Computational Physics

Exercise Sheet 00

Editha Jasniewicz, Rene-Marcel Lehner

SuSe 2023

ASSIGNMENT N°

00

Folder Structure

"*Exercise-specific files are highlighted in each exercise.*"

```
1A_00_Jasniewicz-Lehner.zip
├── E1
│   ├── hello_world
│   └── hello_world.cpp
├── E2
│   ├── a.cpp
│   ├── a.hpp
│   ├── b.cpp
│   ├── b.hpp
│   ├── c.cpp
│   ├── c.hpp
│   ├── makefile
│   ├── plotting.py
│   └── rounding_error.cpp
└── E3
    └── E3.py
```

E0 - Comprehension Questions

Task:

1. What does numerical stability mean and why does it occur?
2. Why can higher accuracy (for example due to finer discretization) lead to numerical instability?

---

1. Numerical stability refers to the behavior of a numerical algorithm or computation when dealing with approximate values, typically due to the limited precision of computer representations of real numbers. Therefore one property of an algorithm or problem is its translatability to a computer representation. However, some problems can be ill-conditioned even before considering the transfer to a computational context.

   An algorithm is considered numerically stable if small perturbations or errors in the input values result in small changes in the output, and numerically unstable if small errors in the input values can cause large errors or unexpected results in the output.

   Numerical stability occurs because of the following factors:

   (a) **Finite precision:** Computers can only store a finite number of digits to represent real numbers, usually in floating-point format. This limitation introduces rounding errors and can lead to loss of precision, especially when performing arithmetic operations on numbers of vastly different magnitudes.

   (b) **Error propagation:** In some numerical algorithms, errors introduced at one step can accumulate and propagate through subsequent steps, potentially leading to inaccurate results. A numerically stable algorithm is designed to minimize such error propagation.

   (c) **Ill-conditioned problems:** Some mathematical problems are inherently sensitive to small changes in the input data, making them difficult to solve accurately using numerical methods. These problems are said to be ill-conditioned, and solving them requires special care to maintain numerical stability.

---

2. Higher accuracy, achieved through finer discretization or higher precision arithmetic, can lead to numerical instability in some cases, especially when the discretized representation of a continuous problem introduces new challenges that are not present in the continuous setting. Here are some reasons why this can happen:

   (a) **Error amplification:** Finer discretization or higher precision arithmetic can sometimes amplify small errors, which then propagate through the computation, leading to instability. This can happen, for example, when solving differential equations with high spatial or temporal resolution, where local errors can accumulate and grow over time or space, causing the numerical solution to diverge from the true solution.

   (b) **Ill-conditioning:** As the discretization becomes finer, the resulting system of equations can become ill-conditioned, meaning the condition number of the system matrix increases. This makes the problem more sensitive to small perturbations in input data or rounding errors, causing the numerical solution to be less accurate or unstable. This is particularly common when solving linear systems using direct methods like Gaussian elimination or LU decomposition.

(c) **Stiffness:** Some problems exhibit stiffness, which is a property of a system where certain components evolve on vastly different timescales or length scales. Finer discretization can exacerbate stiffness, making it harder to choose appropriate time steps or mesh sizes for stable numerical integration. In such cases, specialized numerical methods, like implicit schemes or adaptive mesh refinement, may be required to maintain stability.

(d) **Round-off errors:** Finer discretization or higher precision arithmetic can introduce more opportunities for round-off errors due to the increased number of operations or larger system size. These errors can accumulate and propagate, leading to numerical instability if not properly managed.

It's worth noting that finer discretization or higher precision arithmetic can also improve the stability and accuracy of numerical methods in certain cases. The key is to choose an appropriate discretization or precision level, and to employ numerically stable algorithms that are well-suited to the problem at hand.

"*I hope we'll learn how to distinguish between these cases.*"

E1 - Hello World

Task:

Install a compiler (e.g. GCC) on your system. Test it by writing a program that outputs Hello World. Please use only up to date python or C++ Versions for your codes (python3 or C++11 or higher). This holds for this, as well as for all following exercises.
Voluntary bonus task: If you use GCC, read up on the -Ox compiler flag (with $x \in \{1, 2, 3\}$), which optimizes the code during compilation and can reduce the computation time of the program significantly.

```
1A_00_Jasniewicz-Lehner.zip
├── E1
│   ├── hello_world
│   └── hello_world.cpp
├── E2
│   └── ...
└── E3
    └── ...
```

---

hello_world.cpp introduces a very basic example of how to write and execute a C++ program using the console.

