

Performance Analysis on Covid-19 Models with Discontinuities and Efficient Defect Control for Initial Value ODE solvers

Humaid M. Agowun

Saint Mary's University

2022

Modelling Covid-19 using an SEIR Ordinary Differential Equation (ODE) System

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

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

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

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

$S(t)$ is the number of susceptible individuals, $E(t)$ is the number of exposed individuals, $I(t)$ is the number of infected individuals and $R(t)$ is the number of recovered individuals at a point in time.

Solving the Model

We define the ODE problem as a model function, give some initial values, and the times at which to sample the solution, $y(t) = (S(t), E(t), I(t), R(t))$

```
function f(t, y):  
    (S, E, I, R) = y  
    dSdt = mu*N - mu*S - (beta/N)*I*S  
    dEdt = (beta/N)*I*S - alpha*E - mu*E  
    dIdt = alpha*E - gamma*I - mu*I  
    dRdt = gamma*I - mu*R  
    return (dSdt, dEdt, dIdt, dRdt)
```

```
E0 = 103; I0 = 1; R0 = 0; N = 37.741 * (10**6)  
S0 = N - (E0 + I0 + R0); y0 = (S0, E0, I0, R0)  
t_eval = [0, 1, 2, ..., 95]; tol = 1e-6
```

```
res = ode_solver(f, t_eval, y0, tol)  
plot(t_eval, res)
```

Typical Plot of Solutions to an SEIR Model

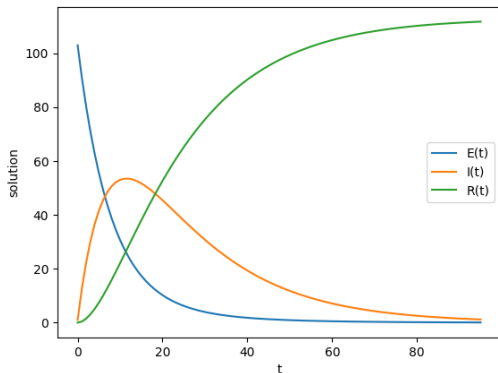


Figure: Solutions to the SEIR model. Note that $S(t)$ is taken to be large enough that it cannot be plotted alongside $E(t)$, $I(t)$ and $R(t)$

Models that include Preventive Measures

Vary the β parameter to model the introduction of Covid-19 measures. The if-statement introduces a discontinuity.

```
function f(t, y):  
    (S, E, I, R) = y  
    // parameters ...  
    if (t < 27):  
        beta = 0.9 # no measures  
    else:  
        beta = 0.005 # severe drop in transmission  
    // ...  
    return (dSdt, dEdt, dIdt, dRdt)
```

How Modern ODE Solvers Work

ODE solvers work by taking a sequence of time steps.

Using an initial step-size h , a solver takes a step from t_0 to $t_0 + h$ using some evaluations of the function $f(t, y)$.

The solver also calculates an error estimate at the end of the step. If that error estimate is within the user provided tolerance, the solver accepts the step. Else, the solver retakes the step with a smaller step-size (e.g. $\frac{h}{2}$).

A solver that employs this type of algorithm is called an error control solver.

Thrashing

The if-statement makes $f(t, y(t))$ discontinuous. This causes difficulties for ODE solvers. *Error control* solvers can solve this type of discontinuous problem but the efficiency is impacted.

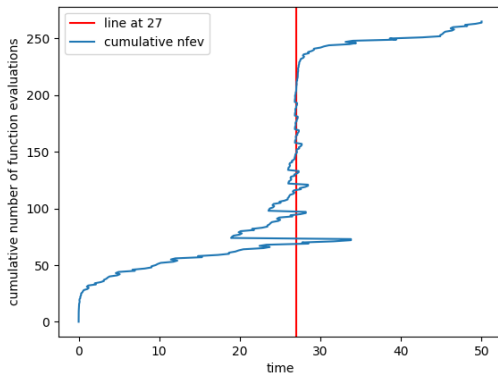


Figure: Number of function evaluations spike at a discontinuity in a process called 'THRASHING', i.e., repeated step-size reductions.

Impacts of Time-dependent Discontinuity

Thrashing impacts the efficiency because the solvers are forced to take very small steps. Thrashing sometimes impacts the accuracy, as some solvers accept an inaccurate step.

