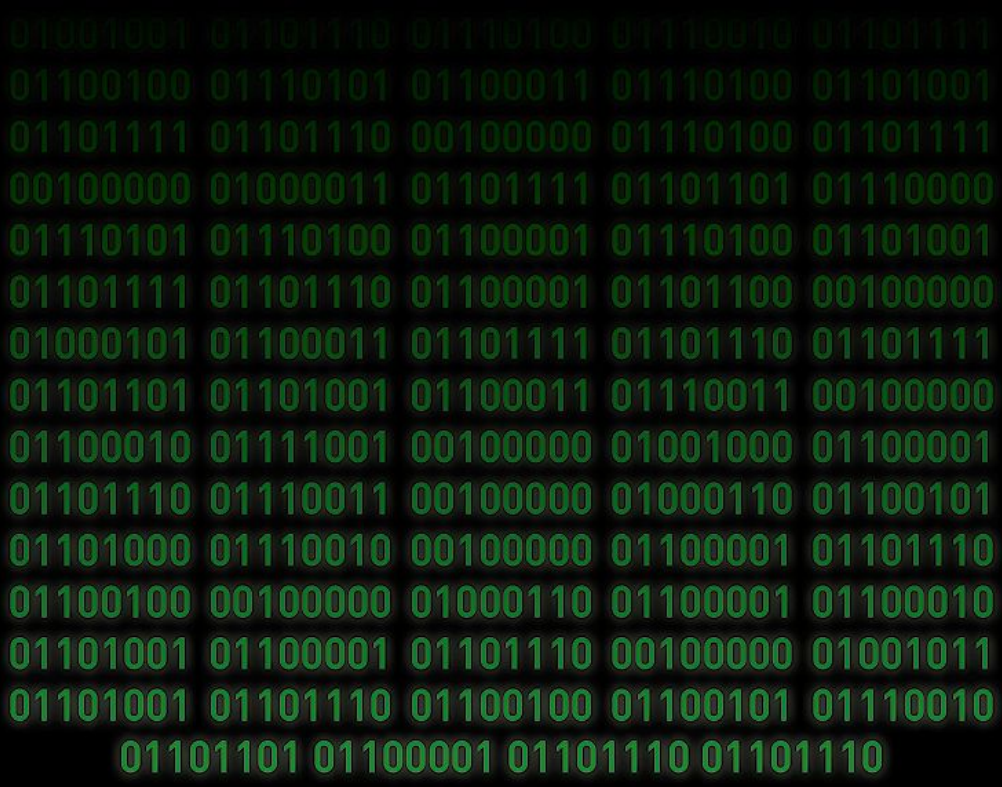
OXFORD

INTRODUCTION TO COMPUTATIONAL ECONOMICS USING FORTRAN

HANS FEHR AND FABIAN KINDERMANN



Introduction to Computational Economics Using Fortran

Introduction to Computational Economics Using Fortran

Hans Fehr
Fabian Kindermann

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Great Clarendon Street, Oxford, OX2 6DP,
United Kingdom

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries

First Edition published in 2018

Impression: 1

Published in the United States of America by Oxford University Press
198 Madison Avenue, New York, NY 10016, United States of America

British Library Cataloguing in Publication Data

Data available

Library of Congress Control Number: 2017949307

ISBN 978–0–19–880439–0 (hbk.)
978–0–19–880440–6 (pbk.)

Printed and bound by
CPI Group (UK) Ltd, Croydon, CR0 4YY

PREFACE

Computer technology has been evolving rapidly in the last century. Although Thomas J. Watson, the first president of IBM, once famously projected that 'there is a world market for maybe five computers', we nowadays rely on computer work in nearly every situation of life. We write emails, book journeys via the internet, and let computers fly airplanes and conduct trains. Naturally, economics has not been unaffected by this trend. On the one hand, the steady increase in computer performance and speed allowed us to solve problems much faster than before. On the other hand, numerical methods that had been invented in the first half of the twentieth century could finally be applied to problems for which analytical solutions did not exist. Nevertheless, owing to the intensive knowledge requirements regarding programming and numerical maths that were needed to solve real economic problems on a computer, the field of computational economics emerged quite slowly. Nowadays however, where a desktop computer or laptop is a high performance machine and various tools for numerical maths are available, computational economics has become increasingly popular.

There exist many textbooks dedicated to the field of computational economics. One category of these books primarily describes the theory and implementation of numerical methods. Judd (1998) and Press et al. (2001) are popular examples of this kind. Other textbooks like Miranda and Fackler (2002), Adda and Cooper (2003), Kendrick, Mercado, and Amman (2006), or Heer and Maussner (2009) show how to apply those methods to economic problems. Most of these books, however, are dedicated to graduate students and require an above-average knowledge of economic theory, programming, and numerical maths. Very often these books are also specializing in a specific field such as international trade and development, dynamic macroeconomics, or finance. They may offer program codes but typically do not provide information on how to install and use a suitable compiler.

This is where our book tries to step in. We offer an introduction to computational economics to students at all levels of education, regardless of their prior programming experience. This book is based entirely on the programming language Fortran. To facilitate the first steps into writing your own codes, we give an introduction to this particular programming language and demonstrate how to download and use free compilers for different operating systems. Our book offers various examples from economics and finance organized in self-contained chapters that speak to the diverse backgrounds of our readers. While early chapters are accessible for undergraduate students, the level of complexity slowly increases, so that the later part of the book is well suited for graduate students at the Master and Ph.D. level. For each of the topics we consider, we first explain some theoretical background, then show how to implement the problem on the computer, and finally discuss simulation results. Since our book serves as an introduction to using

computational methods in various fields of economics and finance, we provide a list of reading material for further study at the end of each chapter. In addition, readers can work through various exercises which promote practical experience and further deepen the economic and technical insights. Therefore, after reading and working through the book, students should have no problem mastering more sophisticated and specialized textbooks and courses in computational economics at the graduate level.

This textbook contains eleven chapters which are organized in three central parts: Part I offers an introduction to programming and tools from numerical maths, Part II presents various applications for beginners, and in Part III we discuss dynamic programming problems for more advanced students and researchers. In order to make it easy for beginners to familiarize themselves with the field of computational economics, Chapter 1 gives an introduction to Fortran, the programming language used throughout the book. In Chapter 2 we discuss several numerical methods that are used frequently throughout this book. Since we do not assume that our readers have a lot of programming experience and only require standard mathematical knowledge, we only introduce the basic concepts and keep the theoretical parts quite condensed. A large set of numerical methods is provided through our toolbox. In the introductory chapters we discuss how to use this toolbox and the methods therein. Hence, after working through the first part of the book, you should have enough knowledge on programming and numerical tools to manage the remaining material in the follow-up parts. Part II comprises five chapters with easily accessible applications. Chapter 3 introduces the static general equilibrium model typically applied to issues of labour markets or international trade. In Chapter 4 we demonstrate different approaches to optimize a portfolio, to price options, and to manage credit and mortality risk. This leads in to Chapter 5, in which we discuss individual savings and investment decisions within the life-cycle framework with uncertain labour income, asset returns, and lifespan. We extend this partial equilibrium life-cycle model to its general equilibrium counterpart, the overlapping generations model, in Chapters 6 and 7. We use this model to investigate dynamic tax problems and optimal pension design in stationary and ageing societies. Finally, Part III of this book consists of four chapters with advanced applications. Chapter 8 introduces numerical solution methods for dynamic programming problems. In Chapter 9 we apply these methods to infinite horizon models of the macroeconomy and study growth, business cycles, and distributional issues. Chapter 10 focuses on advanced life-cycle labour-supply and investment problems using the dynamic programming approach. In the final Chapter 11 we extend the overlapping generations model to account for idiosyncratic earnings risk and study the design of optimal fiscal policies that balance tax distortions and public insurance provision.

While the different chapters somewhat build on one another regarding their economic themes, we tried to keep them as independent and self-contained as possible. Hence, the reader familiar with Fortran could skip Chapter 1 and directly start with Chapter 2, whereas the reader sufficiently familiar with numerical techniques could just take a quick look at Chapter 2 and directly start with studying Parts II and III. The chapters offering

applications follow the same logic and therefore could be also read independently. As already mentioned above, we encourage the reader to work through an extensive number of exercises in each chapter in order to deepen their understanding of the methods learned and to gain practical experience necessary to become a decent computational economist. Note that, especially when working on the first tasks, you will suffer a lot from programming errors and bugs. This is quite normal. Please don't be disappointed, if a program doesn't work at the first or second shot. Code can be written in several minutes. However, getting it to work can take hours or days.

This book is accompanied by a website

`www.ce-fortran.com`.

On this website we provide the source codes to any of the programs discussed in this book. You will also find a link to our toolbox, an instruction on how to install and use it as well as a description of the methods it provides. The website also contains up-to-date information on how to set up a Fortran compiler on your operating system and therefore to get started as quickly as possible on working through this book.