Our approach:

1. Installing all prerequisites, using Vscode, WSL2 (Ubuntu 22.04) and gcc (11.3.0).
2. Writing the .cpp-file.
3. Compiling the .cpp-file by using the command
   `g++ hello_world.cpp -o hello_world`
4. Executing the file using
   `./hello_world`

   The console outputs `Hello, World!`

---

hello_world.cpp

```cpp
// This is a single-line comment
/*
    This is a
    multiline-comment
*/
#include <iostream>    // used for 'cout'

using namespace std;   // to use 'cout' directly instead of typing 'std::cout'

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

Voluntary bonus task

The `-Ox` compiler flag in `GCC` (GNU Compiler Collection) is used to control the optimization level of the compiled code, where x can be 1, 2, or 3. Higher optimization levels enable more aggressive optimization techniques, which can potentially result in faster code execution and reduced code size. However, higher optimization levels can also increase the compilation time and make debugging more difficult, as the optimized code may be significantly transformed from the original source code.
Here's a brief overview of the `-Ox` flags in `GCC`:

- `-O1`: This flag enables a basic set of optimizations that aim to reduce code size and execution time without significantly increasing compilation time. The optimizations enabled by `-O1` include, among others, **constant propagation, dead code elimination, and function inlining.**
- `-O2`: This flag enables a more extensive set of optimizations compared to `-O1`. It aims to further reduce code size and execution time without causing a significant increase in compilation time. The optimizations enabled by `-O2` include all the optimizations enabled by `-O1`, as well as additional optimizations such as **loop unrolling, strength reduction, and instruction scheduling.**
