Scientific Software / Technisch-wetenschappelijke software: Assignment 2

Clémence Duplat

November 27, 2023

1 What changes did you make to your code after feedback? Do you wish to comment on your previous report?

Feedback:
Your assignment is very good
Everything works as intended and your report is good.
However, the design of your code is a bit strange.
A much better design would have been:
- a module containing all the ode solvers
- a program which reads the input, calls the ode solvers of your choice and then outputs the results. As it is now, each program has to read the input and output the result.
Try to avoid unnecessary duplication of code.

This time, I rearranged my code to avoid code duplication. Therefor I added a subroutine for each method that has this form:

```
subroutine solver_forward(S,I,Q,R,D,S_1,I_1,Q_1,R_1,D_1)
        real(KREAL), intent(in) :: S,I,Q,R,D
        real(KREAL), intent(out) :: S_1,I_1,Q_1,R_1,D_1
        real(KREAL), dimension(5) :: fk

        !-----Calculate the derivatives of SIQRD in f_k------!
        call calculate_derivatives(S, I, Q, R, D, fk)

        !-----Call Forward Euler on each method------!
        call forward_euler(dt, S, fk(1), S_1)
        call forward_euler(dt, I, fk(2), I_1)
        call forward_euler(dt, Q, fk(3), Q_1)
        call forward_euler(dt, R, fk(4), R_1)
        call forward_euler(dt, D, fk(5), D_1)
        end subroutine solver_forward
```

Each subroutine used for each method (ex: *solverforward and forwerdeuler*) is contained in a same module. My main program (named *callSolver*) will then initiate my parameters, allocate my arrays and call the desired method. This way there is no duplication of code anymore. When launching my program, the answers for each method are saved. In the module *param* I added subroutines that are used to initialize the parameters and my code is more clear. This way I only have to call the subroutine desired to initialize my parameters. In addition to this, I added a way more efficient stopping criterium in my Backward Euler method that will save us a lot of compile time in the next programs.

```
norm2(b)<tolerance*norm2(x)
```

Effectively, this helps us to take into account the scale of the solution based on the residuals of the solution. It checks if the norm(b), which indicate how close we are from the current solution, is smaller than $tolerance * norm(x)$, which is the acceptable error related to the size of the solution.

2 Was it easy to change the precision? Why (not)? Would you do anything different?

It was easy to change precision because I declared a variable in the beginning:

```
integer, parameter :: KREAL = selected_real_kind(6, 37)
```

If I want to change and go to double precision I should only change this variable like that:

```
integer, parameter :: KREAL = selected_real_kind(15, 307)
```

To make it more easy to go from single precision to double precision, I could do it like that:

```
integer, parameter :: sp = selected_real_kind(6, 37)
integer, parameter :: dp = selected_real_kind(15, 307)
integer, parameter :: KREAL = sp !or dp
```

3 What changes did you have to make to your code in order to accommodate that N and T are no longer known at compile time? What kind of memory are you using? What are the disadvantages?

In my previous code, I already did a dynamic allocation of my arrays based on N. This time, we asked the user to give a specific N and T and I didn't had to make major changes in my code.
The memory I used is *heap memory* because I used alloctable arrays. This type of memory is used for variables whose size is not known at compile time and can change during compile time, which is the case here.
Using this type of memory can be a **disadvantage**. Effectively, it can lead to memory leaks if we don't deallocate correctly the arrays after we used them. We can check if the allocation was done successfully.

4 How did you implement the solver?

To implement our own module *solver* in single and double precision, it is suggested to use the SGESV and DGESV of LAPACK library. To be able to handle both single and double precision without recompilation, we are going to use a generic interfaces. Effectively, we know that "If two or more generic interfaces that are accessible in a scoping unit have the same local name, they are interpreted as a single generic interface."
Therefore, in my module solver, I added an interface :

```
interface solve
    module procedure solve_single, solve_double
end interface solve
```

This way, the module solver will contain two subroutines *solvesingle* and *solvedouble* and when the module *solve* is called, it will automatically detect the right precision and call the subroutine attached to this precision.
This is a huge gain of time if we use them a lot.