Last but not least, writing such a book is never possible without the help of others. First of all, we want to thank Oxford University Press for deciding to become our publisher. We are especially grateful to Katie Bishop, Susan Frampton, and Subramaniam Vengatakrishnan for their assistance in the production of this book. Over the years we have been working on this we have benefited from comments, suggestions, and endorsements from many colleagues, especially Ben Heijdra, Leonhard Knoll, Laurie Reijnders, Alexander Rothkopf, and Andras Simonovits. Thanks also to our former students Theresa Grimm, Maurice Hofmann, Sarah Lenz, Franziska Schlumprecht, Lorenz Schneck, Lukas Schwabe, Maximilian Stahl, and Patrick Wiesmann who provided excellent research assistance and worked on computer codes that provided the basis for certain chapters. Parts of this book were revised when Hans Fehr visited the ARC Centre of Excellence in Population Aging Research (CEPAR) at the University of New South Wales in 2016. He thanks the members of CEPAR for their hospitality during his stay.

Finally, we hope that you enjoy reading this book and that you have as much fun doing computational economics as we do.

Bonn and Würzburg in June 2017

CONTENTS

Part I

An Introduction to Fortran 90 and Numerical Methods

1 Fortran 90: A simple programming language

Before diving into the art of solving economic problems on a computer, we want to give a short introduction into the syntax and semantics of Fortran 90. As describing all features of the Fortran language would probably fill some hundred pages, we concentrate on the basic features that will be needed to follow the rest of this textbook. Nevertheless, there are various Fortran tutorials on the Internet that can be used as complementary literature.

1.1 About Fortran in general

1.1.1 THE HISTORY OF FORTRAN

Fortran is pretty old; it is actually considered the first known higher programming language. Going back to a proposal made by John W. Backus, an IBM programmer, in 1953, the term Fortran is derived from *The IBM **For**mula **Tran**slation System*. Before the release of the first Fortran compiler in April 1957, people used to use assembly languages. The introduction of a higher programming language compiler tremendously reduced the number of code lines needed to write a program. Therefore, the first release of the Fortran programming language grew pretty fast in popularity. From 1957 on, several versions followed the initial Fortran version, namely FORTRAN II and FORTRAN III in 1958, and FORTRAN IV in 1961. In 1966, the *American Standards Association* (now known as the ANSI) approved a standardized *American Standard Fortran*. The programming language defined on this standard was called FORTRAN 66. Approving an updated standard in 1977, the ANSI paved the way for a new version of Fortran known as FORTRAN 77. This version became popular in computational economics during the late 80s and early 90s. More than 13 years later, the Fortran 90 standard was released by both the *International Organization for Standardization* (ISO) and ANSI consecutively. With Fortran 90, the *fixed format* standard was exchanged by a *free format* standard and, in addition, many new features like modules, recursive procedures, derived data types, and dynamic memory allocation made the language much more flexible. From Fortran 90 on, there has only been one major revision, in 2003, which introduced object-oriented programming features into the Fortran language. However, as object-oriented

programming will not be needed and Fortran 90 is by far the more popular language, we will focus on the 1990 version in this book.

1.1.2 WHY FORTRAN?

Fortran is an imperative, procedural, high-level programming language that is designed and optimized for numerical calculations. In detail this means that: (a) a Fortran program consists of statements that will be executed consecutively (i.e. a Fortran program starts with the first line and ends with the last); (b) Fortran code that will be used several times can be stored in functions and subroutines that can be called up from other program parts; and (c) Fortran abstracts from the details of a computer, i.e. having the right compiler, Fortran code could be run on any computer independent of its hardware configuration and operating system. Fortran can also be viewed as a *general purpose programming language*. A general purpose language is a programming language that is designed for software development in many different application domains. In contrast so-called *domain-specific programming languages* are constructed to basically serve one application domain, like e.g. Matlab for matrix operations and Mathematica for symbolic mathematics.

At this point our students usually ask 'So, why Fortran? Why not a domain-specific language or something more "fashionable" like Java?' Let us come up with some justifying points. First, domain-specific languages might be relatively efficient in the specific domain they were created for, however, they are slow in other areas. Matlab, for example, is very good working with matrices of a regular or sparse nature, however, if you have ever tried to run some do-loops in Matlab, you will know what we mean. Second, when it comes to the more 'fashionable' languages like Java or C++, one has to always keep in mind that those languages are usually object-oriented and abundant. They are very good for creating software with graphical user interfaces, but not for running numerical maths code quickly and efficiently. As computational economics usually consists of at least 80 per cent numerical methods, we need a programming language that is efficient in executing do-loops and if-statements, calling functions and subroutines, and in allowing permanent storage of general codes that are frequently used. Fortran is a language that delivers all those features and much more; the only real alternatives we know would be C or a relatively new language called Julia. There is no real reason why one shouldn't use C or Julia instead of Fortran, it's just a matter of preference. Finally, Fortran was used a lot by engineers, numerical mathematicians, and computational economists during the 80s and 90s when a lot of optimizers and interpolation techniques were written. Fortunately, Fortran 90 is backward compatible with FORTRAN 77, i.e. all codes that are out there can easily be included in our Fortran programs.

All in all, it seems reasonable for us to have chosen a language matching all of the features necessary without any of the unnecessary extras that would make execution less efficient.

1.1.3 THE WORKINGS OF HIGH-LEVEL PROGRAMMING LANGUAGES

Low-level assembly languages that are basically used to program computers, microprocessors, etc. do not abstract from the computer's instruction set architecture. Therefore, the syntax of assembly languages depends on the specific physical or virtual computer used. In addition, assembly languages are quite complicated and even a small program requires many lines of computer code.

In opposition to that, high-level programming languages abstract from the system architecture and are therefore portable between different systems. As in all other languages, one has to learn the vocabulary, grammar, and punctuation, called the syntax, of that language. However, compared to real languages, the syntax of a high-level language can be learned within hours. In addition, the semantics, i.e. the meaning or interpretation of the different symbols used, usually follow principles that can be comprehended using pure logic.

As high-level languages abstract from the computer's instruction set architecture, one needs a *compiler*, sometimes also called *interpreter* that interprets the statements given in the program and translates them into a computer's native language, i.e. binary code. The way high-level programming works can be seen from Figure 1.1. First, the programmer has to write the source code in the respective high-level language and feed it to the compiler. The compiler then proceeds in two steps. In the *compilation* step it interprets the source code and creates a so-called object code file which consists of binary code. Binary code files usually have the suffix `.obj`. During this step, the compiler usually performs a syntax validation scan. If there are any syntactical errors, it produces error messages and stops the compilation process. In order to create an executable file, which has the suffix `.exe` in *Microsoft Windows*, the compiler now *links* the object code with all other libraries that are needed to run the program, e.g. external numerical routines. The resulting execution can now be sent to the operating system which advises the hardware to do exactly what the programmer wants.

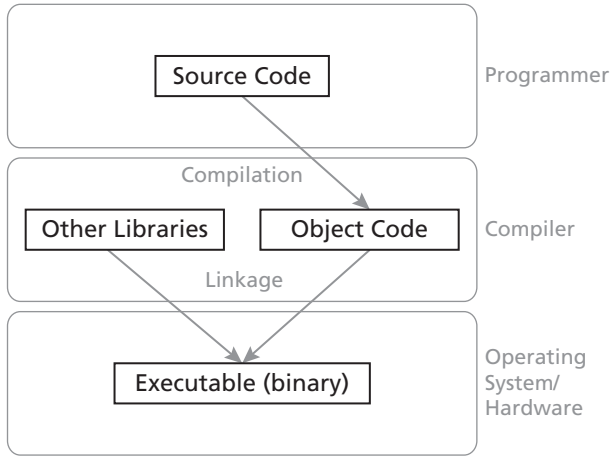


Figure 1.1 The way high-level compilers work

Program 1.1 Hello World

```
program HelloWorld
   write(*,*)'Hello World'
end program
```

1.1.4 FORTRAN COMPILERS FOR WINDOWS, MAC, AND LINUX

The website that accompanies this book

`www.ce-fortran.com`

suggests a freely available Fortran compiler that you can use for running the programs in this book. Just click on the menu item "Compiler" and choose your operating system. The website will guide you through the installation process. Finally, you will be advised how to run a program with your compiler. We therefore use the simplest programming example given in any computational textbook, namely the *Hello World* program shown in Program 1.1. A Fortran program always begins with the word **`program`** indicating that the program starts here. After that, one has to declare the program's name, which must follow the Fortran naming conventions, i.e.

1. the name can't have more than 31 characters
2. it must start with a letter
3. the other characters may be of any kind including symbols
4. capital letters may be used but the compiler is case insensitive
5. the name must not be a valid Fortran command.

