Chapter 2
Motivation

Abstract This chapter introduces the reader to the "world" of numerical modelling using the decay problem as a first benchmark. Discussed are finite differences, explicit and implicit schemes, and conditions of consistency, accuracy, stability, and efficiency. FORTRAN codes and SciLab scripts are used to create a first numerical prediction model and graphical display of results.

2.1 The Decay Problem

2.1.1 The Problem

The so-called *decay problem* is chosen as a first example to demonstrate what numerical modelling is. In mathematical terms, this problem can be expressed as:

$$\frac{dC}{dt} = -\kappa \cdot C \tag{2.1}$$

where C is concentration of a substance, t is time, and κ (the Greek symbol "kappa") is a positive constant parameter. The d symbol refers to a change of a variable with respect to some other parameter such as time as in the above equation.

2.1.2 Physical Interpretation

The term on the left-hand side of Eq. (2.1) refers to the temporal change of concentration per time unit. The right-hand side specifies this temporal change. For $\kappa = 0$, there will no change and C remains unchanged at its initial value. With $\kappa \neq 0$, on the other hand, the right-hand side is negative since concentration is always a positive quantity. Accordingly, C will gradually decrease with time at a rate in proportion to concentration itself at any time instance.

J. Kämpf, *Ocean Modelling for Beginners*,
DOI 10.1007/978-3-642-00820-7_2, © Springer-Verlag Berlin Heidelberg 2009

In mathematics, Eq. (2.1) is termed a *first-order ordinary differential equation* and, as it requires an initial value for C, it is also referred to as an *initial-value problem*.

2.1.3 Example

Grandma used to keep a 10-litre carton of red wine in a storage room. I could not resist and crept every night to this room to get myself 1 litre of wine. To avoid grandma noticing the fast disappearance of the wine, each night, I topped up the carton with tab water. However, I was caught soon or do you really think that grandma would not notice changes in the wine's taste and color owing to dilution with water? Anyway, the reader can easily work out how the concentration of wine changed with every day of my early-year drinking habit. All that needs to be done is to take away 10% of the wine concentration on a daily basis. The following table shows the result of this.

Day	Wine content (l)	Wine concentration (%)
0	10	100
1	9.0	90.0
2	8.1	81.0
3	7.29	72.9
4	6.56	65.6
5	5.9	59.0
6	5.31	53.1
7	4.78	47.8
8	4.3	43.0
9	3.87	38.7
10	3.49	34.9
11	3.14	31.4
12	2.82	28.2
13	2.54	25.4
14	2.29	22.9
15	2.06	20.6
16	1.85	18.5
17	1.67	16.7
18	1.5	15.0
19	1.35	13.5
20	1.22	12.2
21	1.09	10.9
22	0.98	9.8

Figure 2.1 shows a graph of the outcome of this first numerical prediction model, only using a piece of paper and a pen. As expected, there is a decrease of wine content as the days go past. In fact, this decrease is approximately exponential.

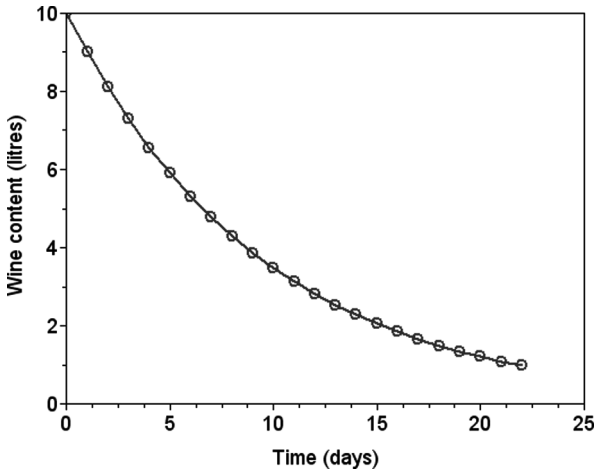

Fig. 2.1 Evolution of wine content

2.1.4 How to Produce a Simple Graph with SciLab

This exercise explains how to use SciLab to produce Fig. 2.1.

