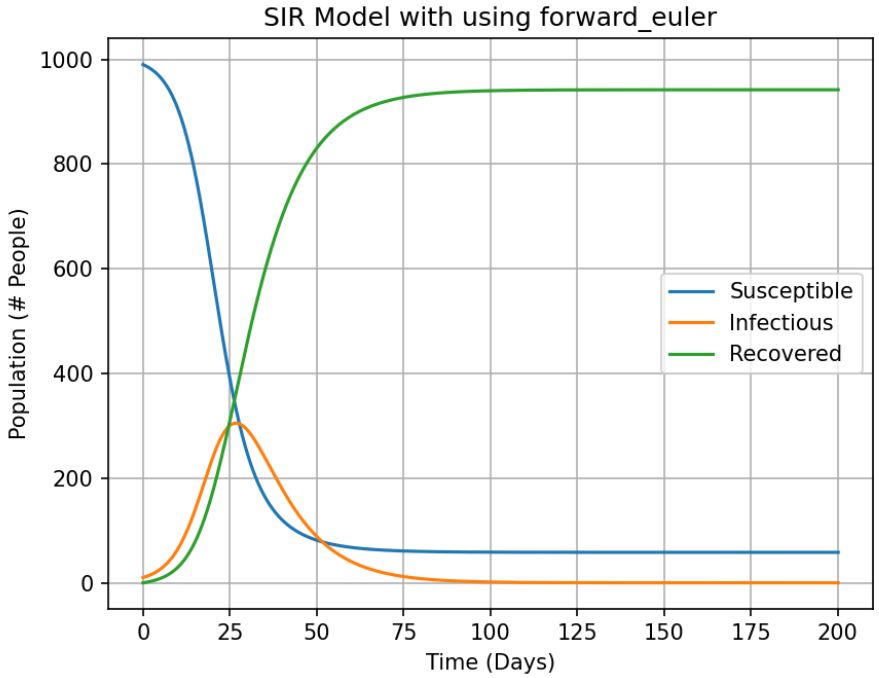
An Explanation of and Comparison of ODE Solvers Applied to an SIR Model

Tom Reed

February 2024



ABSTRACT

In this report, we explain the derivations and workings of several numerical methods used to solve ODEs. We then introduce deterministic SIR models and their definition. We use the ODE_solvers package to apply the different ODE methods to solving the SIR model equations. We discuss the run-times of the methods across a small β-parameter space of the SIR model.

ODE-solver package

The ODE-Solvers package is found at https://github.com/tomcodewizard/ODE-solvers

The package contains a number of numerical methods for solving ODEs within the solver folder. Within this folder, the script ODE_methods.py contains functions carrying out the following solver algorithms:

- Forward Euler
- Backwards Euler
- Midpoint Method
- Heun's Method
- Runge-Kutta RK4
- Adams-Bashforth
- Adams-Moulton

Within this script, each method has two functions; for example forward_euler_singular and forward_euler. The former is coded specifically to solve a single ODE. On the other hand, the latter is optimised to solve systems of ODEs by allowing list inputs for the initial values.

To install the ODE-Solvers package, consult the README.md file.

Numerical Solvers

We will consider ODEs of the initial-value problem form:

$$\frac{dy}{dt} = f(t, y) \tag{1}$$

$$y(0) = y_0$$

where $y \equiv y(t)$ is a function of time t with initial value y_0

The numerical solvers of ODEs described here can be derived Taylor series and Polynomial Interpolation, where we try to approximate the derivative by a series of known terms. In order to do this, we have to split the time period into a series of time-steps. Then, we can derive recurrence relations for the value of y at further time-steps.

- **Local Error**: The error occurring at every step of the method
- **Global Error**: The error between the true value of y and approximation y_n

The local error can be found as the largest order term of the Taylor series being ignored in the derivation. The global error can usually be found as the product of the number of steps and the local error [1]. We say that the method is "First order, Second order, ..." if the global error of the method is proportional to $h, h^2, ...$ etc.

We say the method is unstable if the errors increase as the method iterates. On the other hand, a method is stable if the error decreases over time, such that the approximation of y approaches its true value.