Finally, the program ends with **`end program`**. In between these two statements, we can write the general program code which will be executed in an imperative way, i.e. the program starts with the first line and ends with the last. In the case of Program 1.1 there is only one statement that makes the program write `Hello World` in an unformatted way to the console.

1.2 **Imperative Fortran programs**

By knowing the basics of running a program in Fortran we can now proceed by writing the first imperative Fortran code. In order to describe the general structure of Fortran programs and give an overview of the basic features, we will restrict ourselves to imperative Fortran programs and leave the procedural component (i.e. subroutines and functions) to Section 1.3.

Program 1.2 Hello

```
program Hello

    ! declaration of variables
    implicit none
    character(len=50) :: input

    ! executable code
    write(*,*)'Please type your name:'
    read(*,*)input
    write(*,*)'Hello ',input

end program
```

1.2.1 THE GENERAL STRUCTURE OF FORTRAN PROGRAMS

In general, a Fortran program consists of a declarative part in which all variables needed for the execution are declared[1] and an executable part which gives instructions on what to do with all those variables. The executable part therefore consists of several *statements*, where each statement is usually given on one line. This is what the concept of imperative programming is basically about. The structure can easily be seen from Program 1.2, the interpretation of which is straightforward. Note, that after having written the first piece of executable code, we cannot declare a variable anymore. The statements in italic font in Program 1.2 starting with an exclamation point are comments. Those can be used to make it easier for the user to read the program, however, the compiler completely ignores them.

1.2.2 THE DECLARATION OF VARIABLES

Variables are used to store data during the execution of the program. Table 1.1 summarizes the five data types Fortran knows. Declaring the type of variable at the beginning of a program is not compulsory per se. Nevertheless, Fortan implicitly declares all variables that are used in the executable code as `integer`. This can cause severe problems when we run our program and forget to declare one variable that, for example, should be of `real*8` type. To prevent Fortran from implicit variable declaration by making declaration statements compulsory for any variable, we suggest always using the statement `implicit none` at the beginning of the declarative program part, see Program 1.2. Having specified `implicit none`, the compiler will tell us which variables are not yet declared and show an error message during the compilation step.[2] Program 1.3 shows some examples of variable declarations. The interpretation of the first declaration

[1] Technically speaking, the declaration of a variable induces the compiler to reserve some space in the memory that can be used to store data.

[2] Verify this by running Program 1.2 and deleting the line where `input` is declared. In the console window an error message will now appear when you try to execute the program.

Table 1.1 Different variable types in Fortran

Type	Explanation
logical	can only take the values **.true.** or **.false.**
integer	can store integer data in between $-2^{31}+1$ and $2^{31}-1$ (4-byte integer data).
real*8	stores real data according to the 8-byte data standard, i.e. in between $\sim -10^{308}$ and $\sim 10^{308}$. We suggest declaring all variables as **real*8**, not **real**, as the former usually produces more accurate results.
character	stores character data. If not stated otherwise, such a variable can just store one character. The declaration statement **character**(**len**=n) produces a variable that can store a whole string of characters of length n, where n has to be a positive integer number. Strings of characters will in the following just be called string.
complex	are used to store complex numbers. However, this will not be important in this book.

Program 1.3 Variable declarations

```
program VarDec

    ! declaration of variables
    implicit none
    logical :: logic
    integer :: a, b
    real*8 :: x, y1
    character :: one_char
    character(len=20) :: long_char

    real*8, parameter :: pi = 3.14d0
    integer, parameter :: n = 56

end program
```

statements should be clear from the above explanations. In addition to specifying the type of variable, we can also make it a **parameter** like in the last two statements of the program. This basically tells the compiler that the value of a variable should be fixed at a certain level and not be changed anymore. Specifying a parameter, we immediately have to declare the variables value, e.g. `pi = 3.14d0`.[3]

1.2.3 THE BASICS OF IMPERATIVE PROGRAMMING

After having declared the necessary variables, the first thing we would like to do is give values to those variables deemed necessary and then display these values on the console.

Reading and writing One way of assigning values to variables is making the user of the program type a value to the console. Program 1.4 explains how to do that. In this program,

[3] The `d0` after `3.14` tells the compiler that we are using double-precision variables. Always use `d0` when declaring the value of **real*8** variables. Try to find out what happens if you don't use it.

Program 1.4 Reading and writing

```
program ReadWrite

    ! declaration of variables
    implicit none
    integer :: a
    real*8 :: x

    ! executable code
    write(*,*)'Type an integer number:'
    read(*,*)a

    write(*,*)'Type a real number:'
    read(*,*)x

    write(*,*)a, x

end program
```

we do the following: we first declare an integer variable `a` and a real variable `x`. This is our declarative program part. In the executable part, we first ask the user to type an integer number. The command **`write`**`(*,*)` makes Fortran write something to the console without having a specific format, where the phrase `'Type an integer number:'` in apostrophes declares a text that should be displayed. The **`read`**`(*,*)a` statement induces the compiler to read a number from the console and assign it to the variable `a`. The first of the two stars in parentheses therefore tells the compiler from which location it should read. A `*` denotes the console as reading location. Reading from a file is also possible, however that will be explained later. The second star defines the format in which the number or text will be given. On the console, we always use `*`, which means there is no specific format. The compiler will then automatically check whether the number typed by the user is in the range of the variable we want to assign a value to, e.g. `a`.[4] The same explanation applies to the statements concerning the real variable. Finally, we write the values of `a` and `x` to the console. Note that we can write several variables at once by just separating them with a comma. The same applies to the **`read`** statement.

Formatters In order to display variables in a formatted way, we use formatters. An example of how to use formatters is given in Program 1.5. This program does basically the same thing as Program 1.4, however it also prints the variables `a` and `x` in a formatted way on the console. The formatter, which is shown in apostrophes and parentheses, replaces the second star in the **`write`** statement, and states that we want to write an integer of maximum length three digits, two blank spaces and a real number with a maximum of 10 digits (including the decimal point), where six digits are reserved for the decimal places. Run the program and verify that this is true. There is a formatter

[4] Verify this by running Program 1.4 and typing 12.5 when you are asked to type an integer number. The compiler will then throw up an error message.

Program 1.5 Formatters

```
program Formatter

    ! declaration of variables
    implicit none
    integer :: a
    real*8 :: x

    ! executable code
    write(*,*)'Type an integer number:'
    read(*,*)a

    write(*,*)'Type a real number:'
    read(*,*)x

    write(*,'(i3, 2x, f10.6)')a, x

end program
```

Table 1.2 Formatters for different variable types

Formatter	Explanation
`l2`	value of a logical (T or F) with a total width of 2
`i4`	integer of a maximum of 4 digits
`f12.4`	real of a maximum of 12 digits (including the decimal point) 4 digits are reserved for decimal places
`a`	string of arbitrary length
`x`	a blank space

for each type of variable usually consisting of a letter indicating the type of variable that should be written and a number that defines the width of the output. For real variables, there is a second number specifying the number of decimal places. Table 1.2 summarizes common formatters. Putting a number in front of a formatter means that we want the compiler to write the same type of data several times. Multiplying a whole set of formatters can be realized by using brackets. Consider for example the following statement:

```
write(*,'(a,2x,2(f4.2,2x),a,i2)')'hello',123.456,1.544,'heLLO',12
```

The formatter tells the compiler to first write a string, then two blank spaces, twice a real number of a total of four digits with two decimal places followed by two blank spaces, again a string, and finally an integer with two digits. Note that there is one problem that arises with this statement. If we take a look at the data that should be written, not given through variables however directly typed into the written statement, we see that the first real number will be too big for its formatter. Fortran will therefore just display four star symbols indicating this overflow. Hence, the console output will be

```
hello **** 1.54 hELLO12.
```

You can verify this by writing a program that just consists of the statement above. Then change the formatting code and verify the results.

Program 1.6 Value assignment through arithmetics

```
program Arithmetics

    ! declaration of variables
    implicit none
    real*8 :: a, b, c, d

    ! executable code
    a = 6d0
    b = 2.5d0
    c = exp(b) + a**2*sqrt(b)
    d = max(a,b)*sign(b, a)/mod(9d0,5d0) + abs(c)

    write(*, '(4f10.4)')a,b,c,d

end program
```

Value assignments and calculations We can assign values to variables by stating `<variable> = <value>` as shown above. However, `<value>` does not necessarily have to be typed directly as a number, but can also be the result of some arithmetic operations. Program 1.6 illustrates how to use arithmetic to define variable values. Beneath the standard math operators `+`, `-`, `*`, `/`, and `**` (meaning to the power of), Fortran comes with several intrinsic functions like **`exp`**, **`log`**, etc. a summary of which can be found on our website. In Program 1.6, we first define four variables `a`, `b`, `c`, and `d` as of type **`real*8`**. In the first two lines, we assign the values 6 and 2.5 to variables `a` and `b`, respectively. Up to that point, `c` and `d` are still undefined. We then calculate:

$$c = \exp(b) + a^2 \cdot \sqrt{b} \quad \text{and} \quad d = \max(a, b) \cdot |b| \cdot \operatorname{sgn}(a)/(9 \bmod 5) + |c|.$$

Note that Fortran uses the standard PEMDAS precedence rule. In the last line of executable code we then let the compiler print the values of `a`, `b`, `c`, and `d` on the console, where every variable should have four decimal places. Run the program and compare the results to the ones you obtain by calculating the values of `c` and `d` by hand.