5 Which convergence criterion did you use to solve (2)? Why?

When implementing the *tweaking* code, we used a Newton's method to solve the the problem

$$F(\beta) = target$$

with $F(\beta)$ the maximum value of the sum $I_k + Q_k$ over all k. We also took into account that the sum could not be bigger than the total population.
We computed the derivative of the function using a finite difference approximation and update the beta using Newton iteration. We iterate over the maximum iteration until a stopping criteria is met.
To choose the right stopping criteria, it requires to find a balance between accuracy and computational efficiency. In my case when implementing my *tweaking* program, I decided to choose for the next stopping criteria:

```
if (abs(F_beta - target) < tol*abs(target) then
    exit
endif
```

One part verify if I reached a Fbeta that is in proximity to the target. It is made in relative to the magnitude of the target, which allow our criterion to be scale independent.

If this works very well for Forward Euler and Heun method, we have to add an additional part of code for Backward Euler method. This happens because for Backward Euler, the derivative can come close to 0, producing a huge upadate step for the beta.
I tried to change this by bounding my update step but after hours and hours of searching I didn't manage to come to a right solution for this method.
If you have any indications on where I could go wrong, I would be very pleased.

6 How did you perform your timings in order to ensure that they are reliable?

To perform my timing I used an additional module *timing*. These module contains 1 subroutine and 1 function: *tic()* and *toc()*. One will call the CPU timer and thus start the timer. The other will calculate the elapsed time.
We choose the CPU timer, that refers to the time for wich the CPU was used fpr processing our program, instead of a Wallclock timer, that measures the real ellapsed time from start to finish. It is way better to use the CPU time, because we wan't to know the analyze the performance of my program.

7 Place your three tables for forward Euler, backward Euler and Heun here. What do you observe? Explain any unexpected results. Which algorithm do you think is the best? Why?

I used the python library to implement a code that calculate in a 'row manner' the optimal beta wich occurs to be 0.38. I will use this result to be sure my *tweaking* program works in the right way.

$\Delta\beta = 10^{-i}$	Optimal β	time	Newton iterations
i=1	0.388766193471475	0.021702000000000	56
i=2	0.388766193041365	0.006709000000000	17
i=3	0.388766192946014	0.002547000000000	8
i=4	0.388766193261989	0.001914000000000	6
i=5	0.388766192581053	0.001879000000000	6
i=6	0.388766192567512	0.001873000000000	6
i=7	0.388766192566609	0.001870000000000	6
i=8	0.388766192566523	0.001871000000000	6
i=9	0.388766192566515	0.002025000000000	6
i=10	0.388766192566518	0.001911000000000	6
i=11	0.388766192566588	0.002397000000000	6
i=12	0.388766192566427	0.001879000000000	6
i=13	0.388766192559539	0.001876000000000	6
i=14	0.388766192547318	0.001860000000000	6
i=15	0.388766192705570	0.001875000000000	6
i=16	0.388766192789084	0.002850000000000	9

Table 1: Euler Forward

$\Delta\beta = 10^{-i}$	Optimal β	time	Newton iterations
i=1	0.388424214230609	0.028937000000000	56
i=2	0.388424215732702	0.008579000000000	17
i=3	0.388424216034476	0.003384000000000	7
i=4	0.388424207012546	0.002864000000000	6
i=5	0.388424206124580	0.002882000000000	6
i=6	0.388424206104614	0.002834000000000	6
i=7	0.388424206103196	0.002838000000000	6
i=8	0.388424206103060	0.002867000000000	6
i=9	0.388424206103048	0.002865000000000	6
i=10	0.388424206103043	0.002885000000000	6
i=11	0.388424206103063	0.002866000000000	6
i=12	0.388424206102045	0.002867000000000	6
i=13	0.388424206103798	0.002864000000000	6
i=14	0.388424205891547	0.002864000000000	6
i=15	0.388424207309071	0.002862000000000	6
i=16	0.388424211142499	0.004931000000000	10

Table 2: Heun

We observe that we converge to an **optimal** *beta*for each $\Delta\beta$. We know that when $\Delta\beta$ decrease the accuracy of the derivative improve and we will have a more precise calculation. Here we don't see a significant difference for this.
When the $\Delta\beta$ start to decrease the **Newton iterations** remains almost constant, unless for $i = 16$ where it increase. This might be due because $\Delta\beta$ is too small.
The **compuational time** decreases in the beginning when $\Delta\beta$ becomes smaller, but at a cretain level of $\Delta\beta$ it does not chnage enormous.
We can see that for the Forward Euler method and the Heun method, the optimization algorithm give almost the same results.
Therefore, I think the **best method to chose is the Forward Euler method**. Knowing both heun and forward euler methods work the same way it is not especially necessary to use a second order method. We also know that the Backward Euler method can be very sensitive to small perturbations.

