Scientific Software / Technisch-wetenschappelijke software: Assignment 2

Victor Lepère

November 18, 2023

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

- The first change concerns the backward error stopping criterion within the `EulerBackward` method. We now take into account the order of magnitude of the estimated solution $x_{k+1}^{(s)}$, which could lead to ill convergence when working with extremely small or large number. The absolute error check $\|x_k + \frac{T}{N} f(x_{k+1}^{(s)}) - x_{k+1}^{(s)}\| < \epsilon$ for a given tolerance ϵ becomes a relative error check $\|x_k + \frac{T}{N} f(x_{k+1}^{(s)}) - x_{k+1}^{(s)}\| < \epsilon_{mach} \|x_{k+1}^{(s)}\|$.
- Read and write operations on files have also been extracted from the simulation methods to enhance reusability. In addition to the modularity improvement, it appears that these file operations are highly time-consuming compared to the execution time of the methods themselves. Therefore, they act as a performance bottleneck for the methods. For instance, Newton's method, which calls `EulerForward` multiple times to simulate the model used in the next question, sped up by a factor 37 thanks to this slight improvement.
- A maximum number of iterations for Netwon's method within the `EulerBackward` method has also been added to prevent infinite loops in case of potential non-convergence of the algorithm. By default, the bound is set to 100 but can be changed through the optional argument `nmax_opt`.

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

Changing the precision was relatively straightforward, as all one had to do was modify the value of `wp` to `sp`, `dp`, ... in the code snippet below, located at the beginning of the implementation.

```
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), &
wp = dp
```

Since all variables are declared via `real(wp)` and all literals via `._wp`, the change in precision is applied directly throughout the program.

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?

Dynamic arrays allocated on the heap are now required. This is done by including the `allocatable` keywords within the arrays declarations and by using the intrinsic function `allocate` (resp.

`deallocate`) to allocate (resp. deallocate) memory once the array dimensions have been determined.
The most elementary inconvenience is naturally the increasing syntax complexity of the code. Moreover, the memory allocation on the heap lowers the program speed due to the slowness of the process. Despite the fact that both stack and heap segments are located in the *Random Access Memory*, the memory allocation is not performed in the same way. Indeed, the stack is a downward-growing linear data structure, and allocating memory on the stack simply involves decrementing a stack pointer, which is a fast operation. The heap is a more dynamic data structure and requires a memory management algorithm to find a free contiguous block of memory that is large enough to satisfy the memory allocation request. This process can be slow especially if the heap is subject to fragmentation.
Finally, the use of dynamic arrays makes the program prone to memory leaks. It may happen if the programmer forget to deallocate an allocated resource.

4 How did you implement the solver?

The implementation of the `solve` function is made by means of an interface :

```
interface solve
    module procedure solve_sp
    module procedure solve_dp
end interface
```

The abstraction mechanism above is known as *procedure overloading* and enables to handle the computation in both single and double precision depending on the real kinds of the arguments provided by the user to the `solve` subroutine.
The main advantage of this trick is that the user can call the same procedure while switching from one floating-point precision to another. Thus, if the given matrix `A` and vector `bx` are single (resp. double) precision arrays at procedure call, the subroutine `solve_sp` (resp. `solve_dp`) is used to solve the system through the LAPACK subroutine `sgesv` (resp. `dgesv`).

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

To solve this problem, the following stopping criterion is used :

$$|\beta_{n+1} - \beta_n| < \epsilon_{mach}|\beta_{n+1}| \tag{1}$$

$$\iff \left|\frac{\text{target} - F(\beta_n)}{\frac{d}{d\beta}F(\beta_n)}\right| < \epsilon_{mach}|\beta_{n+1}| \tag{2}$$

$$\iff \left|\frac{\Delta\beta}{1 - \frac{F(\beta_n+\Delta\beta)-\text{target}}{F(\beta_n)-\text{target}}}\right| < \epsilon_{mach}|\beta_{n+1}| \tag{3}$$

Again, the relative error above considers the scale of the β rate although it is less necessary since the range of realistic values for this parameter is quite limited.
Empirical experiments have shown that this criterion leads to the smallest error $E \coloneqq |F(\beta^*) - \text{target}|$ in comparison with other criteria like $|F(\beta_{n+1}) - \text{target}| < \epsilon|F(\beta_{n+1})|$.

