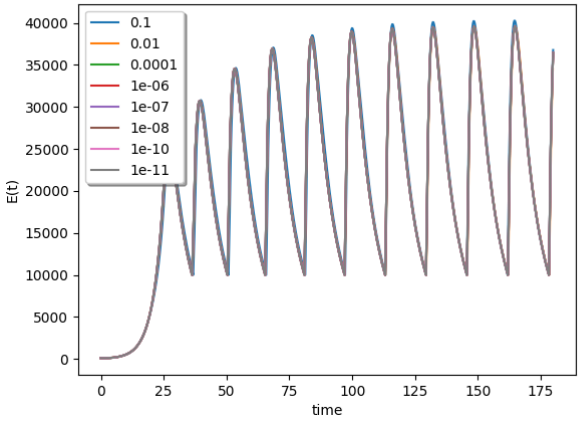

Figure 107: State-dependent discontinuity model tolerance study on the Python version of RK45 with event detection with a steepness of change of 1.

Table 42: Python RK45 state-dependent discontinuity model with exponential change with a steepness of change of 1 tolerance study - number of function evaluations

tolerance	no event detection	with event detection
0.1	320	434
0.01	416	452
0.0001	1100	776
1e-06	2738	1166
1e-07	3506	1658
1e-08	4490	2378
1e-10	7898	5306
1e-11	11042	8126

steepness of change of 1

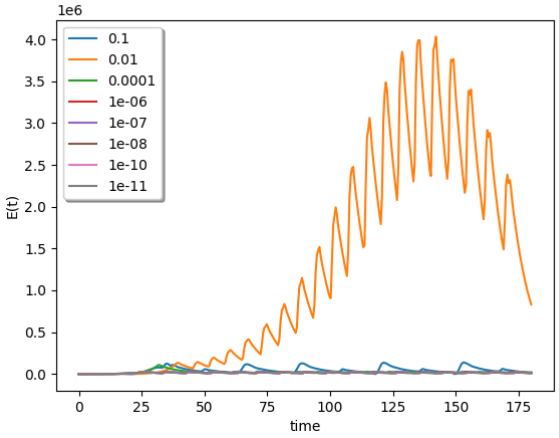

Figure 108: State-dependent discontinuity model tolerance study on the Python version of RK45 without event detection with a steepness of change of 10.

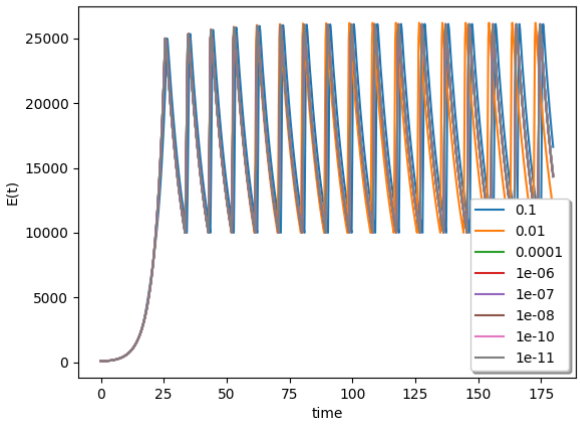

Figure 109: State-dependent discontinuity model tolerance study on the Python version of RK45 with event detection with a steepness of change of 10.

Table 43: Python RK45 state-dependent discontinuity model with exponential change with a steepness of change of 10 tolerance study - number of function evaluations

tolerance	no event detection	with event detection
0.1	410	674
0.01	776	710
0.0001	1346	1040
1e-06	3686	2024
1e-07	5402	2756
1e-08	8036	4046
1e-10	13574	8978
1e-11	18476	13400