Step1: Open a suitable text editor and insert and save the three rows of the above table (without header) in a file called, for instance, "winethief.txt".
Step 2: Fire up SciLab and change to the folder containing this file.
Step 3: Enter the following commands (each followed by <return>):

```
x=read("winethief.txt",-1,3);
plot(x(:,1),x(:,2));
xtitle("  ","Time (days)", "Wine content (litres)","  ")
```

The "read" command imports data from the file. The parameter "−1" is used if the number of rows is unknown. "3" means: read the first three columns, "−1" does not work for columns. The above plot command plots values of row 2 against those of row 1. The last line adds axis labels. The semicolon ";" suppresses output to the SciLab command window.
Step 4: Change line width, font size, etc. in the graphics window by selecting "Edit Figure Properties" in the "File" menu or by clicking "GED" in the graphics window. The reader is encouraged to "play around" with all options available.
Step 5: Export the graph using a suitable format. I often selected the PostScript format and converted images into Portable Network Graphics files (PNG) with ImageMagick.

2.2 First Steps with Finite Differences

2.2.1 Finite Time Step and Time Level

With the use of a discrete *time step* Δt, we may formulate (2.1) as:

$$\frac{C^{n+1} - C^n}{\Delta t} = -\kappa C^n \tag{2.2}$$

where the integer n refers to a certain *time level*. This time index must not be confused with "to the power of". Conventionally, $n = 0$ gives the concentration at start time of your simulation, $n = 1$ refers to the concentration after one time step (of Δt in duration), $n = 2$ refers to the concentration after two time steps, and so on.

2.2.2 Explicit Time-Forward Iteration

It is convenient to move the unknown variable in (2.2) to the left-hand side of the equation and shuffle all known terms to the right-hand side. This gives:

$$C^{n+1} = C^n - \Delta t \cdot \kappa \cdot C^n = (1 - \Delta t \cdot \kappa)\, C^n \tag{2.3}$$

where $C^{n=0}$ refers to the initial concentration that needs to be prescribed together with values of κ and Δt. This iterative method uses values known at a certain time level n to predict the value of C at the next time level $n + 1$ and is therefore called *explicit time-forward iteration*.

2.2.3 Condition of Numerical Stability for Explicit Scheme

As can be seen from (2.3), with every time step, the concentration becomes decimated by a certain fraction in an iterative manner. This fraction is given by the product $\kappa \cdot \Delta t$. It is at hand to request that this product be less than unity, otherwise the predicted concentration would become negative, which would not make sense. For $\kappa \cdot \Delta t > 2$, the magnitude of concentration would even increase. The corresponding condition:

$$\Delta t < \frac{1}{\kappa} \tag{2.4}$$

is called a *condition of numerical stability*. Hence, the prediction of (2.3) is only stable when (2.4) is satisfied. Accordingly, the maximum time step that can be chosen depends on the value of κ.

2.2.4 Implicit Time-Forward Iteration

Alternatively, Eq. (2.1) can be formulated in finite-difference form as:

$$\frac{C^{n+1} - C^n}{\Delta t} = -\kappa \cdot C^{n+1} \tag{2.5}$$

where the concentration on the right-hand side is evaluated at the next time level $n + 1$. This approach might sound strange to some readers, but if we reorganize this equation, we yield a clear separation of known and unknown terms of the form:

$$C^{n+1} = \frac{C^n}{(1 + \Delta t \cdot \kappa)} \tag{2.6}$$

The clear advantage of this *implicit scheme* over the explicit approach is that it is numerically stable for any value of Δt. The denominator in the later equation always exceeds unity, so that concentration gradually decreases with time (and never changes sign).

2.2.5 Hybrid Schemes

One could also use a mix between the explicit and the implicit scheme, which can be formulated as:

$$\frac{C^{n+1} - C^n}{\Delta t} = -\alpha \cdot \kappa \cdot C^{n+1} - (1 - \alpha)\,\kappa \cdot C^n \tag{2.7}$$

where the weighting factor α (the Greek symbol "alpha") has to be chosen from a range between zero and unity. The choice of $\alpha = 1$ gives the fully implicit scheme, whereas $\alpha = 0$ leads to the fully explicit scheme. With $\alpha = 0.5$, we obtain a so-called *semi-implicit* scheme.