Increasing the step-size may cause methods to become unstable, as the errors in the tangent approximations accumulate. Therefore, a method should be analysed and compared with other methods to see the ranges of stability.

Forward Euler

The Forward Euler method is the most basic of the methods presented here. The method is derived from taking the first two terms of the Taylor series for y:

$$y(t) = y(a) + y'(a)(t-a) + ...$$

Notice that $y' = \frac{dy}{dt}$ and let the time-step be $t_{n+1} - t_n = h$

Then,

$$y(t_{n+1}) = y(t_n) + f(t_n, y_n)(t_{n+1} - t_n) + ...$$

$$\implies y(t_{n+1}) \approx y(t_n) + f(t_n, y_n) \cdot h$$

This gives us an approximation for y at time t_{n+1} given the state of y at time t_n. This is called an **explicit** method, since all the terms in the recurrence relation needed to find the next value of y are known each iteration. Explicit methods may become unstable for some values of h.

The local error is proportional to h^2 - this is the term ignored in the Taylor approximation. At t_n, the number of steps is proportional to $\frac{1}{h}$, so the global error is $h^2 \cdot \frac{1}{h} = h$. Therefore, the Forward Euler is a first order method.

All together, the Forward Euler recurrence relation is given by:

$$\begin{aligned} y_{n+1} &= y_n + h \cdot f(t_n, y_n) \\ t_{n+1} &= t_n + h \end{aligned} \tag{2}$$

Backwards Euler

The Backwards Euler method uses the same derivation as the Forward Euler method, but evaluates the derivative at the time t_{n+1} instead of t_n. This means that the slope is now being estimated from the right hand side of each time interval, backwards.

This is harder to solve, as the value of $f(t_{n+1}, y_{n+1})$ is unknown at time t_n. This makes it an **implicit** method. This means that the method will remain stable for all values of h. This comes at a cost as a method must be run to solve the recurrence relation implicitly at each time step. This is done in ODE-Solver using the scipy.optimize.fsolve function. This is a root-finding algorithm, which will find the values of y_{n+1} needed to 'zero' the relation, i.e. solve it.

It can be shown identically, that the Backwards Euler is a first order method in error.

The Backwards Euler recurrence relation is given by:

$$\begin{aligned} y_{n+1} &= y_n + h \cdot f(t_{n+1}, y_{n+1}) \\ t_{n+1} &= t_n + h \end{aligned} \tag{3}$$

Midpoint Method

If we instead evaluate the derivative term at the midpoint of each time interval $t_n + \frac{h}{2}$, we get:

$$\frac{y_{n+1} - y_n}{h} \approx f\left(t_n + \frac{h}{2}, y\left(t_n + \frac{h}{2}\right)\right)$$

$$\implies y_{n+1} \approx y_n + h \cdot f\left(t_n + \frac{h}{2}, y\left(t_n + \frac{h}{2}\right)\right)$$

We then use the Forward Euler method to approximate $y\left(t_n + \frac{h}{2}\right)$

$$y\left(t_n + \frac{h}{2}\right) \approx y_n + \frac{h}{2} \cdot f(t_n, y_n)$$

Putting these together, gives the **explicit** Midpoint Method:

$$y_{n+1} = y_n + h \cdot f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} \cdot f(t_n, y_n)\right)$$

Note that this method is second-order in error, as the substitution of a first order method into another leads to the error becoming a product of the two, i.e. $h \cdot h = h^2$.

The Midpoint Method recurrence relation is given by:

$$\begin{aligned} y_{n+1} &= y_n + h \cdot f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2} \cdot f(t_n, y_n)\right) \\ t_{n+1} &= t_n + h \end{aligned} \tag{4}$$

Heun's Method

Heun's method is derived by modifying the Forward Euler method, such that the derivative term is given by the average of f evaluated at the left and right of the interval.

$$y_{n+1} = y_n + \frac{h}{2} \cdot [f(t_n, y_n) + f(t_{n+1}, y_{n+1})]$$

Again, this is an **implicit** method as the right-hand side is unknown.

We again estimate this using the Forward Euler method.