steepness of change of 10

5 Investigation of the Radau software applied to the state-dependent discontinuity model

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

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

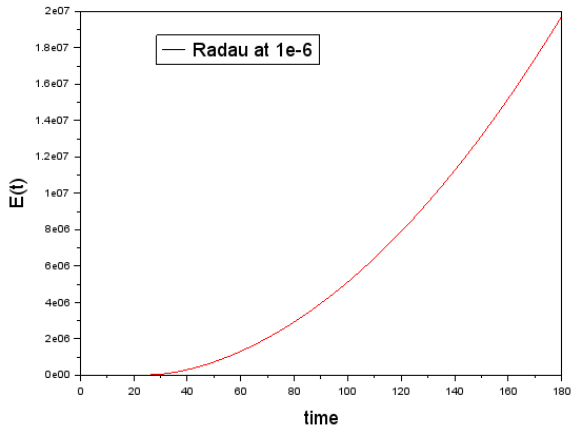

Figure 110: Solution from the Fortran radau5.f solver at tolerance of 10^{-6}.

From Figure 110, we again see the unusual behaviour. We also note that it behaves exactly as Radau does in the R environment. We then repeat the process with a tolerance of 10^{-12}. In Figure 111, we can see that the computed solution now follows the correct pattern, although it is still not as accurate as the solution that we described in Section 3.4.

From this investigation of the Fortran solver, we can conclude that the issue is not with the interface from R to the Fortran solver or the Python implementation. We also added 'print' statements during our investigation to confirm that the parameter β was set to 0.005 when appropriate. The issue appears to be with the 'Radau' algorithm itself. Further detailed investigation of the 'Radau' algorithm will be required in order to determine the source of this issue.

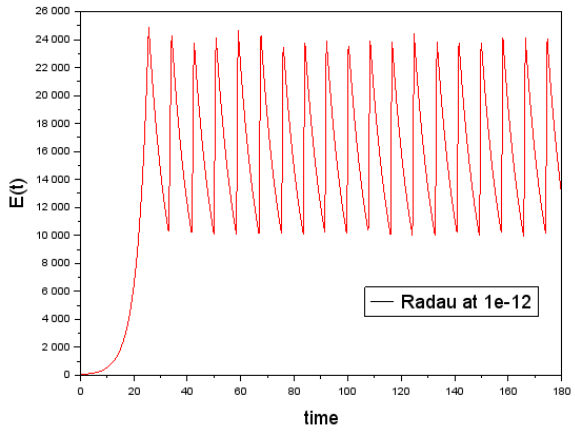

Figure 111: Solution of the state-dependent discontinuity model from the Fortran radau5.f solver at tolerance of 10^{-12}

6 Summary, Conclusions, and Future Work

6.1 Summary and Conclusions

In this report, we have considered the numerical solution of two Covid-19 models based on a standard SEIR model. The models include discontinuities associated with interventions introduced to slow down the spread of the virus. We were particularly interested in investigating the performance of a number of standard solvers available within several computational platforms.

We have discussed stability and discontinuity issues associated with the models. We showed how stability issues associated with the exponential growth of some of the solution components of the models affect the accuracy of the computed solutions. We also showed how discontinuities reduce the efficiency of the solvers and presented a straightforward way to detect that a model is discontinuous.

We then used ODE software packages in R, Python, Scilab, and Matlab to solve the two Covid-19 problems, one with a time-dependent discontinuity and one with a state-dependent discontinuity. *A critical starting assumption for both models is that we consider reasonable implementations that might typically be employed by a researcher.* This includes fixed-step size solvers as well as implementations based on the introduction of 'if' statements into the functions that define the ODE systems.

For the time-dependent discontinuity problem, we have shown that error-control ODE solvers can step over the one discontinuity that is present with sufficiently sharp tolerances while fixed step-size solvers cannot. We have shown that although error-controlled solvers can solve the problem to reasonable accuracy if the tolerance is sufficiently sharp, the use of discontinuity handling in the form of cold starts leads to more efficient solutions that can be obtained using coarser tolerances. We therefore recommend that if the time of a discontinuity is known, cold starts at these times should be employed as they result in more accurate, more efficient solutions that can be obtained at coarser tolerances.

For the state-dependent discontinuity problem, we have shown that even error control solvers cannot successfully step over multiple state-dependent discontinuities. We then introduced event detection and showed how it can be used to accurately and efficiently solve state-dependent discontinuity problems by encoding the intervention imposition and relaxation thresholds as events and applying cold starts. We conclude that using event detection provides an efficient and accurate way to solve such problems.