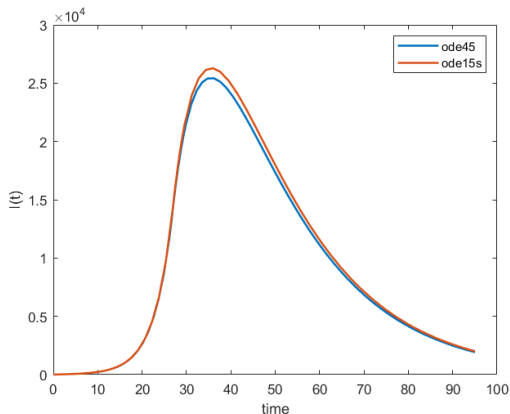


Figure: Robust MATLAB solvers giving different solutions to discontinuous problems.

Solution to Time-dependent Discontinuity

The key idea is to use a COLD START at the discontinuity. No values from the previous call impact the new call. Each call solves a continuous subproblem.

```
y0_1 = (S0, E0, I0, R0)
tspan1 = [0, ..., 27]
sol1 = ode(y0_1, f_0_9, tspan1, 1e-6)

y0_2 = last_row(sol1)
tspan2 = [27, ..., 95]
sol2 = ode(y0_2, f_0_005, tspan2, 1e-6)

sol = concat(sol1, sol2)
```

Impacts of Cold Starts

Cold starts address the efficiency and accuracy problems.

Table: 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 solution is more accurate as well. For example, the MATLAB ode45 solver accurately solves the problem when cold starts are used.

Matlab solvers accurately solve the time-dependent discontinuity model when using a cold start.

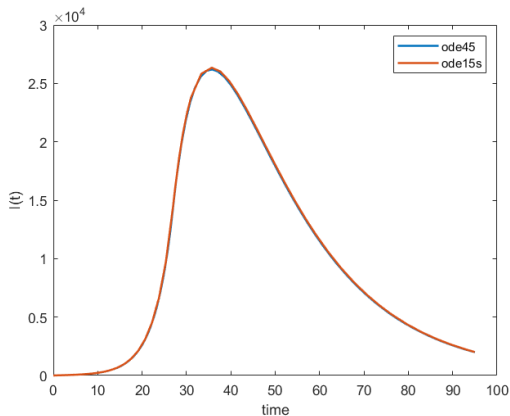


Figure: Robust MATLAB solvers giving accurate solutions to discontinuous problems.

State-dependent Discontinuity

Previously, the if-statement was based on the time. We are now going to discuss state-dependent discontinuities.

```
measures_implemented = False
direction = "up"
function model_with_if(_, y):
    // ...
    if (direction == "up" and E > 25000):
        measures_implemented = True
        direction = "down"
    elif (direction == "down" and E < 10000):
        measures_implemented = False
        direction = "up"

    if measures_implemented: beta = 0.005
    else: beta = 0.9
    // ...
    return (dSdt, dEdt, dIdt, dRdt)
```

Impacts of the State-dependent Discontinuities

Everytime the value of the parameter β changes, there is a discontinuity. Standard ODE solvers, even when using sharp tolerances, cannot solve it with reasonable accuracy.

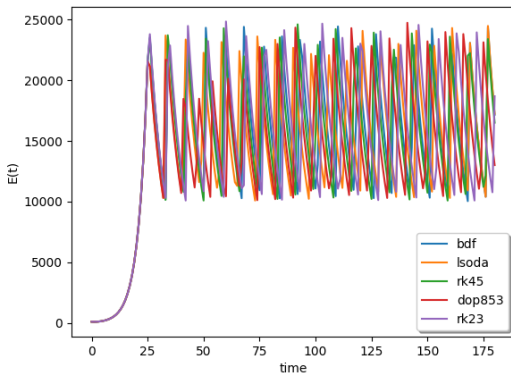


Figure: Solutions to the state-dependent problem using the simple approach with Python solvers at a very sharp tolerance.

Event Detection

Event detection requires the user to also provide a root function, $g(t, y(t))$, that defines the events. An event is said to occur when the root function returns 0.

The solver will call the root function at the end of each successful step. It caches the previous value of the root function and if there is a change in sign between the previous and the current value, this signals to the solver that there is a root somewhere within the step.

