Scientific Software / Technisch-wetenschappelijke software: Assignment 1

Victor Lepère

October 31, 2023

1 What are the results of your first program? Do you notice any differences?

Single precision:

r	gfortran	ifort	nagfor
0.01	304940576.	3.0494058E+08	304940576.
0.009765625	215269968.	2.1526997E+08	215269968.

Double precision:

r	gfortran	ifort	nagfor
0.01	304944890.44090623	304944890.440906	304944890.44090623
0.009765625	215269978.88932064	215269978.889321	215269978.88932064

Although the results are identical, the display is different. Using output formatting, we can see that the predicted number of infections does not vary from one compiler to another.

The results in double precision are very close to the exact value, which is not surprising since more bits are available to represent the decimal part of the number and avoid rounding errors and error propagation. Moreover, in single precision, the error with $r = 0.01$ is considerably higher than with $r = 0.009765625$. Given that floating-point numbers cannot represent the entire continuum of real numbers, the calculation involving a growth rate of $r = 0.01$ was more susceptible to requiring real numbers that fell within the iteration and needed to be approximated. The exponential nature of the calculation further magnified these approximation errors.

2 How did you design your code? Explain how you split functionality across modules/subroutines/functions. Why?

The code[1] has been divided into several procedures. The primary benefit of using subroutines and functions is that it enables to use a piece of code several times without having to duplicate it explicitly. All of these procedures are also encapsulated within a module which can be summarized by the following abstract API :

[1] A `readme` file is available to help compile.

```
!! private auxiliary procedures
function initialize()
function f(xk)
function jac_f(xk)
function norm(x)
subroutine PrintAndSave(tk, x)

!! public subroutines
subroutine EulerForward(T, N, tk, x)
subroutine Heun(T, N, tk, x)
subroutine EulerBackward(T, N, tk, x, tol)
```

These procedures are implemented using a combination of subroutines and functions, depending on the specific task at hand.

On the one hand, functions are primarily used when the procedure's output consists of a single value or a fixed-size array, and this choice is also motivated by readability concerns.

On the other hand, the three numerical methods (`EulerForward`, `Heun`, `EulerBackward`) are implemented by means of subroutines. This decision is driven by the need to avoid stack overflow issues. Indeed, a large sizable, size-dependant array is required to store the approximate solution for all the grid points during the simulation. If this were allocated and returned at each call of one of the methods, it would lead to rapid increases in memory consumption, potentially causing stack saturation. Instead, by using two of the arguments provided to the subroutine (`tk` and `x`) to store the results, it allows to reuse the same arrays instantied by the user multiple times. Furthermore, the use of a subroutine is all the more appropriate since several arrays need to be returned.

A subroutine is also used to implement `PrintAndSave` because it does not need to return anything.

Using a module bring several advantages to the code design. First, the accessibility of the procedures can be easily managed by means of the `private` and `public` keywords which improves the robustness and the understanding of the code from the outside. Then, it defines an *explicit interface* which enables the compiler to check if the routines are being called in the right way and flag any errors. Thanks to the module, the compiler is also able to pass implicit information about the arrays passed to the procedures such that their dimensions (which can be retrieved later via the `size` intrinsic).

As requested in the assignment statement, here is a short description of each procedure and module :

initialize	Auxiliary function for initializing the state vector of the SIQRD model
f	Auxiliary function for extracting the system parameters from *parameters.in* describing the SIQRD model and for returning the corresponding evalutation of xk as a function of these parameters
jac_f	Auxiliary function for extracting the system paramteres from *arameters.in* describing the SIQRD model and for returning the jacobian matrix of f evaluated at xk
norm	Simple vector's euclidian norm computation
PrintAndSave	Auxiliary function for displaying the calculated solution on the flow output and saving it in a sol.dat file
EulerForward	Euler's forward method tailored to the SIQRD model
Heun	Heun's method tailored to the SIQRD model
EulerBackward	Euler's backward method tailored to the SIQRD model
Solvers	Module grouping together all the above procedures and providing an explicit interface for the 3 simulation methods implemented

3 How did you store the data and parameters in your program?

The grid points on $[0, T]$ and the approximate solution in these grid points are respectively stored in a one-dimensional array of length $N + 1$ and in a two-dimensional array of shape $(N + 1) \times 5$. Since the number of grid points $N + 1$ is known at compile time, simple arrays allocated on the stack can be used. As explained earlier, these arrays are passed to the subroutines through the arguments `tk` and `x` due to memory concerns.