This is the **predictor** term and is denoted using the '$*$'

$$y^*_{n+1} = y_n + h \cdot f(t_n, y_n)$$

Then, this is fed into the average term - the **estimator**.

$$y_{n+1} = y_n + \frac{h}{2} \cdot [f(t_n, y_n) + f(t_{n+1}, y^*_{n+1})]$$

Again, Heun's method is a second order method.

All-together, the Heun's Method recurrence relation is given by:

$$\begin{aligned} y^*_{n+1} &= y_n + h \cdot f(t_n, y_n) \\ y_{n+1} &= y_n + \frac{h}{2} \cdot [f(t_n, y_n) + f(t_{n+1}, y^*_{n+1})] \\ t_{n+1} &= t_n + h \end{aligned} \tag{5}$$

Runge-Kutta RK4

Runge-Kutta methods have a greater accuracy than the Euler-derived methods by evaluating f at a number of points in the time/space interval.

The most commonly used, called RK4, uses 4 predictive steps built upon midpoints. Despite the increased accuracy, this method is still **explicit**.

As hinted by the name, RK4 is a 4th order method, so the global error is proportional to h^4.

The Runge-Kutta recurrence relation is given by:

$$
\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f(t_n + \frac{1}{2} \cdot h, y_n + \frac{1}{2} \cdot k_1) \\
k_3 &= f(t_n + \frac{1}{2} \cdot h, y_n + \frac{1}{2} \cdot k_2) \\
k_4 &= f(t_n + h, y_n + h \cdot k_3) \\
y_{n+1} &= y_n + \frac{h}{6} \cdot [k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4] \\
t_{n+1} &= t_n + h
\end{aligned}
\tag{6}
$$

Adams-Bashforth

The Adams-Bashforth method is a multi-step method, meaning y_{n+1} is dependent on not only y_n, but also y_{n-1}. Therefore, the method needs two initial conditions to start the method. In the package ODE-Solvers, the first step is a Forward Euler step:

$$y_1 = y_0 + h \cdot f(t_0, y_0)$$

The method can be derived using Polynomial interpolation or by taking more terms in a Taylor series assumption. See [3] for example derivations.

The method is second order.

The Adams-Bashforth recurrence relation is given by:

$$\begin{aligned} y_{n+2} &= y_{n+1} + \frac{h}{2} \cdot [3 \cdot f(t_{n+1}, y_{n+1}) - f(t_n, y_n)] \\ t_{n+1} &= t_n + h \end{aligned} \tag{7}$$

Adams-Moulton

The Adam-Moulton method is derived in the same way in [3] using Polynomial Interpolation.

This method is explicit, so the relation must use another method to be solved at each time-step.

Similarly, the method is second order.

The Adams-Moulton recurrence relation is given by:

$$\begin{aligned} y_{n+2} &= y_{n+1} + \frac{h}{2} \cdot [f(t_{n+2}, y_{n+2}) + f(t_{n+1}, y_{n+1}))] \\ t_{n+1} &= t_n + h \end{aligned} \tag{8}$$

SIR Model

The SIR model is a system of three differential equations, used in epidemiology to model the spread of infections and progression of pandemics [2]. This model is a *compartment*-based model, where each individual in a population can only belong in one state - S, I or R in this model. Each compartment is defined as such:

S: Number of Susceptible individuals
I: Number of Infected individuals
R: Number of Recovered individuals

The mechanics of the most basic SIR model are given by the following system of ODEs, which we will be studying using the numerical methods we have discussed:

$$\begin{aligned} \frac{dS}{dt} &= -\frac{\beta}{N} \cdot S \cdot I \\ \frac{dI}{dt} &= \frac{\beta}{N} \cdot S \cdot I - \gamma \cdot I \\ \frac{dR}{dt} &= \gamma \cdot I \end{aligned} \tag{9}$$

where,

$N = S + I + R$ is the total population

$\beta > 0$ is the transmission rate

$\gamma > 0$ is the recovery rate

The progression of a disease can be seen by plotting the S, I and R curves together against time.