The solver will then run a root-finding algorithm (Bisection, BRENT, ...) to detect the location of the root.

For example, if we want to detect when y is 100, it is sufficient to define the root function as *return (y - 100);*.

Solving the State-dependent Discontinuity Problem

```
res = array(); t0 = 0; y0 = (S0, E0, I0, R0)
while t0 < 180:
    tspan = [t0, ..., 180]
    if (measures_implemented):
        sol = ode(f_0_005, tspan,
                  y0, events=root_10000)
        measures_implemented = False
    else:
        sol = ode(f_0_9, tspan,
                  y0, events=root_25000)
        measures_implemented = True
    t0 = extract_last_t(sol)
    y0 = extract_last_row(sol)
    res = concatenate(res, sol)
```

In this way, each call to the solver involves a continuous subproblem.

Accuracy of Event Detection Solution

Recall that none of the solutions from the solvers in Python were aligned even at a sharp tolerance. With event detection, all the solutions are aligned. Thus the accuracy is greatly improved.

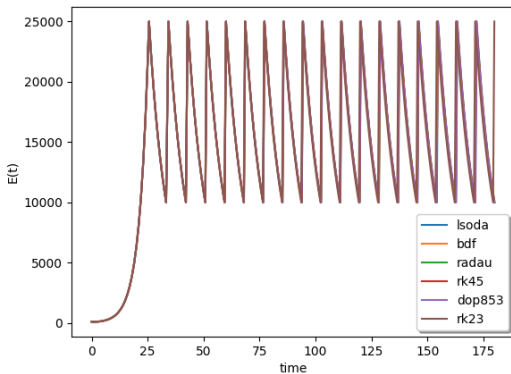


Figure: Solutions to the state-dependent problem using event detection in Python.

Efficiency of Event Detection solution

Table: Efficiency data for Python solvers applied to the 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

Significant reductions in number of function evaluations, even when compared to the default tolerance number of function evaluations. The computed solutions are much more accurate as well.

Summary - Solving Discontinuous Covid-19 Models

Time-dependent discontinuity: Instead of using an if-statement based on the time at which the discontinuity arises, t_{if} , integrate in two parts with a cold start between, i.e, integrate $[t_0, t_{if}]$, cold start, then integrate $[t_{if}, t_f]$.

State-dependent discontinuity: Instead of using an if-statement based on state thresholds, use root functions that detect when a threshold is reached, integrate until a root is detected, cold start, then integrate after the root with the new model.

Efficient Defect Control - Introduction

In standard ODE solvers, error control is only applied at the end of each time step. Standard ODE solvers produce continuous solutions by interpolating the discrete solution values.

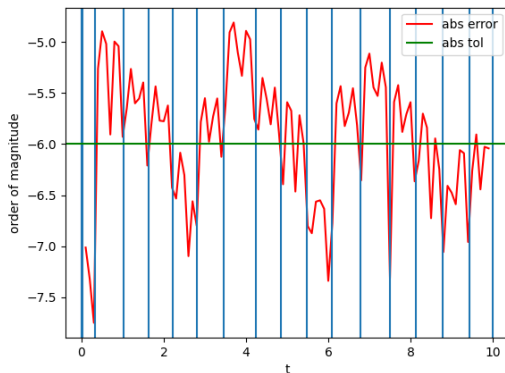


Figure: Robust Python solver 'RK45' produces solutions with errors of one order of magnitude larger than the tolerance within the step.

Defect Control

Users of ODE solvers expect that the solution is accurate to within the tolerance throughout each step. The challenge is thus to produce ODE solvers that provide continuous approximate solutions that satisfy the tolerance. We do so by performing defect control.

The defect, denoted by $\delta(t)$, is the amount by which the continuous numerical solution, $u(t)$, fails to satisfy the ODE. When the ODE is $y'(t) = f(t, y(t))$, the defect is

$$\delta(t) = |u'(t) - f(t, u(t))|. \quad (1)$$

A defect control solver takes a step using a traditional numerical method, extends the discrete approximate solution to obtain a continuous approximate solution and then samples the defect within the step to estimate its maximum value. If that maximum defect is within the tolerance, the solver accepts the step, else it rejects it and takes a smaller step.