- `-O3`: This flag enables even more aggressive optimizations compared to `-O2`. It aims to maximize code performance, potentially at the expense of code size and compilation time. The optimizations enabled by `-O3` include all the optimizations enabled by `-O2`, as well as additional optimizations such as **loop vectorization, aggressive function inlining, and prefetching**.

When using these optimization flags, it's important to be aware of the potential trade-offs between code performance, code size, compilation time, and debuggability. In some cases, higher optimization levels can lead to unexpected behavior due to aggressive optimizations, so it's essential to thoroughly test the code when using `-O2` or `-O3`.

E2 - Rounding error

Task:

In some cases, numerical stability can be established by cleverly transforming or approximating the critical calculation steps. In the following, you will learn this on the basis of three example calculations in order to be able to work out simple stabilization approaches by yourself in the future.
Write a program that calculates the following expressions:

(a) for large $x \gg 1$:

$$\frac{1}{\sqrt{x}} - \frac{1}{\sqrt{x+1}}$$

(b) for small $x \ll 1$:

$$\frac{1-\cos x}{\sin x}$$

(c) for small $\delta \ll 1$:

$$\sin(x+\delta) - \sin x$$

Then look for a numerical calculation path that avoids cancellation. Compare the relative errors between the calculation path with cancellation and without cancellation.

Each of the given terms can be either transformed or approximated so that cancellation doesn't occur. First, we rewrite every given term into a numerically more stable one. We then calculate values for each algorithm and point out if and when cancellation occurs.
This behavior can be easily visualized, which is done in respective plots.

(a) **"Cleverly" transforming**

$$= \frac{\sqrt{x+1}-\sqrt{x}}{\sqrt{x}\sqrt{x+1}} = \frac{(\sqrt{x+1}-\sqrt{x})(\sqrt{x+1}+\sqrt{x})}{\sqrt{x}\sqrt{x+1}(\sqrt{x+1}+\sqrt{x})}$$