1.2.4 CONTROL FLOW STATEMENTS

In Section 1.2.3 we showed how to declare variables and write simple imperative codes in order to assign values. However, writing line after line of value assignments isn't the only thing we can do in Fortran. *Control flow statements* allow for the execution of conditional statements, i.e. the statement will only be executed, if a certain condition is true, and repeated statements, i.e. the same statement will be executed several times. The concept of control flow statements can easily be represented in a *control flow diagram*. Figure 1.2 shows such a diagram for a simple program. It's interpretation is quite intuitive. The program starts where the large black dot is: We first should read an integer number from the console. Then, we should check whether $i > 0$. There now are two conditional statements. If $i \leq 0$, we write an error message that says "i should be greater 0" and the

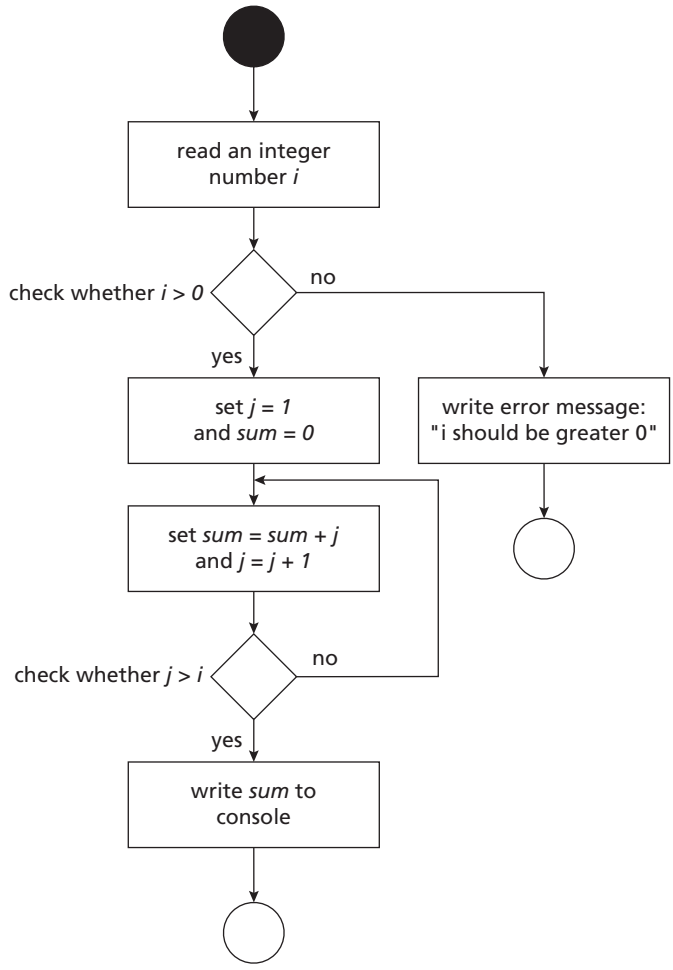


Figure 1.2 A control flow diagram

program stops afterwards, indicated by the circle. If $i > 0$ there is a series of conditional statements which basically sum up all integer numbers from 1 to i. The three statements after the first condition form what we call a *do-loop*. Using a do-loop, we can repeat the same statements, namely `sum=sum+j` and `j=j+1` several times.

If-statements and logical expressions An if-statement is needed to check whether some condition is true or false. The general syntax of an if-statement is

```
if(<condition1>)then
    <executable statements>
elseif(<condition2>)then
    <executable statements>
    [......]
else
    <executable statements>
endif
```

The word **if** indicates that we want to check a certain condition. The condition has to be given in parentheses followed by the word **then**. The compiler will now check whether the condition holds and executes the code that stands within the **then** and the next control flow statement. This next statement can be of three different types. If an if-statement is followed by an **elseif** and `<condition1>` is false, the compiler will automatically check whether `<condition2>` is true. You can line up **elseif** as much as you want. The compiler will then check in hierarchical order, i.e. if the first condition is true, the statements within the first **if** and the first **elseif** will be executed and the compiler jumps to the point where **endif** ends the whole control flow structure. If the first condition is false and the second condition holds, the second set of executable statements will be executed and the compiler again jumps to the **endif** statement etc. An **else** without an **if** indicates which code fragment should be used if none of the above conditions holds, an **endif** tells the compiler where the whole if-construct ends. Of course, **elseif** and **else** statements are optional whereas the **endif** is compulsory. Program 1.7 illustrates the behaviour of if-statements. The program should be self-explanatory. Guess which condition will be true in this case. Run the program and verify it. Then change the value given to `a` and see what happens.

Conditions in the if-statement can be any kind of *logical expression*. A logical expression is an expression that is either true or false. Logical expressions can be created by using relational operators like 'is equal to', 'is not equal to', 'is greater than', etc. Table 1.3 gives an overview of relational operators in Fortran. Beneath the Fortran 90 relational operators, the table also shows the respective FORTRAN 77 operators which are still valid in Fortran 90. In the following, we will only use the Fortran 90 commands. Nevertheless, you can find the old operators in a lot of codes especially in older textbooks. An example for a logical expression is `a + b <= c`. Note that relational operators bind less than mathematical ones, i.e. the compiler will first evaluate all mathematical parts of logical

Program 1.7 If-statements

```
program IfStatements

    implicit none
    integer :: a

    ! initialize a
    a = 1

    ! check for the size of a
    if(a < 1)then
        write(*,'(a)')'condition 1 is true'
    elseif(a < 2)then
        write(*,'(a)')'condition 2 is true'
    elseif(a < 3)then
        write(*,'(a)')'condition 3 is true'
    else
        write(*,'(a)')'no condition is true'
    endif

end program
```

Table 1.3 Relational operators in Fortran

Fortran 90	FORTRAN 77	Explanation
==	**.EQ.**	is equal to
/=	**.NE.**	is not equal to
>	**.GT.**	is strictly greater than
>=	**.GE.**	is greater than or equal to
<	**.LT.**	is strictly lower than
<=	**.LE.**	is lower than or equal to

Table 1.4 Logical operators in Fortran

Operator	Explanation
.and.	true if both expressions are true
.or.	true if at least one expression is true
.eqv.	true if both expressions have the same value
.neqv.	true if both expressions have different values

Table 1.5 Results of logical connections

a	b	a**.and.**b	a**.or.**b	a**.eqv.**b	a**.neqv.**b
.true.	**.true.**	**.true.**	**.true.**	**.true.**	**.false.**
.true.	**.false.**	**.false.**	**.true.**	**.false.**	**.true.**
.false.	**.true.**	**.false.**	**.true.**	**.false.**	**.true.**
.false.	**.false.**	**.false.**	**.false.**	**.true.**	**.false.**

statements and then the relational ones. Let, e.g. `a = 3`, `b = 5`, and `c = 6`. The logical expression will therefore be evaluated in the following way:

```
a + b <= c   ⇒   3 + 5 <= 6   ⇒   8 <= 6   ⇒   .false.
```

The value of the logical expression is therefore false for the above values of `a`, `b`, and `c`. Logical expressions can be linked by means of *logical operators*, a summary of which is given in Table 1.4. In this table, the operators are listed by importance for the compiler. Hence, **.and.** is the most binding operation among the logical expressions and will always be executed first. On the other hand, **.neqv.** is the less binding constraint. Note that you can make, e.g. an **.or.** more binding than an **.and.** by putting the respective expression in parentheses. The pattern therefore is the same as with adding and multiplying. The result of logical connections can be seen from Table 1.5. In addition to connecting logical expressions, we can also negate an expression by simply typing **.not.**<expression> with any kind of logical expression. Consider for example the expression

```
x <= 2 .and. (y <= 3 .or. z < 5) .or. .not. (y <= 3 .and. z < 5)
```

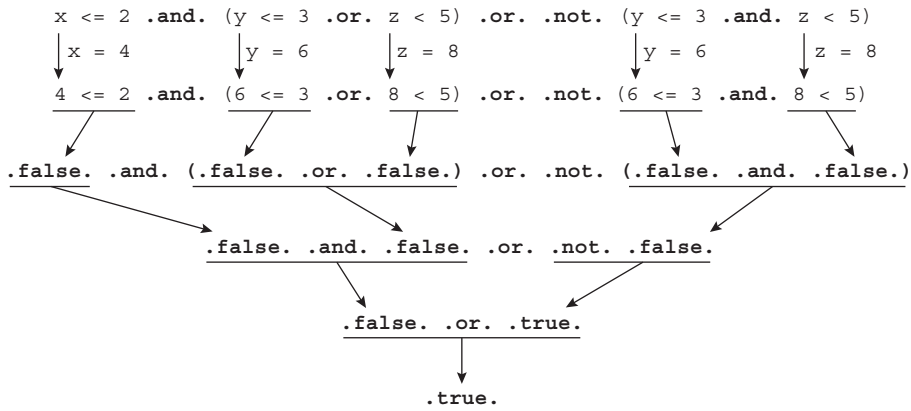


Figure 1.3 Evaluation process of logical expression

with `x = 4`, `y = 6`, and `z = 8`. The evaluation process of this expression can be seen from Figure 1.3. Verify this result by writing a program that defines the three variables `x`, `y`, and `z` and writes the result of the logical expression given above to the console.