RK4

In this presentation, we will augment the Classical Runge-Kutta method of 4th order (RK4) with interpolants of order 4, 6 and 8 to provide defect controlled solutions.

$$k_1 = hf(y_i, t), \quad (2)$$

$$k_2 = hf(y_i + \frac{k_1}{2}, t + \frac{h}{2}), \quad (3)$$

$$k_3 = hf(y_i + \frac{k_2}{2}, t + \frac{h}{2}), \quad (4)$$

$$k_4 = hf(y_i + k_3, t + h), \quad (5)$$

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \quad (6)$$

Related Work

Related work to produce a continuous approximate solution involves using a continuous Runge Kutta method. These methods require the computation of additional evaluations of $f(t, y)$ beyond those required by the discrete Runge-Kutta methods. In our work, we use methods that use no additional function evaluation to produce a continuous approximate solution. This significantly improves the efficiency.

Table: Number of $f(t, y)$ evaluations for discrete vs continuous RK method

order	discrete	continuous
4	4	4
5	5	6
6	6	7
7	7	9
8	8	13

HB4

The Hermite cubic is a popular 4th order interpolant. That is, if the space between interpolated data values, h , is reduced to $h/10$, we expect the interpolation error over the new set of data values to be $e/10^4$, where e was the interpolation error with a spacing of h . For the step from t_i to t_{i+1} , the HB4 scheme is constructed to interpolate both the solution value and derivative value at t_i and t_{i+1} .

$$u(t_i + \theta h) = h_{00}(\theta)y_i + h_i h_{10}(\theta)f_i + h_{01}(\theta)y_{i+1} + h_i h_{11}(\theta)f_{i+1}, \quad (7)$$

$$\theta = (t - t_i)/h_i. \quad (8)$$

When θ is 0, $t_i + \theta h$ is t_i . When θ is 1, $t_i + \theta h$ is t_{i+1} . Using the above, we can produce 4 linear systems of 4 equations to derive the 4 cubic polynomials, $h_{00}(\theta)$, $h_{01}(\theta)$, $h_{10}(\theta)$ and $h_{11}(\theta)$.

RK4 with HB4

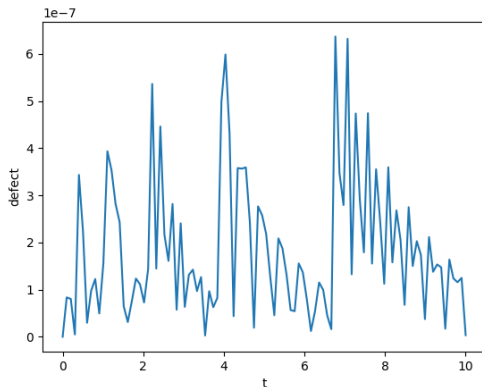


Figure: Augmenting RK4 with HB4 successfully controls the defect at a tolerance of 10^{-6} .

Issues for RK4 with HB4

The Hermite cubic, HB4, is an interpolant of 4^{th} order. To perform defect control we need its derivative. Differentiating a Hermite interpolant decreases the order by 1. The order of the derivative is 3.

The interpolation error thus affects the computation of the defect because the numerical solution is 4^{th} order but the derivative is of 3^{rd} order.

Derivation of HB6 and weight parameters

HB6 is *multistep*. It will require 3 data points and will be defined over 2 steps. Those 2 steps may be of different size. We thus introduce a weight parameter α . We assume that the step from t_i to t_{i+1} is of size h but the step from t_{i-1} to t_i is αh . The interpolant interpolates the solution and derivative values from the 3 data points.

$$\begin{aligned} u(t_i + \theta h) = & d_0(\theta)y_{i-1} + h_i d_1(\theta)f_{i-1} + d_2(\theta)y_i \\ & + h_i d_3(\theta)f_i + d_4(\theta)y_{i+1} + h_i d_5(\theta)f_{i+1}, \end{aligned} \tag{9}$$

When θ is $-\alpha$, $t_i + \theta h$ is t_{i-1} . When θ is 0, $t_i + \theta h$ is t_i . When θ is 1, $t_i + \theta h$ is t_{i+1} . Using the above, we can produce 6 linear systems of 6 equations to derive the coefficients of the 6 quintic polynomials $d_i(\theta)$, $i = 0, \dots, 5$ in terms of α .