The rate parameters $\beta, \mu, \gamma, \alpha, \delta$ characterizing the SIQRD model are stored as module variables. The primary advantage of this approach is that they are accessible to all the procedures within module and the file `paramters.in` only needs to be read once[2], which is an expensive operation.

Finally, the parameters T and N are provided as arguments to the various subroutines.

4 How do you ensure your implementation is correct?

The SIQRD model is nonlinear and includes interactions and transitions between multiple compartments, making it challenging to find closed-form solutions for direct comparison with the results of our simulation although for some parameter choices there exists an analytic solution. However, other tests presented below can be conducted to detect potential implementation mistakes.

One useful property of the SIQRD model that we can use to check the correctness of our implementation is the conservation law about the total population. Indeed, it is clear that the following relation holds:

$$S(t) + I(t) + Q(t) + R(t) + D(t) = constant \quad \forall t$$

Thus, a first simple test to ensure that the simulations through the numerical

[2]Within the `initialize` function.

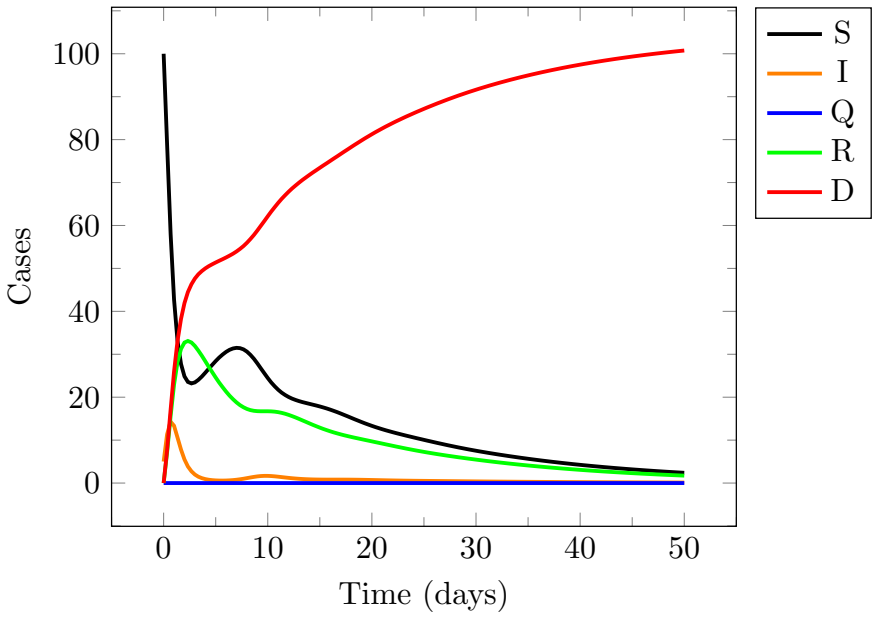

Figure 1: No measures

methods are realistic consists in verifying that

$$S(t_k) + I(t_k) + Q(t_k) + R(t_k) + D(t_k) = S_0 + I_0 \quad k = 0, \ldots, N$$

That is the case in the tested examples (up to a rounding error).

Another easy check we can make is to verify that the different methods give similar results. As we will see in question 6, this is also the case.

Finally, we can calibrate the simulation parameters and see whether the results obtained coincide with our expectations. Consider the following scenario. A highly infectious disease spreads (high β), with as many people succumbing to it as recovering from it ($\gamma = \alpha$). The disease can also be caught several times ($\mu > 0$). The two figures 1, 2 compare situations where measures are taken to isolate the infected in quarantine or not. The mortality rate drops sharply when measures are taken. In the opposite case, the entire population is almost decimated after several waves of contamination. Cleary, these simulations meet our expectations.

The figures 3, 4 show the evolution of an infectious but non-mortal disease without isolating people. We can see that if individuals can become susceptible again after being immune, then the number of infected individuals converges to a non-zero value, i.e. the disease remains in the population indefinitely. Conversely, it will disappear of its own accord if $\gamma = 0$.