Applying Methods to SIR Model

The SIR model was implemented into the SIR_example.py script. The script loops through each of the ODE solver methods from ODE_methods.py and solves the SIR model over some fixed parameters. The script can then plot the results when run_solver() is given the argument plot=True. Comparison of each of the plots showed that each method produced the same plot, see Figure 1. This showed that the methods were stable and correct for the parameters tested.

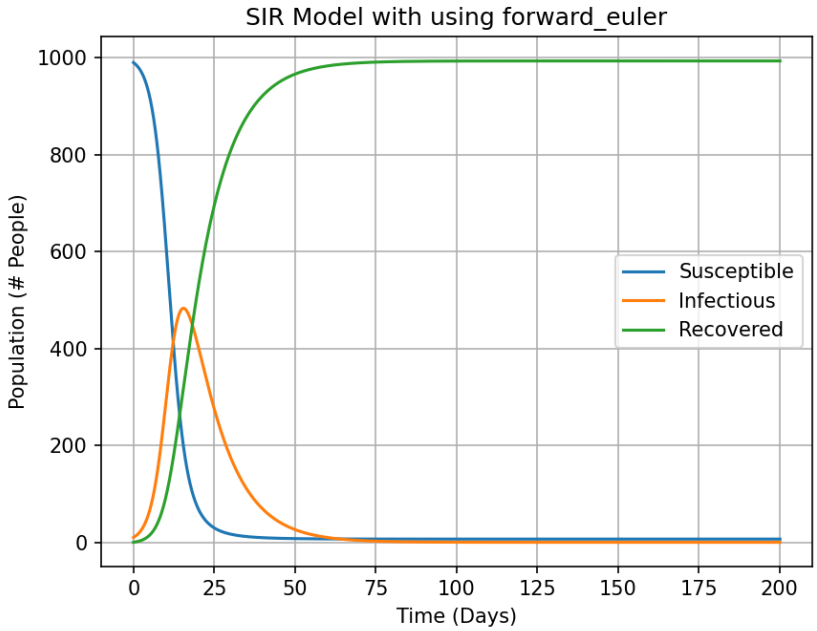


Figure 1: This simulation was run with the parameters $S_0 = 990$, $I_0 = 10$, $R_0 = 0$, $\beta = 0.5$, $\gamma = 0.1$, $t_0 = 0$, $t_{final} = 200$ and $h = 0.1$.

The, the run-times of each method was investigated. A run-time decorator found in utility .py was applied to the method to get the run-times of each method. These times were put into a spreadsheet recorded_solve_times.xlsx. The parameters passed through were all fixed, apart from the transmission rate β which was varied through values $\beta = 0.2, 0.5, 0.8$. The decorator was very simple to apply once it had been coded up, but a limitation of it was that the times had to be manually written to the spreadsheet. Furthermore, a more accurate run-time would have been given by averaging run-times over multiple runs of the same parameter cases. Also, due to time-constraints, only a small β-parameter space was investigated.