RK4 with HB6

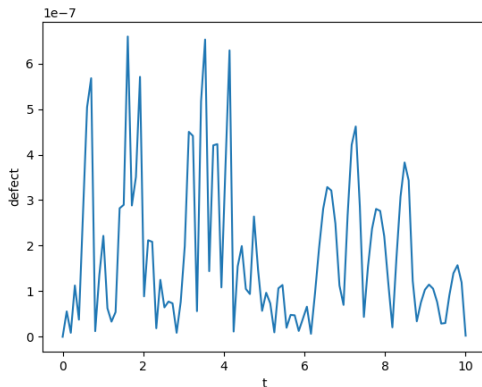


Figure: Augmenting RK4 with HB6 successfully controls the defect at a tolerance of 10^{-6} .

RK4 with HB6 is more efficient than RK4 with HB4

Across the several problems that we used to test RK4 with HB4 and RK4 with HB6, we find that RK4 with HB6 tends to take three times fewer steps than RK4 with HB4. (For example, in the problem that we plot in this talk, RK4 with HB6 used 27 steps when RK4 with HB4 used 88 steps.)

HB6 is of 6^{th} and its derivative is of 5^{th} order. Thus its interpolation error will be smaller than the numerical solution error.

This improvement in efficiency is entirely because of having a smaller interpolation error.

To produce an 8^{th} order interpolant, we need to use 4 data points and thus three steps. We thus use two parameters α and β such that the step-sizes are αh for the step from t_{i-2} to t_{i-1} , h for the step from t_{i-1} to t_i and βh for the step from t_i to t_{i+1} . The equation for the interpolant is:

$$\begin{aligned} u(t_i + \theta h) = & d_0(\theta)y_{i-2} + h_id_1(\theta)f_{i-2} + d_2(\theta)y_{i-1} + h_id_3(\theta)f_{i-1} \\ & + d_4(\theta)y_i + h_id_5(\theta)f_i + d_6(\theta)y_{i+1} + h_id_7(\theta)f_{i+1}. \end{aligned} \quad (10)$$

The interpolant is required to interpolate the solution and derivative at t_{i-2} , t_{i-1} , t_i and t_{i+1} . When θ is $-1 - \alpha$, $t_i + \theta h$ is t_{i-2} . When θ is -1 , $t_i + \theta h$ is t_{i-1} . When θ is 0 , $t_i + \theta h$ is t_i . When θ is β , $t_i + \theta h$ is t_{i+1} . Using the above, we can produce 8 linear systems of 8 equations to derive the coefficients of the 8 septic polynomials $d_i(\theta)$, $i = 0, \dots, 7$ in terms of α and β .

RK4 with HB8

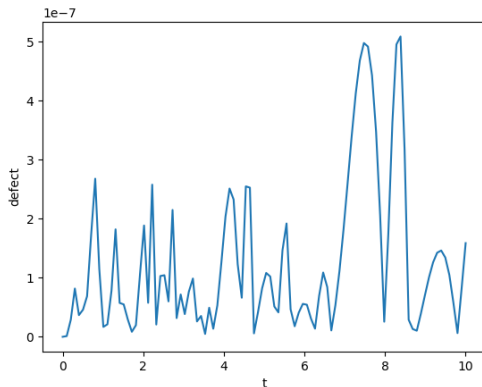


Figure: Augmenting RK4 with HB8 successfully controls the defect at a tolerance of 10^{-6} .

Use of HB8 for higher order Runge-Kutta methods

RK4 with HB8 is not significantly more efficient than RK4 with HB6. They take about the same number of steps across the problems we used. (For example, in the problem plotted in this talk, RK4 with HB6 used 27 steps but RK4 with HB8 used 20 steps. Remember that RK4 with HB4 used 88 steps.)

The main purpose of HB8 is that it can be used to efficiently augment a 6th order Runge-Kutta method.

It can also be used to provide defect control for an 8th order Runge-Kutta method though that would not be optimal as the derivative of the interpolant is only of 7th order when the discrete solutions are of 8th order.

Summary

We have used 4^{th} , 6^{th} and 8^{th} order interpolants to provide defect control to RK4.

These methods are more efficient as they require no additional function evaluations unlike the standard approach which makes use of continuous Runge-Kutta methods. They are thus more efficient.