2.2.6 Other Schemes

There are more advanced schemes such as the "Runge-Kutta scheme" or the "Adams-Bashforth scheme", not discussed here, that in addition to current and future time level consider a number of sub-time steps. The accuracy and efficiency of the prediction model can be significantly improved with such schemes.

2.2.7 Condition of Consistency

The exact analytical solution of the decay problem (2.1) for an initial concentration of C_o is given by:

$$C(t) = C_o \exp(-\kappa \cdot t) \tag{2.8}$$

where "exp" is the exponential function. A numerical model is said to be *consistent* if its finite-difference solution converges toward the solution of the governing differential equation when the numerical time step (or grid size) is made vanishingly small. This implies that the concentration predicted by our model should get the closer to the true solution for a decrease of the time step Δt.

2.2.8 Condition of Accuracy

A certain error referred to as *truncation error* is made when using finite differences. *Round-off errors* are another source of error, being related to the fact that computers can represent numbers only with a finite number of digits. Both errors should stay reasonably small over the duration of a simulation.

2.2.9 Condition of Efficiency

Large programs may require substantial computer space for data output and storage, and completion of model runs may take a long time. Hence, model codes have to be written in an efficient manner such that the task is completed within a reasonable time span and without "stuffing up" the computer with enormous amounts of data.

2.2.10 How Model Codes Work

The compiler translates the FORTRAN 95 code line by line and *from top to bottom*. This implies that parameters must be declared and specified before they can be manipulated. Declaration means specification of the type of the parameter. There are integers, real numbers, arrays, characters and logical parameters.

Specification means allocation of values to parameters. In principle, each line of a computer code can only have a single unknown on the left-hand side of an equation, such as "$x = b + c$", where b and c have to be declared and assigned values farther up in the code, and x has to be at least declared.

2.2.11 The First FORTRAN Code

Write a first FORTRAN code that prints "Hello World" on the display. This is an old tradition among modellers. The solution is the code:

```
PROGRAM first
write(6,*)"Hello World"
END PROGRAM first
```

FORTRAN programs start with a PROGRAM *name* statement and finish with a END PROGRAM *name* statement. My program is called "first", but the reader is free to choose a different name. Save this file as "first.f95".

2.2.12 How to Compile and Run FORTRAN Codes

Open the Command Prompt window (on Windows systems this is found under Start => All Programs => Accessories => Command Prompt) and move to the folder containing your FORTRAN source file.

Step 1: Compile the program by entering the command:

```
g95 -o first.exe first.f95
```

where "-o" specifies the name of the executable program.
Step 2: Correct errors until the compiler does not return any error messages.
Step 3: If the compiling process was successful, the newly created file "first.exe" can be executed by simply double-clicking its icon in a file window or by entering "first" <return> in the Command Prompt window.

The result of this will be the display of "Hello World" in the Command Prompt window. Congratulations!

2.2.13 A Quick Start to FORTRAN

All constants, parameters, variables and arrays have to declared before use. Hereby, full numbers called "integer" (-3, 0, 1, 3, etc.) are distinguished from decimal numbers called "real" (1.2, 4.2, -5.23, etc.). Other options are logical expressions (true or false) and text (characters). Constant parameters are declared at the beginning of the code with:

```
INTEGER, PARAMETER :: nx = 11 ! horizontal dimension
INTEGER, PARAMETER :: nz = 5 ! vertical dimension
```

```
REAL, PARAMETER :: G = 9.81 ! acceleration due to gravity
REAL, PARAMETER :: RHOREF = 1028.0 ! reference density
REAL, PARAMETER :: PI = 3.14159265359 ! pi
```