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

To ensure compiler independance of timings, the standard Fortran intrinsic function `cpu_time` is preferred over `system_clock`, which is implemented differently depending on the compiler. Moreover, measuring the time consumed by processes on the CPU is more pertinent in this context than measuring real elapsed time, as the latter can be affected by other external tasks

running on the machine.
Taking the averaged execution time computed over multiple executions can enhance the robustness of time measurements. This approach is more expensive but reduces the variance of the results. Other statistics, such as the median, which is less susceptible to outliers, and the variance, are also valuable to compute for a more precise understanding of the code's temporal behavior.

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?

$\Delta\beta$	β^*	time $[ms]$	iterations
10^{-1}	$3.886827183531352 \cdot 10^{-1}$	32.58	129
10^{-2}	$3.886827183531351 \cdot 10^{-1}$	4.54	37
10^{-3}	$3.886827183531351 \cdot 10^{-1}$	1.66	13
10^{-4}	$3.886827183531351 \cdot 10^{-1}$	1.19	9
10^{-5}	$3.886827183531351 \cdot 10^{-1}$	0.97	7
10^{-6}	$3.886827183531351 \cdot 10^{-1}$	0.95	7
10^{-7}	$3.886827183531351 \cdot 10^{-1}$	0.83	6
10^{-8}	$3.886827183531351 \cdot 10^{-1}$	0.83	6
10^{-9}	$3.886827183531351 \cdot 10^{-1}$	0.84	6
10^{-10}	$3.886827183531351 \cdot 10^{-1}$	0.85	6
10^{-11}	$3.886827183531351 \cdot 10^{-1}$	0.83	6
10^{-12}	$3.886827183531351 \cdot 10^{-1}$	0.83	6
10^{-13}	$3.886827183531351 \cdot 10^{-1}$	0.83	6
10^{-14}	$3.886827183531351 \cdot 10^{-1}$	1.07	8
10^{-15}	$3.886827183531351 \cdot 10^{-1}$	1.20	9
10^{-16}	$3.886827183531351 \cdot 10^{-1}$	2.39	19

Table 1: Forward Euler

$\Delta\beta$	β^*	time $[ms]$	iterations
10^{-1}	$3.883414205155445 \cdot 10^{-1}$	55.16	126
10^{-2}	$3.883414205155444 \cdot 10^{-1}$	11.81	37
10^{-3}	$3.883414205155444 \cdot 10^{-1}$	4.34	13
10^{-4}	$3.883414205155444 \cdot 10^{-1}$	3.11	9
10^{-5}	$3.883414205155444 \cdot 10^{-1}$	2.48	7
10^{-6}	$3.883414205155444 \cdot 10^{-1}$	2.47	7
10^{-7}	$3.883414205155444 \cdot 10^{-1}$	2.19	6
10^{-8}	$3.883414205155444 \cdot 10^{-1}$	2.15	6
10^{-9}	$3.883414205155444 \cdot 10^{-1}$	2.51	7
10^{-10}	$3.883414205155444 \cdot 10^{-1}$	2.17	6
10^{-11}	$3.883414205155444 \cdot 10^{-1}$	2.17	6
10^{-12}	$3.883414205155444 \cdot 10^{-1}$	2.17	6
10^{-13}	$3.883414205155444 \cdot 10^{-1}$	2.49	7
10^{-14}	$3.883414205155444 \cdot 10^{-1}$	2.82	8
10^{-15}	$3.883414205155444 \cdot 10^{-1}$	3.43	10
10^{-16}	$3.883414205155444 \cdot 10^{-1}$	6.51	20