Do-loops A do-loop is a construct that allows you to repeat a set of statements several times. An example of how to use do-loops is given in Program 1.8. This program shows three examples of do-loops. The first loop increments the integer variable `j` from 1 to 10 and writes its value to the console in every iteration. The general syntax for a do-loop that directly increments a variable is

```
do <variable> = <beginning>, <ending>, <stepsize>
    <executable statements>
enddo
```

The word **`do`** indicates that we want to start a loop. We then have to state which variable we want to increment or decrement during the loop. `<beginning>` and `<ending>` then tell the compiler what the starting value is and at which value to stop. `<stepsize>` is optional and specifies which step size to use. Finally, we write down the code that should be executed for every iteration of the loop. The **`enddo`** tells the compiler where the executable code of the loop ends. An example for a different step size is given in the second loop. We hereby let `j` take the values from 10 to 1 downwards. The last do-loop shows an alternative formulation. We hereby first initialize `j` at value 1. We then write a general loop without a specification statement for a variable. Within the loop we let the compiler write the value of `j` to the console and add 1 to `j`. Note that the command `j = j + 1` simply takes the current value of `j`, adds one to this value and directly stores the result in `j` again. The if-statement in the loop checks, whether `j` has exceeded a value of 10. If this is the case, the exit command makes the compiler leave the current do-loop.

Program 1.8 Do-loops

```
program DoLoops

    implicit none
    integer :: j

    ! perform a do-loop for j = 1 to 10
    do j = 1, 10
        write(*,'(i3)')j
    enddo

    ! write a blank line
    write(*,*)

    ! perform a do-loop for j = 10 to 1
    do j = 10, 1, -1
        write(*,'(i3)')j
    enddo

    ! write a blank line
    write(*,*)

    ! alternative do-loop
    j = 1
    do
        write(*,'(i3)')j
        j = j + 1

        ! exit the do-loop
        if(j > 10)exit
    enddo

end program
```

The statement `if(j > 10)exit` is thereby just a shortcut for that can be used if there is just one executable statement associated with the if-statement.

```
if(j > 10)then
    exit
endif
```

Knowing the concepts of if-statements and do-loops, we can now implement the program specified by the control flow diagram in Figure 1.2. Program 1.9 shows how to do that.

1.2.5 THE CONCEPT OF ARRAYS

An array is a construct that allows storage of a whole set of data under one variable name. Arrays, like variables, are defined in the declarative part of a program. Typical examples of array declarations are

Program 1.9 Summing up in do-loops

```
program SummingUp

    implicit none
    integer :: i, j, sum

    ! read the variable
    write(*,'(a)')'Type an integer variable that is > 0'
    read(*,*)i

    ! check whether i > 0
    if(i > 0)then

        ! initialize sum at 0
        sum = 0

        ! do the summing up
        do j = 1, i
            sum = sum + j
        enddo

        ! write the result to the console
        write(*,'(a,i3,a,i10)')'The sum of 1 to ',i,' is ',sum

    else

        ! write the error message
        write(*,'(a)')'Error: i should be greater than 0'
    endif

end program
```

```
real*8 :: a(10)
integer :: b(12, 5, 16)
real*8 :: c(0:12, -3:5)
```

The numbers in parentheses, after the array's name, indicate its dimension. `a`, e.g., is an array of dimension 1 with 10 entries in the only dimension. `b` is of dimension 3 with 12, 5, and 16 elements in the three dimensions, respectively. Last but not least, `c` is an array of dimension 2 with 13 and 9 elements in the two directions. However, the index of dimension 1 starts at 0 and that of dimension 2 at −3. The single elements of an array can be accessed in the executable code by just saying, e.g., `a(5)`, `b(1, 3, 16)`, or `c(0, -2)`. The statement `b(1, 3, 16)` then returns the value of the array at positions 1, 3, and 16 in the three dimensions, respectively. Note, that if not defined otherwise, an array's index always starts with 1. In the case of `c`, however, we made the array index start with 0 and go up to 12 in the first dimension, meaning a total size of 13.

An example of how to deal with arrays is given in Program 1.10. In the declarative part of the program, we first define two arrays of same size, the indices of which start at 0 and go up to 10, as well as an integer variable that serves as counter for our do-loops. In the first step of our executable program part we initialize the values of `x`. We do this by running a do-loop over `j` from the first to the last array element, where we let `x` be

Program 1.10 Handling arrays

```
program Arrays

    implicit none
    real*8 :: x(0:10), y(0:10)
    integer :: j

    ! initialize x and calculate y
    do j = 0, 10
        x(j) = 1d0/10d0*dble(j)
        y(j) = exp(x(j))
    enddo

    ! output table of values
    write(*,'(a)')'          X          Y'
    do j = 0, 10
        write(*,'(2f10.3)')x(j), y(j)
    enddo

end program
```

equidistant points on the interval $[0, 1]$.[5] Having set `x`, we can go over to calculating `y`. We want `y` to be the exponential function evaluated at the different values of the array `x`. This again is done by means of do-looping. Finally, we let Fortran print the values of `x` and `y` to the screen.

Nevertheless, do-looping is not the only way to work with arrays. Let's consider the example given in Program 1.11. This example contains several new features. We start with introducing a parameter `n` that defines the size of any array in this program. Note that `n` must have the parameter property. If not, the declaration of arrays will not work. Next, we define three different arrays, two of dimension 1 and one of dimension 2. We initialize the array `x` as in Program 1.10. Next, we want the array `y` to have the entries of `x` and add 1 to each entry. We can do this again with a do-loop as seen in the program before, however there is a shortcut to that. If we use the notation `y(:) = x(:) + 1d0`, it has the same effect. This statement basically tells the compiler to take all entries of `x`, add 1 to each of them and store the results in `y`. The `(:)` therefore indicates that we want to do something with each entry of an array. We can also perform an action with only part of an array. If we for example stated `y(1:6) = x(0:5) + 1d0`, the compiler would take the elements of `x` at the indices from 0 to 5, add 1 to each of them, and store them in `y` at the positions 1 to 6. Next we calculate the values of `z` by means of a separate do-loop for each of the dimensions of `z`. Last, we output a two-dimensional table of values for `z`. We therefore let the compiler print a headline that consists of the string `'   X/Y | '` and all the values of `y` followed by a blank line. We then output one line for each set of numbers

[5] In the initialization statement for `x`, the term **`dble`**`(j)` converts the integer value of `j` to a **`real*8`** value. This should always be done when using integers in calculating **`real*8`** values. See the exercise section for an explanation.

Program 1.11 Alternative array handling

```
program AlternativeArrays

    implicit none
    integer, parameter :: n = 8
    real*8 :: x(0:n), y(0:n), z(0:n, 0:n)
    integer :: j, k

    ! initialize x
    do j = 0, n
        x(j) = 1d0/dble(n)*dble(j)
    enddo

    ! give y the values of x plus 1
    y(:) = x(:) + 1d0

    ! calculate z
    do j = 0, n
        do k = 0,n
            z(j, k) = x(j)**2 + y(k)
        enddo
    enddo

    ! output table of values
    write(*,'(a,9f7.2)')'    X/Y | ', y(:)
    write(*,'(a)')'        | '
    do j = 0, n
        write(*,'(f7.2,a,9f7.2)')x(j),' | ', z(j, :)
    enddo

end program
```

in `z` that has the same index in the first dimension. Unfortunately, we can't write `n` into the formatter, as the formatter is a string. Hence, we have to manually type the number 9 (which corresponds to the length of the array `y`), as creating a general formatter with parameter `n` would be far beyond the scope of this introduction.

1.3 Subroutines and functions

Subroutines and *functions* can be used to store codes that are frequently used within one program. While a subroutine just executes an imperative code, a function is a construct that, like a function in maths, receives some input value and calculates a return value.