$$= \frac{(x+1)-x}{\sqrt{x}\sqrt{x+1}(\sqrt{x+1}+\sqrt{x})} = \boxed{\frac{1}{\sqrt{x}\sqrt{x+1}(\sqrt{x+1}+\sqrt{x})}}$$

This analytical transformation doesn't contain any subtractions. Addition and multiplication however are numerically stable.

If we assume $x \gg 1$, the term can be further reduced to

$$\approx \frac{1}{\sqrt{x}\sqrt{x}(2\sqrt{x})} = \boxed{\frac{1}{2x^{3/2}}}$$

with $\sqrt{x+1} \approx \sqrt{x}$ for $x \gg 1$.

(b) **Approximating** $\cos x$

Both $\sin x$ and $\cos x$ have easy to handle taylor (expansion) series. However, in this case only the approximation of $\cos x$ is necessary, since the division by $\sin x$ is numerically stable.

Approximating $\cos x$ to the second order yields

$$\frac{1-\cos x}{\sin x} = \frac{1-\left(1-\frac{x^2}{2}+\frac{x^4}{24}-\dots\right)}{\sin x} \approx \frac{\frac{x^2}{2}}{\sin x} = \boxed{\frac{x^2}{2\sin x}}\,.$$

This term is numerically stable.

(c) **Approximating** $\sin(x+\delta)$

Approximating the first $\sin$ at δ (1st order) leads to the expression

$$\sin(x+\delta) - \sin x \approx \underbrace{(\sin x + \delta\cdot\cos x)}_{\text{1st order}} - \sin x = \boxed{\delta\cdot\cos x}\,.$$

Again, cancellation is bypassed by avoiding subtraction.

Data & Visualisation

We are interested in determining when cancellation effects due to limited precision occur. To achieve this, we iterated through very large and small numbers in order to identify a cut-off point.
Additionally, the cut-off point and the respective stable calculation can be visualized in comprehensible plots.

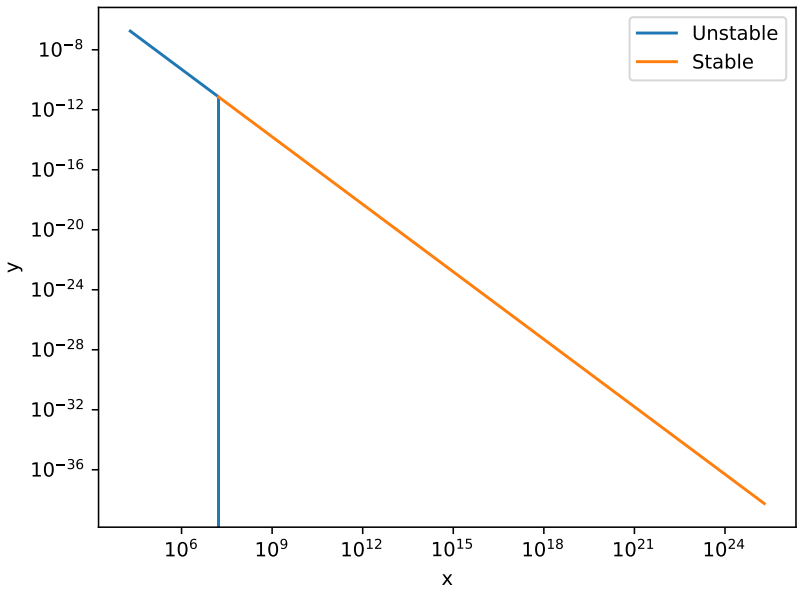

Figure 1.1: **a) Cut-off:** `x=1.676e+07, f(x)=7.28717e-12`

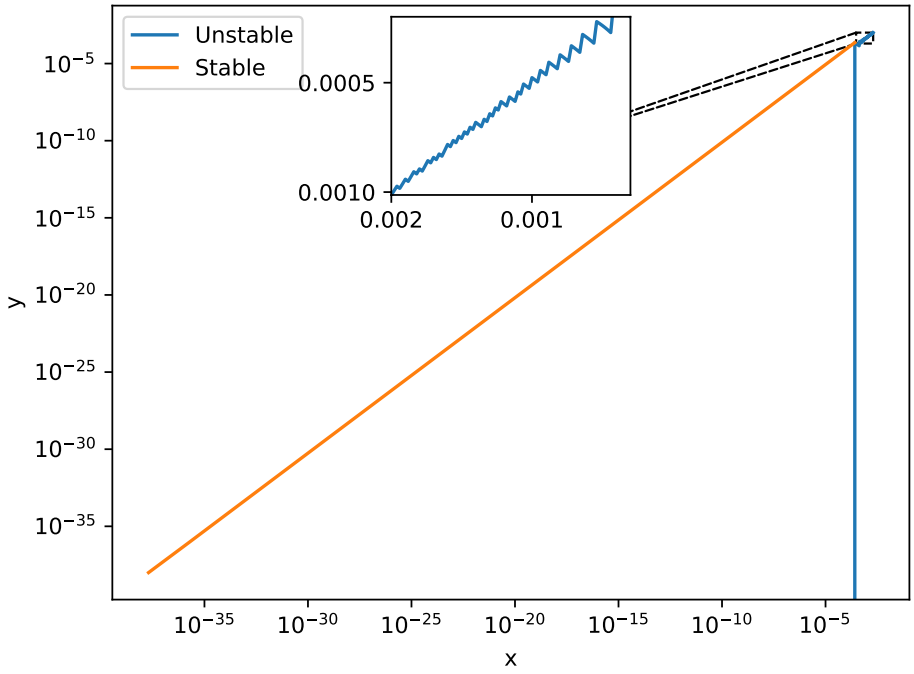

Figure 1.2: **b) Cut-off:** `x=0.00026, f(x)=0.000229249`

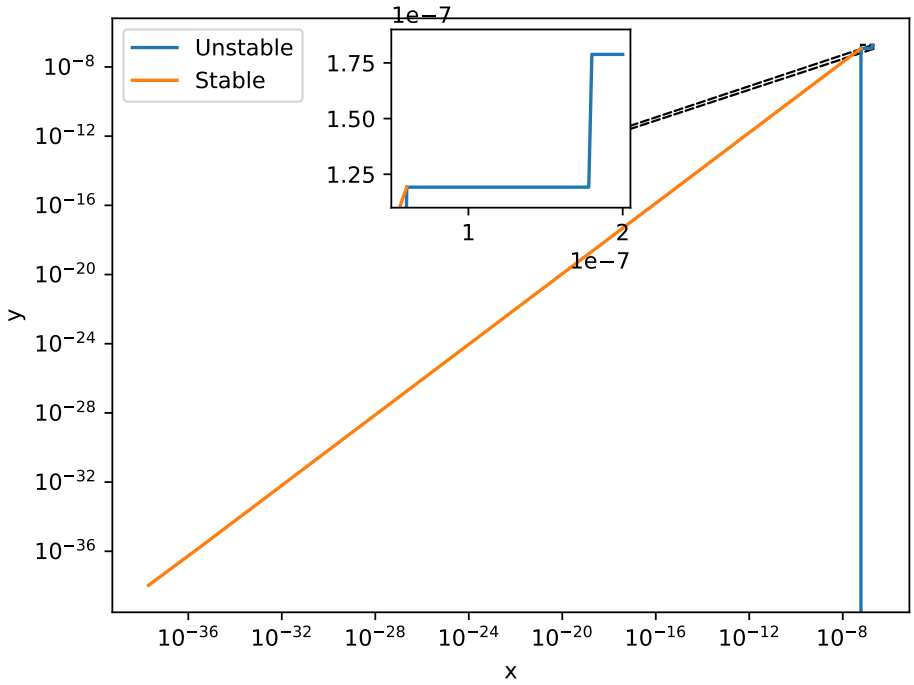

Figure 1.3: **c) Cut-off:** `δ=6e-08, f(δ, x=1)=1.19209e-07`

Files & Execution

All files can be compiled and run with the makefile.

`makefile`