Text after a pronunciation mark is treated as a comment and ignored by the compiler. Although not required, comments are very useful aids to highlight the structure of the program and as reminders for future uses. Parameters that are allowed to change values during the program's execution are declared with:

```
REAL :: wspeed ! wind speed
INTEGER :: k ! grid index
CHARACTER(3) :: txt
```

In this example, "txt" is a character with three letters. One-dimensional and two-dimensional arrays are declared with:

```
REAL :: eta(0:nx+1) ! sea-level elevation
REAL :: w(0:nz+1,0:nx+1) ! vertical velocity
```

With nx=11 and nz = 5, for instance, "eta" obtains 11+2=13 so-far unassigned elements: eta(0), eta(1), ··· ..., eta(11), eta(12), and "w" is a two-dimensional array of 13 columns and 7 rows. After the declaration section, values can be assigned to these arrays using DO loops such as:

```
DO k = 0,nx+1
  IF(k > 50) THEN
     eta(k) = 1.0
  ELSE
    eta(k) = 0.0
  END IF
END DO
```

This DO-loop repeats certain calculations for the index "k" running from 0 at steps of 1 to nx+1. If the reader wants to do it the other way around, the solution is:

```
DO k = nx+1,0,−1
   IF(k > 50) THEN
     eta(k) = 1.0
   ELSE
     eta(k) = 0.0
   END IF
END DO
```

This example also includes an IF statement. Options are ">" (greater than), "<" (less than), "==" (equal to), ">=" (greater or equal), "<=" (less or equal), and "/="

(not equal). If there is only one line in an IF-statement, the "ELSE" and closing "END IF" statements can be dropped, such as in the following example:

```
DO k = nx+1,0,−1
  eta(k) = 0.0
  IF(k > 50) eta(k) = 1.0
END DO
```

Files for output can be opened with the statement:

```
OPEN(10, file = 'Ex1a.txt', form = 'formatted', status = 'unknown')
```

The first entry (10) is a reference unit number for subsequent WRITE or READ statements. The "file" entry specifies the desired filename. The "form" entry specifies whether to have ASCII numbers or binary numbers as output. I chose ASCII output. The status entry "unknown" implies new creation of a file if this does not exist, otherwise an existing file will be overwritten. Be careful not to overwrite files that might be needed in the future. Other status options are "new" or "old".

Output of data is done via "WRITE" statements such as

```
WRITE(10,*) G
```

where the unit number (10 here) refers to a file opened before, and the "*" symbol creates a standard format. Note that the unit number 6 is reserved for output to the screen as in our first FORTRAN code. Similarly, "READ(5,*)" reads input from the keyboard. Several outputs at once can be produced with:

```
WRITE(10,*) eta(10), eta(20), eta(30)
```

Doing this repeatedly, there will be three columns of data. Output of an entire row of an array is done with:

```
WRITE(10,*) (eta(k), k = 1,nx)
```

Files no longer needed for output should be closed with the statement:

```
CLOSE(10)
```

2.3 Exercise 1: The Decay Problem

2.3.1 Aim

The aim of this exercise is to predict the decay of a substance according to (2.1) using a FORTRAN code based on either the explicit or the implicit scheme.

2.3.2 Task Description

Consider a substance that has an initial concentration of 100% and use a decay constant of $\kappa = 0.0001$ per second (or $\kappa = 10^{-4}\ \mathrm{s}^{-1}$). Choose different values of the time step to verify whether the prediction becomes more accurate for a finer temporal resolution. Explore both the explicit and the implicit scheme.

2.3.3 Instructions

Use any text editor to write the FORTRAN code and save the file under the name "Exercise1.f95". Blanks or other unusual symbols are not permitted here. Other filenames may be used, but the reader should make sure that the filename is not too long and that it has something to do with the exercise. The file extension "f95" identifies this file as a FORTRAN 95 source code.

2.3.4 Sample Code

The Fortran code for this exercise, called "winethief.f95" can be found in the folder "Exercise 1" on the CD-ROM accompanying this book.