It is simplier to use the explicit method like Forward Euler because the problem is not stiff but in another case it is usefull to use an implicit more stable method like Heun or Backward Euler.

7.1 Error for Backward Euler

Like said before, I encountered a lot of problems for my program when using Backward Euler method. I looked hours and hours to try and find a solution but it never found a right solution. Here is the table with an updating step controlled manually. The stopping criteria is never met and it take a lot of CPU time. This answer is not correct and it is maybe due to the Backward Euler, but I tested it and it was correct, or to the calculation of the Jacobian, which is normally also correct. I have an optimal beta of 0.39 but the loop encounter the maximum iteration of 1000 wich is way too much.

8 What effect do the optimization flags have? Do you always want to use optimization? Are there other flags which may be useful when compiling your code?

As for the exercice session we will try the flag combinations -Og, -O2, -O3 and -Ofast.
We know that optimization flags plays an important role in how the compiler optimizes the algorithm, so we will try and explain all of them.

- -0g : In our case we see that the CPU time is way faster
- -02, -03 : "Increases optimization levels, the higher the number, the more optimization is done." A higher number can require additional compilation time but reduce execution time. We can see that the CPU time with -03 flag decrease.
- -0fast : "Enables a range of optimizations that provide faster, though sometimes less precise, mathematical operations"

When I first tried to debug my code, I did not use optimization flags but when your are certain that your solution is correct, it might be usefull to use them. Effectively, it helps maximize the performance. We test them because some of them can introduce some bugs.

9 What do you see when you run your code under valgrind?

```
==558969==
==558969== HEAP SUMMARY:
==558969==     in use at exit: 0 bytes in 0 blocks
==558969==   total heap usage: 48 allocs, 48 frees, 239,972 bytes allocated
==558969==
==558969== All heap blocks were freed -- no leaks are possible
==558969==
==558969== For lists of detected and suppressed errors, rerun with: -s
==558969== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

We can interpret the results as follows:

- We have no memory leaks, everything has been deallocated correctly
- We allocated and freed 48 times, giving a total amount of memory allocated of 239,792 bytes
- We have no errors in our code (this doesn't mean that it worked correctly, it just worked)

10 Did you test on different compilers? What changes do/did you have to make?

In addition to the *gfortran* compiler, we can test our code on *nagfor* and *ifort*.

11 Which LAPACK routine did you select for the eigenvalue computation? Which parameters of this subroutine do you use? Which ones do you not use? Why?

For the computation of my eigenvalues, we used the LAPACK routine DGEEV. It computes for an N-by-N real nonsymmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors. Here you can see all the parameters of this subroutine

```
subroutine dgeev ( character  JOBVL,
                   character  JOBVR,
                   integer  N,
                   double precision, dimension( lda, * )  A,
                   integer  LDA,
                   double precision, dimension( * )  WR,
                   double precision, dimension( * )  WI,
                   double precision, dimension( ldvl, * )  VL,
                   integer  LDVL,
                   double precision, dimension( ldvr, * )  VR,
                   integer  LDVR,
                   double precision, dimension( * )  WORK,
                   integer  LWORK,
                   integer  INFO )
```

In our case, we only need to compute the real part of the eigenvalues of the Jacobian to discuss the stability. Indeed, the real part of each eigenvalue needs to be smaller than or equal to zero to ensure the stability. It then states that small perturbations will decay over time and that it will return to it's equilibrium point.
Therefore, we do not need all the parameters of the subroutine.
The next parameters are required:JOBVL and JOBVR (but set to 'N' to indicate that the left/right eigenvectors doesn't have to be computed), N (size of matrix),A(matrix), LDA(leading dimension), WR(store real part), WI(store imaginary part), WORK(workspace array), LWORK(dimension of WORK), INFO(status about the execution).

12 What are the eigenvalues that you compute?

For the first case, we export the parameters from *eigenvalues1.in*. The real part of the eigenvalues computed are:

```
-0.0000000000000000
0.0000000000000000
-0.0000000000000000
-0.25000000000000000
-5.0000000000000044E-002
```

The system is thus **stable** because there are no strict positive real part.
For the second case, we export the parameters from *eigenvalues2.in*. The real part of the eigenvalues computed are:

```
 -0.0000000000000000
0.0000000000000000
```

```
-0.0000000000000000
-0.25000000000000000
0.14999999999999997
```

The system is **unstable** because the last eigenvalue has a positive real part. This lead to a grow of perturbation.