From the examination of the different packages, we also found a certain inconsistency. We noted that R and Scilab do not use the interpolation capabilities for some of their solvers by default. Using the method of forcing the solver to integrate exactly to given output points reduces the efficiency of the solver because the solver is no longer allowed to take as big a step as it should.

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

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

When a researcher has a problem that has a state-dependent discontinuity, we recommend that they determine the conditions under which these discontinuities occur and then 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 a level of efficiency and accuracy that is not possible using a simple implementation.

6.2 Future Work

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

We also propose to do the same type of analysis on Covid-19 PDE models with discontinuities to see how error-controlled and non-error-controlled PDE solvers differ. We will also investigate the use of a PDE solver with event detection for these models.

References

[1] C. Ohajunwa, K. Kumar, and P. Seshaiyer, "Mathematical modeling, analysis, and simulation of the COVID-19 pandemic with explicit and implicit behavioral changes," *Comput. Math. Biophys.*, vol. 8, pp. 216–232, 2020. [Online]. Available: https://doi.org/10.1515/cmb-2020-0113

[2] C. C. Christara, "Private communication," 2021.

[3] J. R. Dormand, *Numerical methods for differential equations*, ser. CRC Revivals. CRC Press, Boca Raton, FL; CRC Press, Boca Raton, FL, 2018, a computational approach, Reprint of the 1996 original [MR1383317], Library of Engineering Mathematics. [Online]. Available: https://doi.org/10.1201/9781351075107

[4] K. Soetaert, T. Petzoldt, and R. W. Setzer, "Solving differential equations in R: package desolve," *Journal of Statistical Software*, vol. 33, no. 1, pp. 1–25, 2010.

[5] A. C. Hindmarsh, "ODEPACK, a systematized collection of ODE solvers," in *Scientific computing (Montreal, Que., 1982)*, ser. IMACS Trans. Sci. Comput., I. IMACS, New Brunswick, NJ, 1983, pp. 55–64.

[6] P. E. Van Keken, D. A. Yuen, and L. R. Petzold, "DASPK: a new high order and adaptive time-integration technique with applications to mantle convection with strongly temperature- and pressure-dependent rheology," *Geophys. Astrophys. Fluid Dynam.*, vol. 80, no. 1-2, pp. 57–74, 1995. [Online]. Available: https://doi.org/10.1080/03091929508229763

[7] E. Hairer and G. Wanner, *Solving ordinary differential equations. II*, 2nd ed., ser. Springer Series in Computational Mathematics. Springer-Verlag, Berlin, 1996, vol. 14, stiff and differential-algebraic problems. [Online]. Available: https://doi.org/10.1007/978-3-642-05221-7

[8] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[9] P. Bogacki and L. F. Shampine, "A 3(2) pair of Runge-Kutta formulas," *Appl. Math. Lett.*, vol. 2, no. 4, pp. 321–325, 1989. [Online]. Available: https://doi.org/10.1016/0893-9659(89)90079-7

[10] "Hairer's website," http://www.unige.ch/ hairer/software.html, accessed: 2021-08-01. [Online]. Available: http://www.unige.ch/ hairer/software.html

[11] S. L. Campbell, J.-P. Chancelier, and R. Nikoukhah, "Modeling and simulation in SCILAB," in *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. Springer, 2010, pp. 73–106.

[12] L. F. Shampine, *Numerical solution of ordinary differential equations*. Chapman & Hall, New York, 1994.

[13] "Scilab github," https://github.com/scilab/scilab, accessed: 2021-08-01. [Online]. Available: https://github.com/scilab/scilab

[14] L. F. Shampine and H. A. Watts, "Practical solution of ordinary differential equations by Runge–Kutta methods," 12 1976. [Online]. Available: https://www.osti.gov/biblio/7318812

[15] L. F. Shampine and M. W. Reichelt, "The Matlab ODE suite," *SIAM Journal on Scientific Computing*, vol. 18, no. 1, pp. 1–22, 1997.