Subroutines Program 1.12 demonstrates how to use subroutines in a Fortran program. Subroutines, as well as functions, are defined after the main program code. The part of the program which contains subroutines and functions is separated from the main code by means of the **`contains`** statement. We declare a subroutine by typing the keyword

Program 1.12 Subroutines

```
program Subroutines

    implicit none
    real*8 :: a, b, c, d

    a = 3d0
    b = 5d0

    ! call subroutine
    call addIt(a, b)

    ! redefine values
    c = 10d0
    d = 2d0

    ! call subroutine again
    call addIt(c, d)

! separates main program code from subroutine and functions
contains

    subroutine addIt(a, b)

        implicit none

        ! input arguments
        real*8, intent(in) :: a, b

        ! other variables
        real*8 :: c

        ! executable code
        c = a + b
        write(*,'(2(f8.2,a),f8.2)')a,' + ',b,' = ',c

    end subroutine

end program
```

subroutine and a name afterwards. The name, `addIt` in the case of Program 1.12, must follow the Fortran naming conventions as described in Section 1.1.4 and must neither correspond to the program name nor any name of a variable declared in the main program. After having specified the name, we can tell Fortran which communication variables the subroutine receives from the main program.[6] In our case, we specified two input variables, `a` and `b`, the sum of which will be calculated and written to the console. Note that communication variables do not have to be simple input arguments per se, they can also be arrays or, as we will see later, functions or subroutines. If we pass on a variable from the main program and change its value within the subroutine, the change will also be adopted by the main program.[7] The structure of a subroutine is now pretty much the

[6] You do not need to specify any communication variable. Just type empty parentheses in this case.

[7] We recommend always using communication variables to pass data on to a subroutine or function. Yet, it would generally be possible to use variables that are declared in the main program within a subroutine. Verify

same as that of a main program. We first have a declarative part in which we define all variables used in the subroutine, including communication variables. In addition, we can regulate whether our communication variables should be of input or output type. This is done via an `intent` statement. If a variable's intent is `in`, we are not allowed to change its value throughout the subroutine.[8] If the intent is `out`, the variable must be given a value within the subroutine. Specifying `inout` has the same effect as not declaring any `intent`. After having declared all variables, the executable part of our subroutine specifies what to do with them. We can use any kind of executable statements we know from our main programs.

When it comes to using a subroutine in a main program, we use the keyword `call` followed by the name of the subroutine that should be called.[9] In addition, we have to specify which variables of the main program to pass on to the subroutine, where the number and type of variables must be exactly the same as those specified in the subroutine's declaration. In our case, we use the subroutine with different variables twice.

Functions Functions are equally easy to use. Program 1.13 shows an example. Declaring a function is very similar to declaring a subroutine. However, in addition to specifying the type and intent of communications variables, we also have to state the function's return value. This is done using a type declaration with the function's name. The name will therefore serve as a regular variable throughout the executable part of the function. The value given to this variable finally is the return value of the function, which in our example corresponds to the sum of `a` and `b`. Having specified the function we now can use it in our main program. Since the function has a return value, we do not call it via a `call` statement like a subroutine, but instead assign the return value directly to a variable, `res` in our case. `res` now contains our function value, i.e. the sum of `a` and `b`. Finally, we write the result to the console.

Passing arrays to functions or subroutines Sometimes one would like to pass an array to a function or subroutine. This can be done in two ways, see Program 1.14. If we take a look at the function declaration, we see that communication variable `a` is specified like we would specify a regular array in a main program. The declaration statement therefore tells the compiler that it should expect an array of dimension 1 and length 2 to be passed to the function. In the case of `b`, we used the so-called assumed-shape statement.

this by deleting the input variables `a` and `b` from the subroutine and calling up the subroutine without input variables. How does the output change compared to Program 1.12? Note that when a variable is declared in a subroutine that has the same name as a variable in the main program, the subroutine will always use the 'local' variable, not the 'global' one from the main program.

[8] Verify this by trying to change the value of `a` in the above subroutine. When compiling, there now should be an error in the output window saying that this is not allowed.

[9] Subroutines do not necessarily have to be called up by a main program, but can also be used by other subroutines.

Program 1.13 Functions

```
program Functions

    implicit none
    real*8 :: a, b, res

    a = 3d0
    b = 5d0

    ! call function
    res =  addIt(a,b)

    ! output
    write(*,'(2(f8.2,a),f8.2)')a,' + ',b,' = ',res

contains

    function addIt(a, b)

        implicit none

        ! input arguments
        real*8, intent(in) :: a, b

        ! function value
        real*8 :: addIt

        ! executable code
        addIt = a + b

    end function

end program
```

Typing `a:` instead of a valid integer number, we allow the length of `b` to be anything. The compiler therefore will just expect an array of dimension 1, but with an arbitrary length. However, we can refer to `b`'s length, which will be calculated on-the-fly during runtime, by typing **`size`**`(b)`.[10] We consequently define our return value as an array of two dimensions, the first of which is of size 2 and the second of the same size as `b`. The executable code of our function should then be straightforward.

In the main program, we now specify three arrays. The two input arrays of dimension 1 and the array that should store the result of our function. This array must have dimension 2 with the size of `a` in dimension 1 and that of `b` in dimension 2. We then initialize the arrays `a` and `b`. There is a shortcut to do this by saying `a(:) = (/<values>/)`, where `<values>` is just a list of initialization values separated by commas. Finally, we can call our function, assign the return value to `res`, and write the result to the console.

[10] If you declare a multidimensional array, you can refer to the length into any dimension by stating **`size`**`(b, <`**`dim`**`>)` where <**dim**> is just an integer number declaring the dimension.

Program 1.14 Passing arrays to functions

```
program ArrayFunc

    implicit none
    real*8 :: a(2), b(5), res(2,5)

    a(:) = (/3d0, 4d0/)
    b(:) = (/1d0, 2d0, 3d0, 4d0, 5d0/)

    ! call function
    res =  addIt(a,b)

    ! output
    write(*,'(5f8.2/5f8.2)')res(1,:),res(2,:)

contains

    function addIt(a, b)

        implicit none

        ! input arguments
        real*8, intent(in) :: a(2), b(:)

        ! function value
        real*8 :: addIt(2, size(b))

        ! local variables
        integer :: j, k

        ! executable code
        do j = 1, 2
            do k = 1, size(b)
                addIt(j,k) = a(j) + b(k)
            enddo
        enddo

    end function

end program
```

1.4 Modules and global variables

A module is a construct that allows storage of frequently used codes and variables in a separate location. The codes and variables can then be used by different programs.

1.4.1 STORING CODE IN A MODULE

Module 1.15m gives an example of a simple module which contains a function to calculate the volume of a sphere.[11] The structure of a module is similar to that of a regular program,

[11] The respective program is called `prog1_15m.f90` in our program database.

Module 1.15m Storing code in modules

```
module Volume

    implicit none

    ! module parameter
    real*8, parameter :: pi = 3.14159265358d0

! separates variable declarations from subroutine and functions
contains

    ! for calculating volume of a sphere
    function vol(r)

        implicit none

        ! input and output variables

        real*8, intent(in) :: r
        real*8 :: vol

        ! calculation
        vol = 4d0/3d0*r**3*pi

    end function

end module
```

however, a module begins with the keyword **`module`** and ends with **`end module`**. The variable declaration part again starts with an **`implicit none`** statement followed by any variable declaration. In our case, we defined a parameter of type **`real*8`** that gives us the value of π. In contrast to a main program, a module does not contain any directly executable codes. Nevertheless, we can store functions and subroutines within a module. The statement **`contains`** again indicates that we want to start the part of the module where those are declared. Functions or subroutines are declared in the same way we have seen before, see Programs 1.12 and 1.13. In the case of Module 1.15m we declared a function `vol` that calculates the volume of a sphere with radius `r`. The radius is given as an input argument to the function via the **`real*8`** variable `r`.

We can now write a very simple program that uses the module `Volume`. There are multiple ways of doing so. The easiest is to include the module code into the main program. This is done through an **`include`** statement, see Program 1.15. The include statement tells the compiler that we want to use the main program together with the module that is stored in the file `prog1_15m.f90`.[12] When we run the program, the compiler will automatically compile the module prior to compiling the program and therefore make it ready for usage. Including a module this way ensures that any change we

[12] We can also store several modules in the file `prog1_15m.f90`, see next example.

Program 1.15 Calling module code from Module 1.15m

```
include "prog01_15m.f90"

program Sphere

    ! import variables and subroutines from module Volume
    use Volume

    implicit none

    write(*,'(f12.4)')vol(1d0)

end program
```

make in the module file will immediately be recognized by the compiler.[13] The `include` statement, however, only tells our compiler that it should compile the code we have stored within our module file. In the program `Sphere` itself we then still have to indicate that we want to use the module `Volume` by typing `use Volume` before any variable declaration statement. We can now access any declared variable, function, or subroutine of that module. Hence, we can write `vol(1d0)`, i.e. the volume of a sphere with radius 1, to the console.