5 How do you set the precision of the floating point numbers? Why?

For both questions, this is made by using the simple following lines :

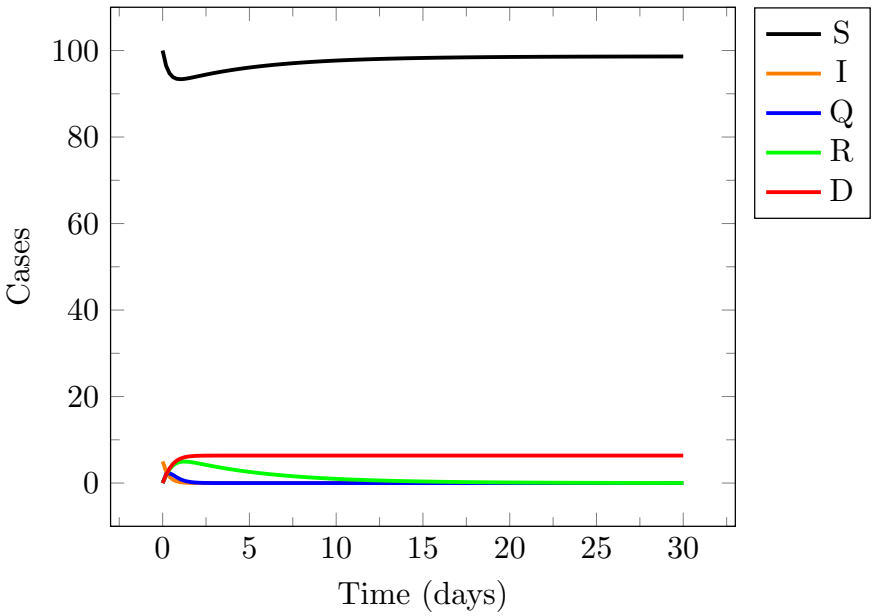

Figure 2: Lockdown

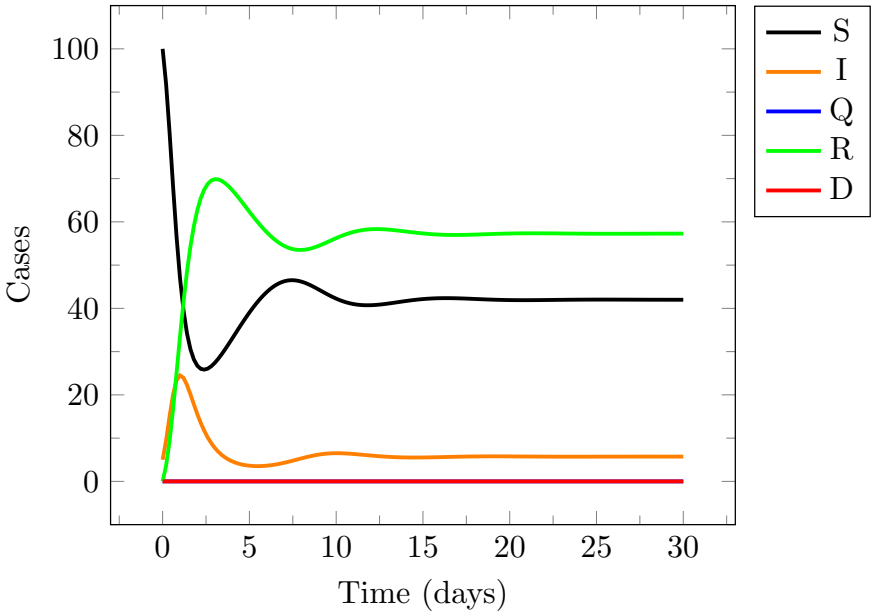

Figure 3: $\gamma > 0$

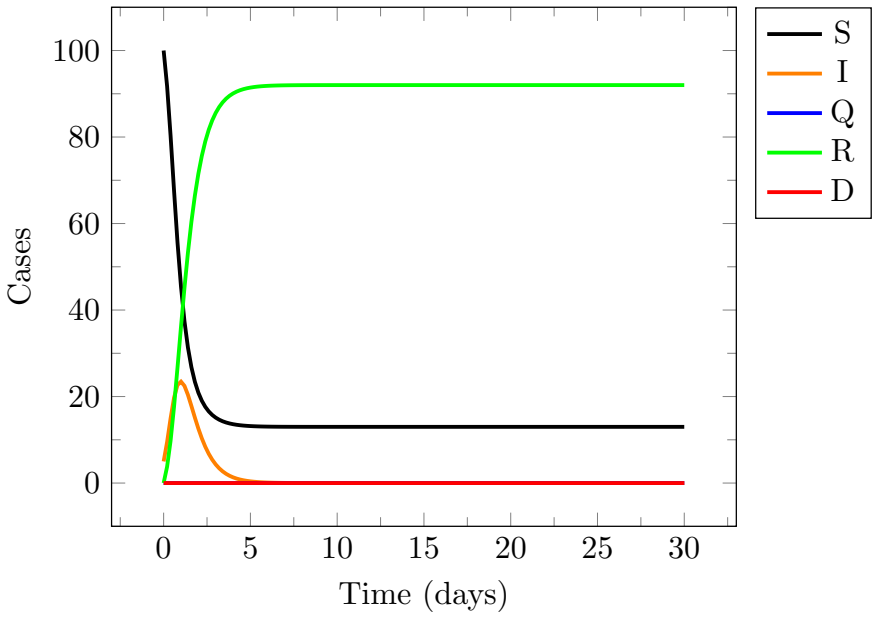

Figure 4: $\gamma = 0$

```
integer, parameter ::              &
sp = selected_real_kind(6, 37),    &
dp = selected_real_kind(15, 307), &
ep = selected_real_kind(18,4931), &
qp = selected_real_kind(33,4931), &
p = sp

real(p) :: r
r = 0._p
```