2.3.5 Results

As a result of the model run, the data output files "output1.txt" or "output2.txt" should appear in the file list. The MODE parameter in the code switches between the explicit and the implicit schemes. To avoid the recompiling procedure, values for "mode" could be alternatively read from the keyboard with "READ(5,*) mode".

Figure 2.2 shows model results for a time step of $\Delta t = 3600\,\mathrm{s}$ using either the explicit scheme (2.3) or the implicit scheme (2.6). As can be seen, the explicit scheme slightly underestimates the correct concentration, whereas the implicit scheme slightly overestimates concentration. A semi-implicit approach would probably give the best solution, but this remains to be verified by the reader.

With a time step of 3600 s, completion of the model run took only a few seconds on my computer. The accuracy of the prediction can be substantially improved with choice of a much finer temporal resolution with a time step of, say, $\Delta t = 1\mathrm{s}$, which the reader can easily verify.

2.3.6 Additional Exercise for the Reader

Repeat this exercise with use of the hybrid scheme (2.7) and explore the solutions for $\alpha = 0.25$, 0.5 and 0.75.

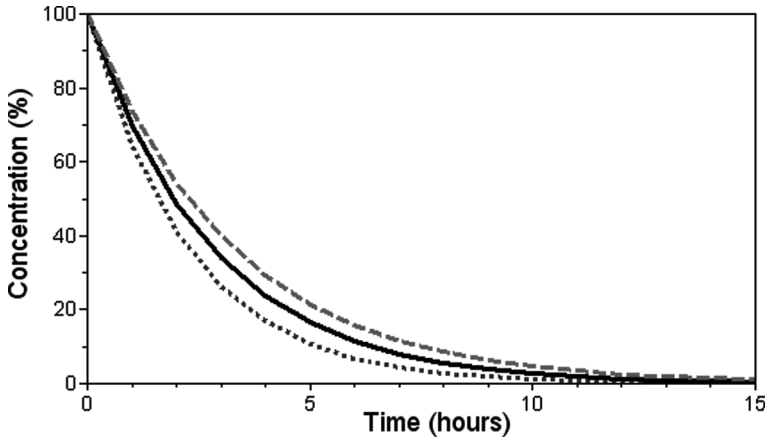

Fig. 2.2 Evolution of concentration (%) as a function of time. The *upper* and *lower curves* show results using the implicit and the explicit schemes, respectively. The *middle curve* displays the analytical solution according to Eq. 2.8

2.4 Detection and Elimination of Errors

2.4.1 Error Messages

If a FORTRAN code contains errors, the compiler will return one or more error messages. There are a few important steps to follow for successful detection and elimination of errors.

2.4.2 Correct Errors One by One

Only correct one error at a time with reference to the first error message. Often other errors are just followers of the first one. Similarly, an important rule is that the code should be compiled after each single alteration made. It can be tedious to locate errors after a dozen changes have been implemented without verification. Errors are often the result of a lack of concentration.

2.4.3 Ignore Error Message Text

Occasionally, the compiler's error messages are confusing and misleading. Therefore, I recommend to ignore message text and rather focus on the line number this message refers to.

2.4.4 Frequent Errors

Errors are frequently associated with misspelling such as “0” (zero) instead of the letter “o” or vice versa, or mistaking “1” (one) as the letter “l”. Often an ENDDO, ENDIF or a bracket is just missing.

2.4.5 Trust Your Compiler

Important is to always trust the compiler. If there is an error message then there is an error in your code. The error message text might be misleading, but the presence of at least one error is a fact. In a state of frustration after endless unsuccessful search after errors, the reader should take a good rest, perhaps a walk or a sleep. Breaks are always important!

2.4.6 Display Warnings

Warnings can be displayed with addition of “-Wall” (display all warnings) in the compiling command. For Exercise 1, for example, this command reads:

```
g95 -Wall -o winethief.exe winethief.f95
```

Warning messages should be explored for potential errors in the code.

http://www.springer.com/978-3-642-00819-1

Ocean Modelling for Beginners
Using Open-Source Software
Kaempf, J.
2009, XV, 175 p. With online files/update., Hardcover
ISBN: 978-3-642-00819-1