1.4.2 THE CONCEPT OF GLOBAL VARIABLES

A global variable is a variable that is present in any part of a program, i.e. in the main program as well as any subroutine and function. Typical examples of global variables are model parameters. One way to realize this concept is to pass a variable we want to be global to a subroutine and function of our main program. However, if we have larger programs this can become messy. Therefore, we can use modules to make life easier.

Consider for example Module 1.16m, which actually consists of two separate modules. The first module `Globals` is pretty simple, as it contains no subroutines or functions. It just declares two variables, `beta` and `eta` denoting time preference and risk aversion in a simple two-period life-cycle model, where households' utility function is given by

$$u(c_1, c_2) = \frac{1}{1-\eta} \cdot c_1^{1-\eta} + \frac{\beta}{1-\eta} \cdot c_2^{1-\eta}. \tag{1.1}$$

This utility function is specified in the second module `UtilFunc`. The module only contains a function `utility` that takes as input variables values for c_1 and c_2 and just

[13] An alternative way of handling the module file would be to precompile it using the *Compile* button in the *Build Toolbar*. However this way the compiler sometimes does not recognize changes that have been made within the module file.

Module 1.16m A module to store global variables

```
module Globals

    implicit none

    ! time preference
    real*8 :: beta

    ! risk aversion
    real*8 :: eta

end module


module UtilFunc

    implicit none

contains

    ! a utility function
    function utility(c1, c2)

        use Globals

        implicit none
        real*8, intent(in) :: c1, c2
        real*8 :: utility

        utility = 1d0/(1d0-eta)*c1**(1d0-eta) &
                          + beta/(1d0-eta)*c2**(1d0-eta)

    end function

end module
```

calculates the value of lifetime utility. Note that this function uses the module `Globals` and especially the parameters defined therein.[14]

Suppose now, we want to write a program with which we can calculate the utility function for different combinations of c_1 and c_2 that satisfy the budget constraint

$$c_1 + c_2 = 1,$$

where we assumed the present value of income and the interest rate to be 1 and 0, respectively. An example of such a program could be Program 1.16. Again we have to tell the compiler in which file we have stored our modules by means of the **`include`** statement. In the main program, before declaring any program specific variables, we tell the program to use the `Globals` module that contains our global variables `beta` and `eta` as well as the `UtilFunc` module that contains the utility function. We then define some

[14] It is important that the module `Globals` is defined prior to the module `UtilFunc`, since the latter uses the former. If the modules were arranged in reverse order, an error statement would tell us that `UtilFunc` can not find the module `Globals`.

Program 1.16 A program that uses global variables

```
include "prog 01_16m.f90"

program CalcUtil

    use Globals
    use UtilFunc

    implicit none
    real*8 :: c1, c2, util
    integer :: j

    ! initialize parameters
    beta = 0.9d0
    eta = 2d0

    ! calculate utility for different consumption pairs
    ! between 0.3 and 0.7
    do j = 0, 20
        c1 = 0.3d0 + (0.7d0-0.3d0)/20*dble(j)
        c2 = 1d0-c1
        util = utility(c1, c2)
        write(*,'(3f10.4)')c1, c2, util
    enddo

end program
```

variables that are used later on. Afterwards, we can initialize our global model parameters `beta` and `eta`. Those parameter values will consequently be stored in the module `Globals` and then be used by the function `utility`. Finally, we calculate the utility function resulting from different combinations of `c1` and `c2` that satisfy the household budget constraint by means of the function `utility`.

1.5 Installing the toolbox

This textbook comes with a toolbox, i.e. a collection of subroutines and functions, that will frequently be used in this book mostly to solve numerical problems. These subroutines and functions will be discussed in detail in the next chapters. The toolbox and the necessary programs should automatically have been installed together with the compiler. However, you can also download and install the toolbox directly from our website

`www.ce-fortran.com`

Just click on the menu item *Toolbox* and the website will guide you through the installation process. This menu item is also the place to look for updates. The toolbox requires the installation of the program *GNUPlot*. GNUPlot is a drawing program that can be used to draw graphs on Windows, Mac, and Linux machines. The toolbox contains some

interface routines that help you with that. Two things to note on the toolbox: First, we provide the toolbox in plain text format. So if you want to take a deeper look at the subroutines provided therein feel free to do so. Second, when you install the toolbox it will be compiled with the compiler you installed on your computer. By doing so it is ensured that the compiler will always be able to access the toolbox routines without problems.

1.6 Plotting graphs with the toolbox and GNUPlot

We use program GNUPlot for plotting graphs in Fortran as Fortran itself does not provide any plotting routines. Plotting graphs is, however, very simple, as the toolbox provides respective interface routines that transfer your data into GNUPlot graphs. In this section, we demonstrate how to plot both two- and three-dimensional data.

1.6.1 TWO-DIMENSIONAL PLOTTING

Program 1.17 shows how to plot graphs with GNUPlot using the toolbox. Of course, GNUPlot has to be installed on your computer. In order to use any subroutine or function located in the toolbox, we have to include the toolbox by stating `use toolbox`. We then declare two arrays, one for our x-axis data and one for the y-axis data, respectively. We initialize `x` at equidistant points on the interval [0, 1] via the first do-loop. Afterwards, we calculate the y-data we want to plot, which in the first case is equal to x^2. Note that we used the shortcut method discussed in Section 1.2.5. In order to tell the toolbox to include our x and y data in a plot, we call the subroutine `plot`. The minimum requirement for using `plot` is to provide two equal size arrays of type `real*8`, the first containing the x-axis data and the second the y-axis data. The subroutine `plot` can be called with many more arguments which we will discuss later. Having used the subroutine `plot`, our x and y data is included in the plot we want to make. In order to actually draw the data with GNUPlot we have to call the subroutine `execplot`. When we call this subroutine, a window will appear on the screen displaying our x and y data as in Figure 1.4. The execution of our main program will pause. In order to close the plot window and let the program continue, you have to click on the console window and press **RETURN**.

We can also plot several graphs into one window. This is done in the second part of the program. We therefore first calculate the y data as $\sqrt{x}$ and write it to our next plot by calling the subroutine `plot`. This time we include one of the *optional arguments* of the subroutine `plot`. These optional arguments are used by stating their name, in our case `legend`, and by specifying a value, in our case the name of the graph that should appear in the legend of the plot, namely `'square root'`. All the optional arguments of `plot` and

Program 1.17 Plotting graphs with the toolbox

```
program Plotgraphs

    use toolbox

    implicit none
    real*8 :: x(0:100), y(0:100)
    integer :: i1

    ! Initialize x values
    do i1 = 0, 100
        x(i1) = 1d0/100d0*dble(i1)
    enddo

    ! Calculate plot data
    y = x**2
    call plot(x, y)

    ! execute plot program
    call execplot()

    ! Calculate data for roots
    y = x**(1d0/2d0)

    ! you can specify a legend entry in the plot as follows
    call plot(x, y, legend='square root')

    ! the same for a cubic root
    y = x**(1d0/3d0)
    call plot(x, y, legend='cubic root')

    ! execute plot program and give the plot a title
    call execplot(title='Roots')

    ! plot has many more options that are specified here
    y = x**(1d0/2d0)
    call plot(x, y, color='green', linewidth=3d0, marker=2, &
            markersize=0.7d0,  noline=.false., legend='square root')

    y = x**(1d0/3d0)
    call plot(x, y, color='#5519D6', marker=5, markersize=1.2d0, &
              noline=.true., legend='cubic root')

    call execplot(xlim=(/0d0, 1.1d0/), xticks=0.1d0, &
           xlabel='x-Axis', ylim=(/0d0, 1.4d0/), yticks=0.2d0, &
           ylabel='y-Axis', title='Roots', legend='rs', &
           filename='testplot', filetype='eps', output='testdata')

end program
```

`execplot` are discussed in detail below. Having included the x and y data for the square root we can proceed and calculate the data for a cubic root. We can include this data into the graph by just calling the plot statement again. In total the toolbox would allow you to place up to 1,000 graphs in one plot. Having added all the relevant plot data to our graph, we can again make GNUPlot draw our required lines by calling `execplot`. This time we specify the optional argument `title` which will define the title of our plot. The resulting plot will look like the one in Figure 1.5.