This allows the user to switch easily between respectively single, double, extended and quadruple precision as described in the *IEEE 754* standard.

The intrinsic function `selected_real_kind` is used here to obtain the kind value for the required precision in a compiler-independent way.

6 What do you see when you compare the three methods implemented for simulating the SIQRD model?

When comparing the simulations with the parameters $(\beta, \mu, \gamma, \alpha, \delta, S_0, I_0, N, T) = (10.0, 0.0, 10.0, 1.0, 0.0, 100, 5, 150, 30)$, we obtain as output the values represented in figures 5, 6, 7. One can see that the only difference between the results is the convergence values of the different variables, which are not the same while the global shape is similar. Here are the final values of each simulation computed at the time step t_N:

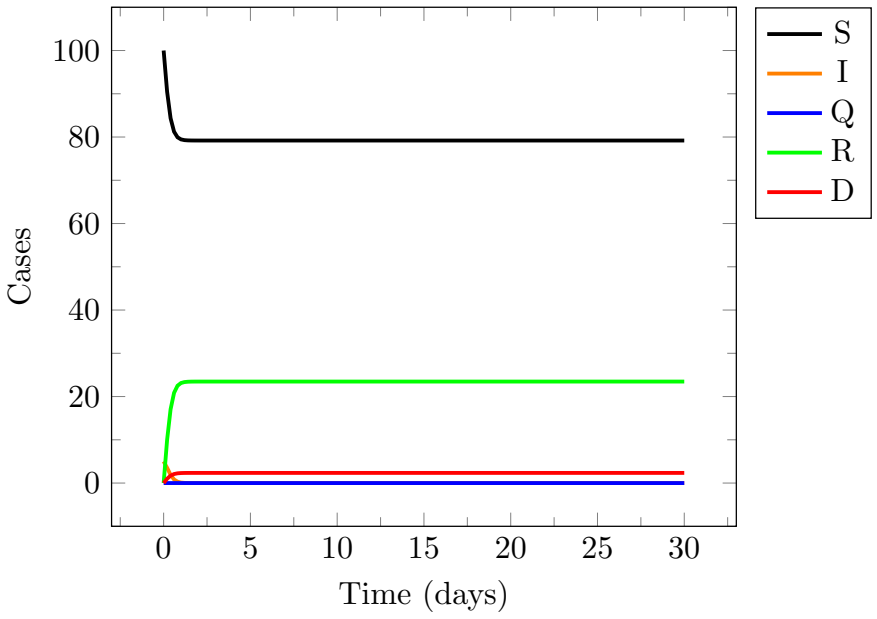

Figure 5: Visualization of the solution using Euler's forward method

Method	$S(t_N)$	$I(t_N)$	$Q(t_N)$	$R(t_N)$	$D(t_N)$
Euler Forward	79.18871	0.0	0.0	23.46480	2.34648
Heun	81.16818	0.0	0.0	21.66529	2.16653
Euler Backward	82.28303	0.0	0.0	20.65181	2.06518

The three methods therefore give different results. As Heun's method is second-order, it can be considered more accurate. Euler's methods are less precise, but have their advantages. Indeed, the euler forward method is particularly simple to implement and is computationally inexpensive, unlike the euler backward method, which requires a linear system to be solved at each iteration. This complexity does, however, add to the stability of the method. In simulations where some components of the system change much more rapidly than others, it can therefore be more accurate than the other two methods, which would require a very small time step to obtain similar results. In view of the parameters considered here, e.g. the high transmission rate β that results in quick infection spread, we're very close to such a scenario.

From a model perspective, it appears that the number infectious of people quickly reaches 0 due to the high infection and recovery rates. However, a few individuals have passed away because of the small non-zero death rate. No one has been quarantined since $\delta = 0$. Finally, the last populations are either susceptible, which means they have not contracted the desease during the simulation but are still vulnerable, or immune for life ($\mu = 0$).