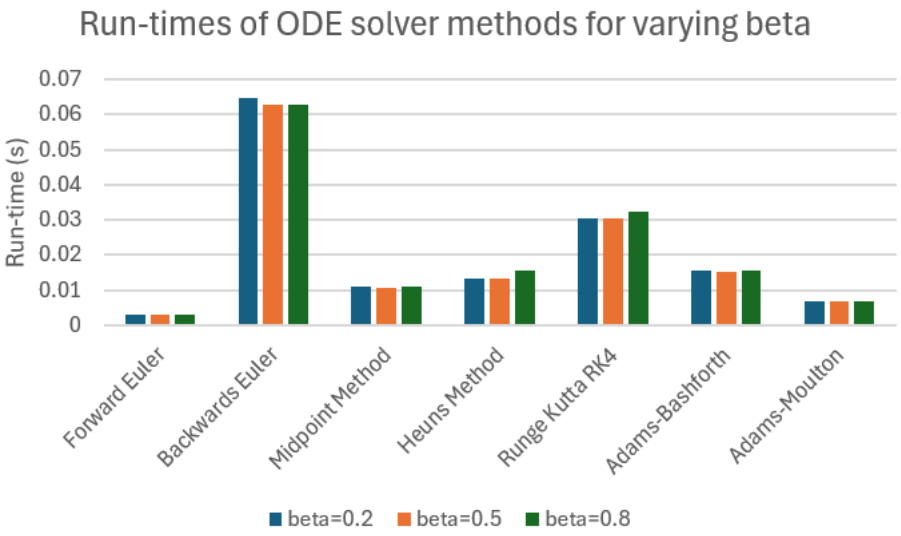


Figure 2: The run-times of the ODE methods using time.perf_counter() for values $\beta = 0.2, 0.5, 0.8$. These simulations were run with the parameters $S_0 = 990$, $I_0 = 10$, $R_0 = 0$, $\gamma = 0.1$, $t_0 = 0$, $t_{final} = 200$ and $h = 0.1$. It is clear that varying β does not impact the run-times of the each method much.

Figure 2 shows the results of the experiments. It was found that the run-times varied very little when varying the parameter for each ODE method. Therefore, it is not clear if there is a small dependence. However, it seems likely that this is due to noise in the run-times and taking the average over a large number of runs would make this more evident.

These results were averaged over all β-values for each method and collated into Figure 3, to make comparison of methods clearer. The Figure clearly shows that Forward Euler was the fastest to run, followed by Adams-Moulton and the Midpoint Method. Backwards Euler by far took the longest to run, folowed by the Runge Kutta RK4. It is worth keeping in mind that the run-times were of the order of a hundreth of a second, so these comparisons show more of an insight into how the run-times may differ for a more complicated system.

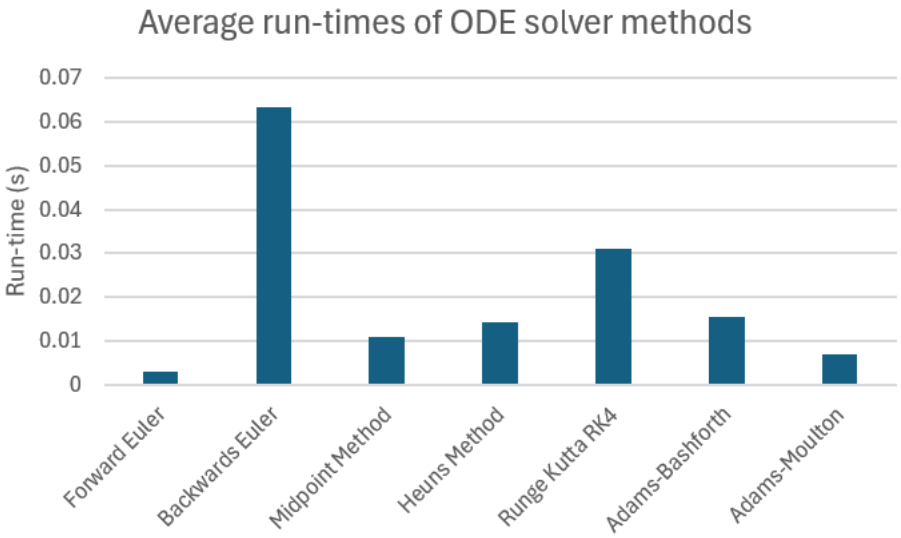


Figure 3: The results from Figure 2, where the β runs have been averaged, to give a singular run-time for each method.

We may imply from our results that the Forward Euler is the best method for solving the SIR method, with the quickest run-time and easiest implementation However, we have seen that the method may suffer from instability and has a relatively large order of error compared to the other methods.

Therefore, multi-step methods such as the Adams methods provide a good compromise of time efficiency with smaller error. In particular, the Adams-Moulton method was the second fastest and is stable for all time-step values.

References

[1] NTNU. Errors and stability, 2017.

[2] Emma Southall, Z Ogi-Gittins, AR Kaye, WS Hart, FA Lovell-Read, and RN Thompson. A practical guide to mathematical methods for estimating infectious disease outbreak risks. *Journal of Theoretical Biology*, page 111417, 2023.

[3] Wikiversity. Adams-bashforth and adams-moulton methods, 2014.