# Scientific Software / Technisch-wetenschappelijke software: Assignment 1

Clémence Duplat

October 31, 2023

## 1 Question 1: Exponential growth

In this part it is asked to implement a Fortran 95 code to predict the number of infected individuals after a certain number of days using a simplified model given by :

$$I_d = (1 + r)^d * I_o \tag{1}$$

where r (the daily infection rate), $I_o$ (the number of infections on day 0) and d (the number of days) are asked to the reader in order to predict $I_d$ (the predicted number of infections after d days).

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

Exactly like we did in Session 1, we will set the precision of the floating point as follows:

```
!-----Double precision----!
integer, parameter :: dp = selected_real_kind(15, 307)
!-----Single precision----!
integer, parameter :: sp = selected_real_kind(6, 37)
!--to make it easy to switch between the 2 precision format--!
integer, parameter :: precision = dp
```

Now that the program seems correctly implemented, we will be able to focus on the different results obtained when using single(sp) and double precision(dp). As a small reminder, single precision uses 32 bits to represent a floating point number while double precision uses 64 bits. To be more precise, as seen in the the Scientific Software book, a floating point number can be represent(converted) as follows: the sign s (1 bit for double and single precision), the exponent e (8 bits for single and 11 for double precision) and the mantissa m (23 bits for single and 52 for double precision). We will apply this to different cases to see how and where it differs.

## 1.2 What do you observe when you run your first program on different compilers? Can you explain what happens based on the IEEE 754 format?

**In the first case** let's take $I_o = 100$,$d = 1500.5$ and $r = 0.01$ and compute it on the one hand in single precision and on the other hand in double precision. The answer $I_d$ differs a lot in single and double precision. To understand why, we are going to see the difference in rounding at each step of the equation. At the first step of the calculation, we can see that in single precision the sum $1.0 + 0.01$ will, contrarily to double precision, not be equal to 1.010000 but to 1.00999999. This happened because the representation of 0.01 in binary is 0.0001100110011.... And because the mantissa is 23 bits long in single precision, the last 1 is truncated (end with ...01). That's why it is rounded to a number in the floating-point number system that is the closest to 0.01. In double precision the mantissa is 52 bits long and so the last 1 is not truncated (it ends with ...011). 0.01 is correctly represented in double precision without being rounded.

**In the second case**, let's take $I_o = 100$, $d = 1500.5$ and $r = 0.009765625$. We can see that $I_d$ in single and double precision barely differs. By looking at each step we can see that it globally doesn't differ due to the fact that r can be respresented exactly in single and double precision without being rounded (due to his mantissa that is not repeated at infinity).

# 2 Question 2: Simulating SIQRD model

In this part it is asked to implement Euler's forward method, Euler's backward method and Heun's method in Fortran 95 to simulate the SIQRD model that is given to us.

*SIQRD: Susceptible, Infectious, Quarantined (infected but not infectious), Recovered (and immune) and Deceased The parameters definition are: $\beta$ the infection rate, $\mu$ the rate at which immune people become again susceptible, $\gamma$ the recovery rate, $\delta$ the rate at which infected people get tested and are quarantined and $\alpha$ the death rate.*

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

To make my code easier to compile,it contains subroutines, modules and functions.
Here is an overview of the structures of my **functions**:

```
function S_der(S,I,R) result(result_value)
        !-----Function to calculate S(t)-------!
        !----Same structure for the other SIQRD functions----!
        real(KREAL), intent(in):: I,S,R
        real(KREAL) :: result_value
```

*We have the same structure as above for all the derivatives of my SIQRD model*

```
function identity_matrix(n) result(mat)
    !--------Function to calculate an identity matrix of size n------!
    integer, intent(in) :: n
    real(KREAL), dimension(n, n) :: mat

function calculate_jacobian(S, I, R, Q, D) result(Jacobian)
    !-------Function to calculate the Jacobian of my SIQRD model---------!
    real(KREAL):: Jacobian(5, 5)
    real(KREAL), intent(in) :: S, I, R, Q, D
```

All these functions are put into a **module** name Derivatives

```
module Derivatives
```

For each programme (see *README*) I implemented a corresponding subroutine to make it easier.
Here is an overview of the structures of my **subroutines**:

```
subroutine forward_euler(dt, x_0,f, x)
    !----------Forward euler formula to compute approximation-------------!
    real(KREAL), intent(in) :: dt,x_0,f
    real(KREAL),intent(out) :: x

subroutine heun(dt,x_0,f,f_k,x)
    !--------------Heun formula to compute approximation------------------!
    real(KREAL), intent(in) :: dt,x_0,f,f_k
    real(KREAL),intent(out) :: x

subroutine backward_euler(dt, x_0, f, x, max_iterations, tolerance)
    real(KREAL), intent(in) :: dt, tolerance
    integer, intent(in) :: max_iterations
    real(KREAL), dimension(5),intent(in) :: x_0,f
    real(KREAL), dimension(5),intent(out) :: x
```