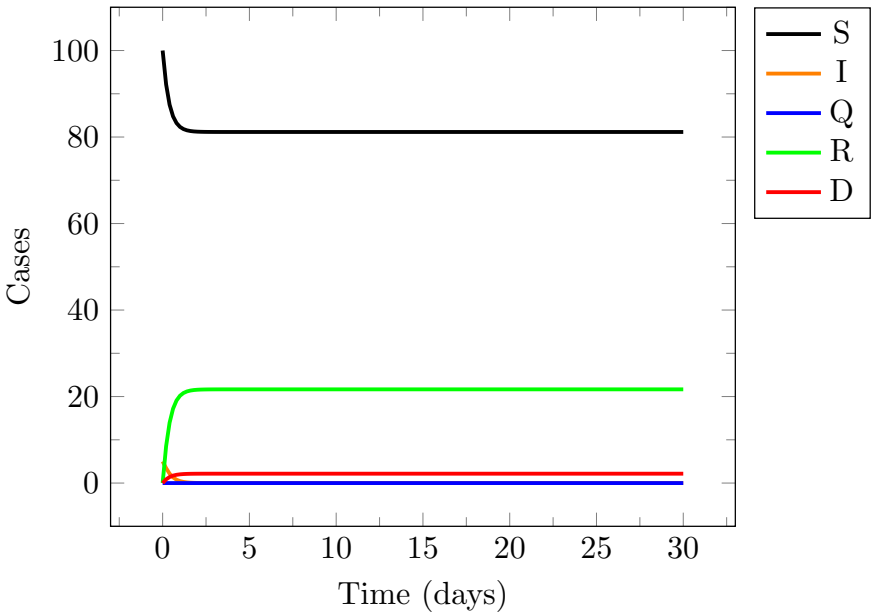

Figure 6: Visualization of the solution using Heun's method

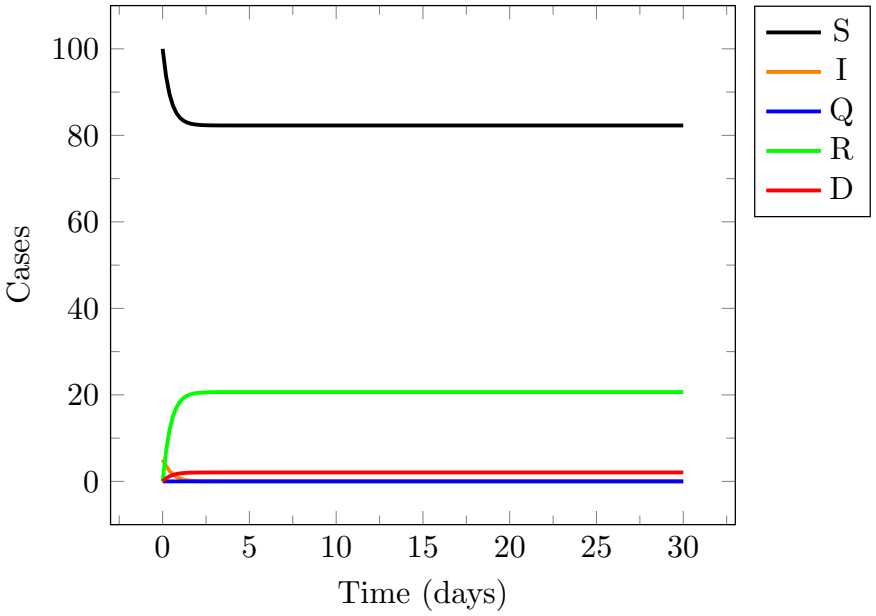

Figure 7: Visualization of the solution using Euler's backward method

7 Which stopping criteria did you choose for Newton's method? Why?

The stopping criterion based on the forward error cannot be implemented in practice since it requires knowing the exact solution of the non-linear system $x^*_{k+1} = x_k + \frac{T}{N} f(x^*_{k+1})$ which is precisely what we aim to compute by means of the Newton's method within the implementation. Indeed, the non-linear system does not have any straightforward analytical solution due to the complexity of the SIQRD model. Thus, I chose to focus on the backward error as stopping criterion.

Moreover, thanks to the implicit nature of the Euler's Backward method, it is considered backward stable, implying that it provides an accurate solution for a nearby problem. This sets it apart from the Euler's Forward method. Furthermore, assuming that the SIQRD model is coupled with realistic parameter choices, the reference book[3] assures us of the forward stability of the algorithm. In other words, the forward error divided by the condition number of the problem is guaranteed to be small.

[3] *Writing Scientific Software : a guide to good style* by S. Oliveira and D. Stewart.