Table 2: Heun

$\Delta\beta$	β^*	time $[ms]$	iterations
10^{-01}	$3.891969040783693 \cdot 10^{-1}$	7883.38	126
10^{-02}	$3.892346678900045 \cdot 10^{-1}$	2056.99	38
10^{-03}	$3.892346678900045 \cdot 10^{-1}$	770.59	14
10^{-04}	$3.892346678900045 \cdot 10^{-1}$	513.86	9
10^{-05}	$3.892346678900045 \cdot 10^{-1}$	464.04	8
10^{-06}	$3.892346678900045 \cdot 10^{-1}$	466.44	8
10^{-07}	$3.892346678900045 \cdot 10^{-1}$	517.71	9
10^{-08}	$3.892346678900045 \cdot 10^{-1}$	414.71	7
10^{-09}	$3.892346678900045 \cdot 10^{-1}$	415.78	7
10^{-10}	$3.892346678900045 \cdot 10^{-1}$	416.45	7
10^{-11}	$3.892346678900045 \cdot 10^{-1}$	412.89	7
10^{-12}	$3.892346678900045 \cdot 10^{-1}$	415.34	7
10^{-13}	$3.892346678900045 \cdot 10^{-1}$	416.26	7
10^{-14}	$3.892346678900045 \cdot 10^{-1}$	519.73	9
10^{-15}	$3.892346678900045 \cdot 10^{-1}$	571.89	10
10^{-16}	$3.892346678900045 \cdot 10^{-1}$	977.29	18

Table 3: Backward Euler

Several observations can be made about the above tables. Firstly, Newton's method with different simulation methods requires a similar number of iterations to converge as a function of $\Delta\beta$. This is because the stopping criterion (1) focuses on the evolution of the iterated parameter β rather than the function to be cancelled, which is more influenced by the simulation method. Additionally, the development of the criterion (3) shows that evaluations of F only occur in the form of a proportional ratio.
Secondly, The optimal β^* calculated in the three tables is identical for $\Delta\beta \leq 10^{-2}$, regardless of the $\Delta\beta$ value used to approximate the derivative. This is because, for a sufficiently accurate approximation, the number of significant digits doubles at each iteration, thanks to Newton's method's quadratic convergence.
Another observation is that the number of iterations tends to increase for very small $\Delta\beta$ values. The smaller $\Delta\beta$ is, the more accurate the approximation is. However, the next iterate in the Newton method is chosen by following the tangent of slope $\frac{d}{d\beta}F$ and its intersection with the β-axis. Thus, the smaller $\Delta\beta$ is, the more the choice of the next iterate will be influenced by the small fluctuations of the objective function that are possibly decorrelated with the global behavior of the function, slowing down convergence.

Finally, the execution time[1] is the most significant difference between the three methods. The backward Euler's method is the most time-consuming. For this set of parameters, Heun's method seems to be an optimal choice, as it offers a good compromise between precision[2] and speed. Note, however, that despite its slowness, the Euler Backward method is stable for a larger choice of parameters. So, depending on the eigenvalues of the Jacobian and the time step used, the latter may be a better choice depending on the parameter set.

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?

- `-O1` : For all three methods, the number of iterations and the optimal β's remain unchanged. Only the execution time decreases by 15 to 20 % except when $\Delta\beta = 0.1$ with the forward Euler and Heun methods. In this case, the execution time increases by 40 to 70 %.

[1]The times shown in the tables are average times calculated over 10 executions
[2]Recall that Heun's method is of the second order.

- `-Og` : This optimization flag also reduces execution time by a factor slightly below that achieved with `-O1` and additionally improves the debugging experience.
- `-O2` : While the `-O1` and `-Og` flags add $\sim 0.1s$ to the wall compilation time, `-O2` doubles it by adding $\sim 0.25s$. This time, the execution time decreases this time by an average of 60 %. The timings for $\Delta\beta = 0.1$ are still longer but closer to the non-optimized values.
- `-O3` : It produces results very similar to `-O2` with timings even slightly shorter.
- `-Ofast -march=native` : The recorded timings are slightly below those obtained with `-O3` at the expense of disregarding strict standards compliance and portability. This optimization flags change the number of iterations to converge especially for small values of $\Delta\beta$.

It is generally recommended to use optimizations only once the code is in its final version because optimization mechanisms tend to increase compilation time and reduce the ability of the code to be debugged, which can be counterproductive when the code is still under development.
Many useful flags are available. For example[3], the `-Wall` flag generates warnings for all potential problems in the code while the `-g` flag produces debugging information. In the third exercise session, we also discovered the `-fcheck=bounds` flag which prevents out-of-bounds indexing when working with arrays.