```
CXX = gcc
CXXFLAGS = -Wall -O0 -std=c++17

SRC_DIR = .
OBJ_DIR = ./build/Debug
SRC_FILES = $(wildcard $(SRC_DIR)/*.cpp)
OBJ_FILES = $(patsubst $(SRC_DIR)/%.cpp, $(OBJ_DIR)/%.o, $(SRC_FILES))

EXECUTABLE = outDebug

all: $(EXECUTABLE)

$(OBJ_FILES): | $(OBJ_DIR)

$(OBJ_DIR):
  mkdir -p $(OBJ_DIR)

$(EXECUTABLE): $(OBJ_FILES)
  $(CXX) $(CXXFLAGS) -o $@ $^ -lstdc++ -lm

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.cpp
  $(CXX) $(CXXFLAGS) -c -o $@ $<

.PHONY: clean run

clean:
  rm -f $(OBJ_DIR)/*.o $(EXECUTABLE)

run: $(EXECUTABLE)
  ./$(EXECUTABLE)
```

`make` - Compile the code. (Check the `c++XX` version at the very top)
`make run` - Run the code.

```
1A_00_Jasniewicz-Lehner.zip
├── E1
│   └── ...
├── E2
│   ├── a.cpp
│   ├── a.hpp
│   ├── b.cpp
│   ├── b.hpp
│   ├── c.cpp
│   ├── c.hpp
│   ├── makefile
│   ├── plotting.py
│   └── rounding_error.cpp
└── E3
    └── ...
```

For easier assessment, here is an outline of the main code.
outline.cpp (not included in .zip)

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include "a.hpp"
#include "b.hpp"
#include "c.hpp"

using namespace std;
// Function to write values to a file for plotting (omitted for brevity)
// All limits are chosen by try and error to demonstrate the cut-off point
int main() {
    // Define variables and flags for different calculations
    int steps;
    float limit_a, limit_a_stable, limit_b, limit_b_stable, limit_c, limit_c_stable;
    float* a, *a_stable, *b, *b_stable, *c, *c_stable;

    // a) Calculate unstable and stable results for a)
    steps = 1000;
    limit_a = 2e7;
    limit_a_stable = 2e25;
    a = a_func(limit_a, steps);
    a_stable = a_func_stable(limit_a_stable, steps);

    // Print and/or write results for a)

    // Free memory for a)
    delete[] a;
    delete[] a_stable;

    // b) Calculate unstable and stable results for b)
    steps = 100;
    limit_b = 2e-3;
    limit_b_stable = 2e-36;
    b = b_func(limit_b, steps);
    b_stable = b_func_stable(limit_b_stable, steps);

    // Print and/or write results for b)

    // Free memory for b)
    delete[] b;
    delete[] b_stable;

    // c) Calculate unstable and stable results for c)
    steps = 100;
    limit_c = 2e-7;
    limit_c_stable = 2e-36;
    c = c_func(limit_c, steps);
    c_stable = c_func_stable(limit_c_stable, steps);

    // Print and/or write results for c)

    // Free memory for c)
    delete[] c;
    delete[] c_stable;

    return 0;
}
```

E3 - Stability

Task:

Higher accuracy does not always mean higher stability. We will study this by using the Euler method known from the lecture and writing a first numerical integration as preparation for later tasks.
The differential equation

$$\dot{y}(t) = -y(t), \quad y(0) = 1$$

with the analytical solution

$$y(t) = \exp(-t)$$

shall be solved numerically using the Euler method and the symmetric Euler method with a step size Δt. The Euler method yields the recursion

$$y_{n+1} = y_n(1 - \Delta t)$$

and the symmetric Euler method yields

$$y_{n+1} = -2\Delta t y_n + y_{n-1}.$$

(a) Implement both the Euler method and the symmetric Euler method and start with initial values $y = 1$ and for the symmetric Euler method additionally with $y_1 = \exp(-\Delta t)$. Compare your results with the analytical solution in the interval $t \in [0, 10]$.

Hint: Find an appropriate value for Δt.

(b) Compare your results from the previous part of the task with the results when you start the Euler method with $y_0 = 1 - \Delta t$ and the symmetric Euler method with $y_0 = 1$ and $y_1 = y_0 - \Delta t$. Interpret your results.

(a) The focus of this task lies in visualization and interpretation, making it relatively straightforward. There is only one file, `E3.py`, which does the math and creates two plots, 1.4 and 1.5.

For a), we find that the **Euler method** yields the most accurate results, especially for larger values, while the **symmetric Euler method (sEm)** is less accurate and diverges in an alternating pattern. However, the **sEm** is *more* accurate for values closer to the starting value, which can be seen in 1.4.

(b) Changing the starting values doesn't affect the overall behavior of the two methods, particularly the alternating-diverging pattern from the **sEm**. Nevertheless, a change in the starting values *does* effect the quality of the approximation.

Since b) provided starting values that don't match the analytical solution, the approximations become less accurate. Especially the **sEm** diverges more quickly and aggressively, which renders the approximation useless.