In addititional, I have one **module** for declaring some parameters. The choice of making a module Parametres is principally to have an easier access to the parameters used in the SIQRD model ($\beta$,$\mu$,$\gamma$,$\delta$,$\alpha$) without having to repeat them when defining my functions. This module is note necessary but it is to avoid redundancy.

```
module Parametres
```

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

I stored my data into allocatable arrays to be able to store each iteration of my methods for the N+1 grid points. All my variables have a single precision.

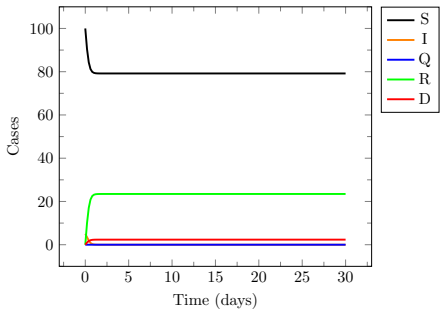
## 2.3 How do you ensure your implementation is correct?

To ensure that my implementation is correct, I will use the properties of the model. Indeed, as advised in the assignment, a good parameter combination can check if my methods are correct or not. By testing viable models and change the values of the parameters, we can conclude if our methods are correct or not. For example, by defining only S and I to a random value and Q,R and D to zero, the sum of the population should remain constant at the long term. We will also see that some methods convergences faster than others.

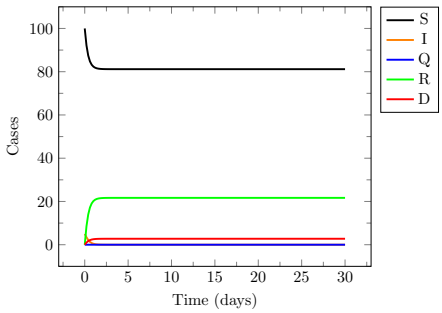
## 2.4 What do you see when you compare the three methods implemented for simulating the SIQRD model?

To compare my models with the given parameters in the assignement, we will exctract the data and plot them. On the next figure you can see the difference for the differents methods. First of all, we see that the sum S+I+Q+R+D

**Comparaison methods for** $\beta = \gamma = 10.0, \alpha = 1.0, \mu = \delta = 0.0, N = 150, T = 30, S_o = 100$ **and** $I_o = 5$

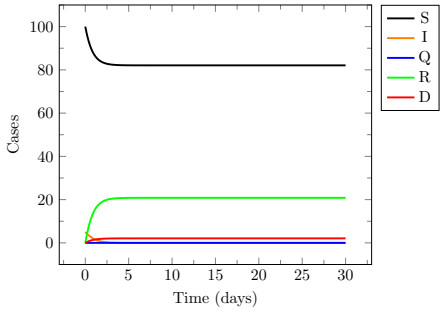



Euler Forward

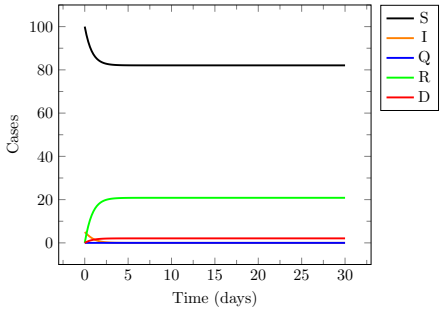



Heun

Backward euler method :




Forward error




Backward error

remains constant over time(it has been checked numerically too), so my methods

are correctly implemented. The methods don't not differ a lot, unless the S and R curve that converges at different values for Euler Forward and Heun. We see that Heun and Euler backward are quiet similar.

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

For backward euler method, their exist 2 different stopping criteria. The first one is the **forward error** ($||x^s_{k+1} - x^*_{k+1}||$ ) and the second one is the **backward error** ($||x_k + dt * f(x^s_{k+1}) - x^s_{k+1}||$). We repeat the iterative method untill one of the criteria's is sufficiently small. The choice depends on the specific problem and the convergence behavior of the function. In our case, we don't see a big difference on the plots between the 2 criterias. By looking closer at the implementation of backward euler, we can see that the backward euler method with forward error converges faster then with the backward error. In this case, it is better to use forward error but using forward error is not a bad decision.