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

The terminal output is shown below.

```
==2267743== Memcheck, a memory error detector
==2267743== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2267743== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2267743== Command: ./q3.out 1000 100 1e-6 20000
==2267743==
==2267743==
==2267743== HEAP SUMMARY:
==2267743==     in use at exit: 0 bytes in 0 blocks
==2267743==   total heap usage: 64,057 allocs, 64,057 frees, 2,662,544 bytes allocated
==2267743==
==2267743== All heap blocks were freed -- no leaks are possible
==2267743==
==2267743== For lists of detected and suppressed errors, rerun with: -s
==2267743== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The summary indicates that the number of memory blocks allocated on the heap (= 64 057) is equal to the number of freed blocks, i.e. there is no memory leak to be concerned about. Less than 3 MB are allocated during the process, thus the program consumes a quite low quantity of memory with respect to the standard RAM size of modern computers. The *error summary* further confirms that no runtime errors, such as invalid read/write operations, occured during program execution.

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

The only change made to the code involves the `exit` intrinsic subroutine which is not recognized by the `nagfor` compiler. To fix this issue, the following update is sufficient :

```
call exit(STATUS)
! becomes
stop STATUS
```

[3] Considering `gfortran`.

Indeed, the `stop` statement is known by all the compilers and produces a similar effect.
To enhance project organization, the `Makefile` has been written so that the module files are located in the same subfolder as the source files. Compilers provide options that allow achieving such structuring. However, these options may vary from one code to another. Thus, the variables `MOD_FLAG_IN` and `MOD_FLAG_OUT` specify to the compiler where to respectively look for and place the module files according to their documentation.

```
ifeq ($(FC),gfortran)
MOD_FLAG_OUT := -J$(SRC)
else ifeq ($(FC),ifort)
    MOD_FLAG_OUT := -module $(SRC)
else ifeq ($(FC),nagfor)
    MOD_FLAG_OUT := -mdir $(SRC)
else ifeq ($(FC),g95)
    MOD_FLAG_OUT := -fmod=$(SRC)
endif

ifeq ($(FC),gfortran)
    MOD_FLAG_IN := -I$(SRC)
else ifeq ($(FC),ifort)
    MOD_FLAG_IN := -I$(SRC)
else ifeq ($(FC),nagfor)
    MOD_FLAG_IN := -I $(SRC)
else ifeq ($(FC),g95)
    MOD_FLAG_IN := -I$(SRC)
endif
```

Finally, note that `g25` is not installed on the departmental computers. Since students are not in the sudoers file, the `g25` compiler has not been tested.

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?

The used LAPACK routines are `sgeev` for single precision and `dgeev` which compute the eigenvalues and, optionally, the left and/or right eigenvectors of a real nonsymmetric matrix. The subroutines feature the following parameters :

```
subroutine dgeev(JOBVL, JOBVR, N, A, LDA, WR, WI, VL, LDVL, VR, LDVR, WORK, LWORK, INFO)
```

As we only want to verify the stability of the system with respect to the equilibrium point and not compute the perturbation values over time, it is not necessary to compute the eigenvector of the jacobian matrix. This is communicated to the routine by setting up `JOBVL` and `JOBVR` to `'N'`. In this case, dummy values[4] can be given to `VL, LDVL, VR, LDVR` since they will not be referenced within the routine.
`A`, `N`, `LDA` are respectively used to provide the jacobian matrix, its order (= 5) and its leading dimension (= 5 too given that the matrix is not a submatrix) while `WR` and `WI` store the real and imaginary parts of the computed eigenvalues. In our case, only `WR` is relevant for the analysis.
Finally, `WORK` is a work array of size `LWORK` required for internal calculations that must be allocated by the user. To know the optimal value to be passed to `LWORK`, we introduce a workspace query by calling a first time `dgeev` with `LWORK = -1`. After the subroutine execution, the value of `LWORK` which optimizes performances and memory consumption is contained in `WORK(1)`. The integer `INFO` allows us to check whether any error occurred during the call of the LAPACK routine and to get a detailed feedback according to its value and the documentation.

12 What are the eigenvalues that you compute?

	λ_1	λ_2	λ_3	λ_4	λ_5	stability
`eigenvalues1.in`	0	0	0	-0.25	-0.05	*stable*
`eigenvalues2.in`	0	0	0	-0.25	0.15	*unstable*

[4] To avoid errors, keep `LDVL`, `LDVR` > 0 nevertheless.