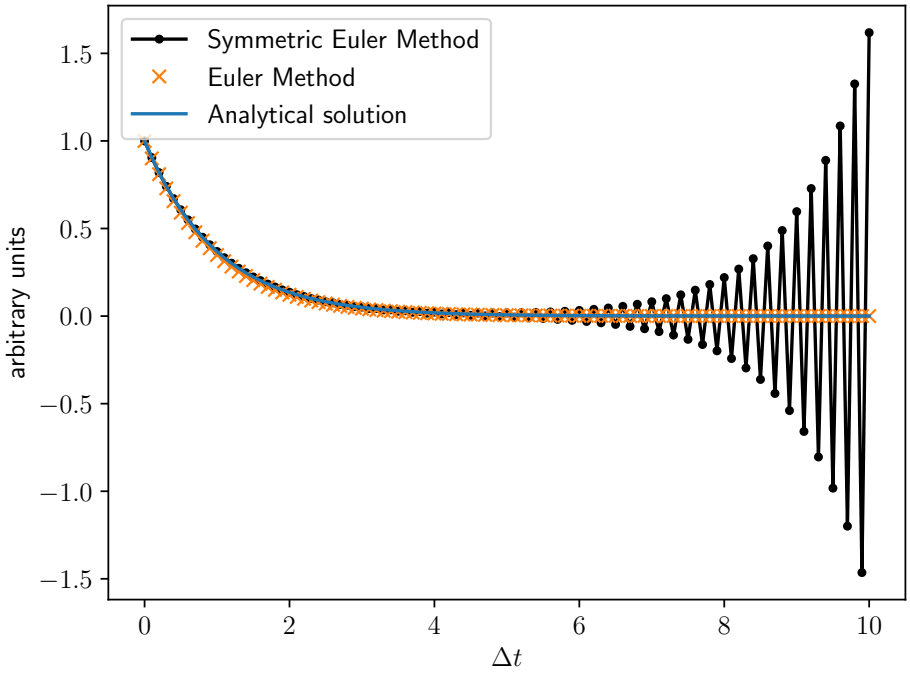

Figure 1.4: `steps=100`, $\Delta t = 0.1$

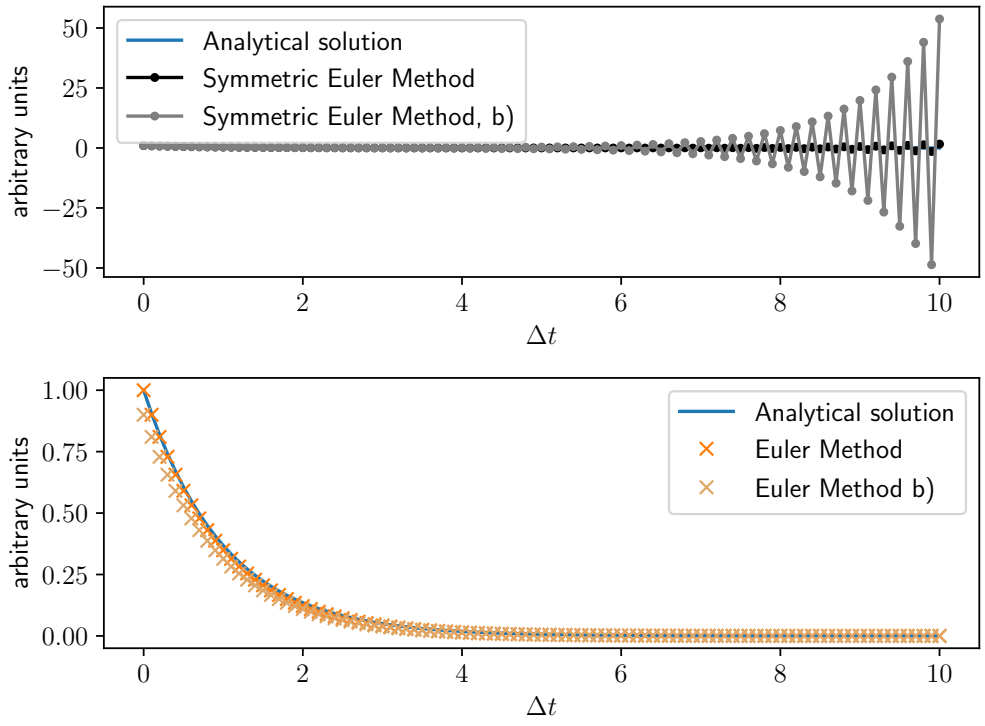

Figure 1.5: `steps=100`, $\Delta t = 0.1$