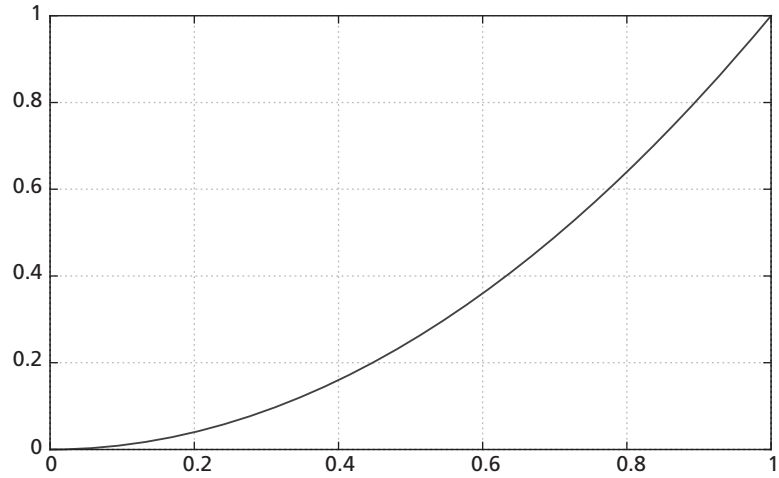


Figure 1.4 Plotting a graph with GNUPlot

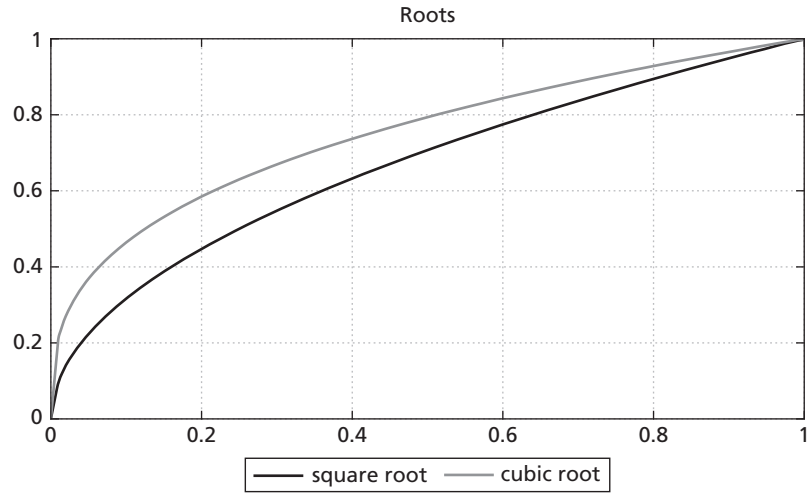


Figure 1.5 Plotting a graph with GNUPlot (2)

Note: The top line drawn in the graph is the cubic root, the bottom line is the square root

The last part of the program repeats the plot of square and cubit root, but this time it applies all optional arguments to the subroutines `plot` and `execplot`. In the subroutine `plot` you can specify which colour should be used for the graph, how thick the line should be, whether the data points you supplied should be marked with a marker and how this marked should look like. The `noline` statement tells GNUPlot whether it should use a connection line between the supplied data points or only show the markers. Finally the `legend` statement specifies the legend entry for the graph. The subroutine `execplot`

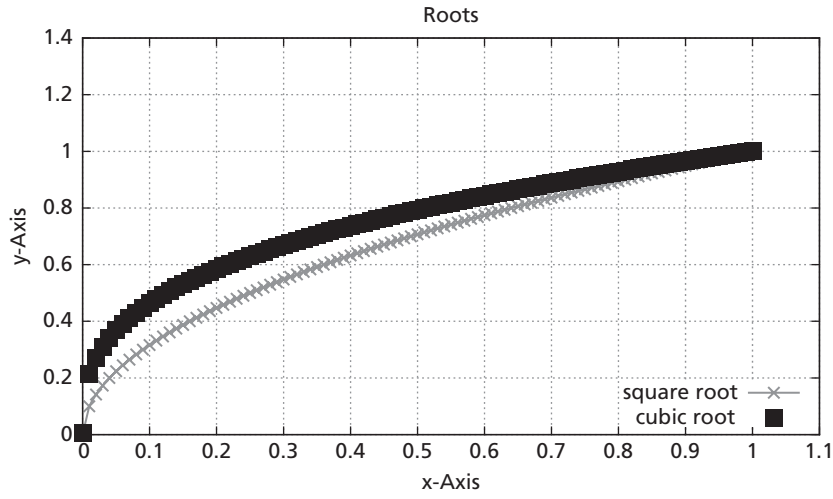


Figure 1.6 Plotting a graph with GNUPlot (3)

receives a whole lot of arguments related to the format of the plot. In addition, we can make GNUPlot export the graph in `.eps` or `.png` format. On our website we provide detailed information about all the optional arguments of `plot` and `execplot`, what kinds of values need to be supplied and what they actually do. We also show how to use GNUPlot to create histograms. The graph resulting from the last statement is shown in Figure 1.6.

1.6.2 THREE-DIMENSIONAL PLOTTING

The conventions for plotting three-dimensional data are very similar to those of two-dimensional plotting. However, in the 3D case we allow only one curve or surface per graph. Hence, we do not have to separately call a plotting routine and the subroutine that execute GNUPlot. Instead, all of this is packed together in the subroutine `plot3d`. Program 1.18 gives an example of how three-dimensional plotting works in practice with our toolbox.

There are two ways to represent three-dimensional data, depending on how the plot should look. In the first part of the program, we want to plot the function

$$z = f(x, y) = \sin(x) \cdot \cos(y)$$

on the interval $(x, y) \in [-5, 5] \times [-3, 3]$. In order to create a plot of this function, we generate some plotting data like in the previous Program 1.17. For the plotting to work correctly, we have to represent the function on a rectilinear grid. This means that we generate x-axis and y-axis data separately and evaluate the function $f(x, y)$ at each

Program 1.18 Plotting graphs with the toolbox

```
program Plotgraphs3D
    [......]
    ! Initialize x values
    do i1 = 0, nplot1
        x(i1) = -5d0 + 10d0/dble(nplot1)*dble(i1)
    enddo

    ! initialize y values
    do i2 = 0, nplot2
        y(i2) = -3d0 + 6d0/dble(nplot2)*dble(i2)
    enddo

    ! get z values
    do i1 = 0, nplot1
        do i2 = 0, nplot2
            z(i1, i2) = sin(x(i1))*cos(y(i2))
        enddo
    enddo

    ! call 3D plotting routine
    call plot3d(x, y, z)

    ! call 3D plotting routine with complete configuration
    call plot3d(x, y, z, color='black', linewidth=0.5d0, marker=2, &
                markersize=0.3d0, noline=.false., &
                xlim=(/-6d0, 6d0/), xticks=1.0d0, xlabel='x-Axis', &
                ylim=(/-3d0, 3d0/), yticks=0.5d0, ylabel='y-Axis', &
                zlim=(/-1d0, 1d0/), zticks=0.2d0, zlabel='z-Axis', &
                zlevel=0d0, surf=.true., surf_color=2, &
                transparent=.true., view=(/70d0, 50d0/), &
                title='sin(x)*cos(y)', filename='testplot', &
                filetype='eps', output='testdata')


    ! initialize spiral data
    do i1 = 0, nplot1
        c(i1) = 20d0*dble(i1)/dble(nplot1)
        a(i1) = sin(c(i1))
        b(i1) = cos(c(i1))
    enddo

    ! plot spiral
    call plot3d(a, b, c, linewidth=2d0)

end program
```

combination of data points x and y. We chose to partition the intervals $[-5, 5]$ and $[-3, 3]$ into 81 and 51 equidistant points, respectively. The x and y data are stored in arrays `x(0:nplot1)` and `y(0:nplot2)`.[15] From evaluating the function at each data-point combination, we get a two-dimensional array `z(0:nplot1, 0:nplot2)` that contains the function values of f. We can now simply plot our function by feeding our data into the subroutine `plot3d`. This subroutine again calls GNUPlot and produces a surface plot of

[15] Note that we are not required to chose an equidistant partition. Instead, any arbitrary partition of the intervals would be allowed.

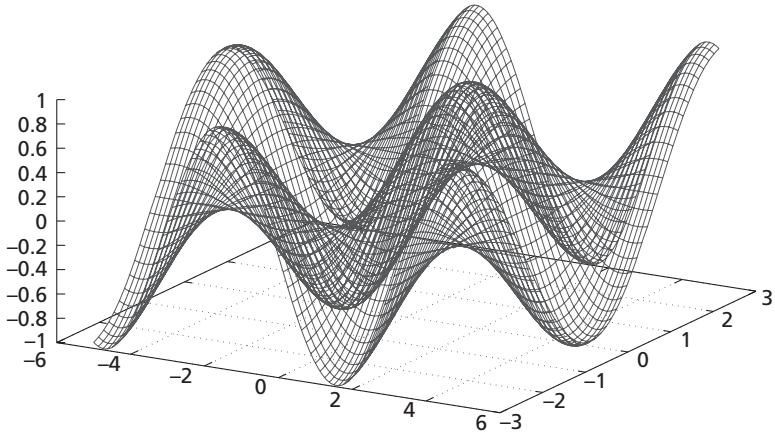


Figure 1.7 Plotting a surface with GNUPlot

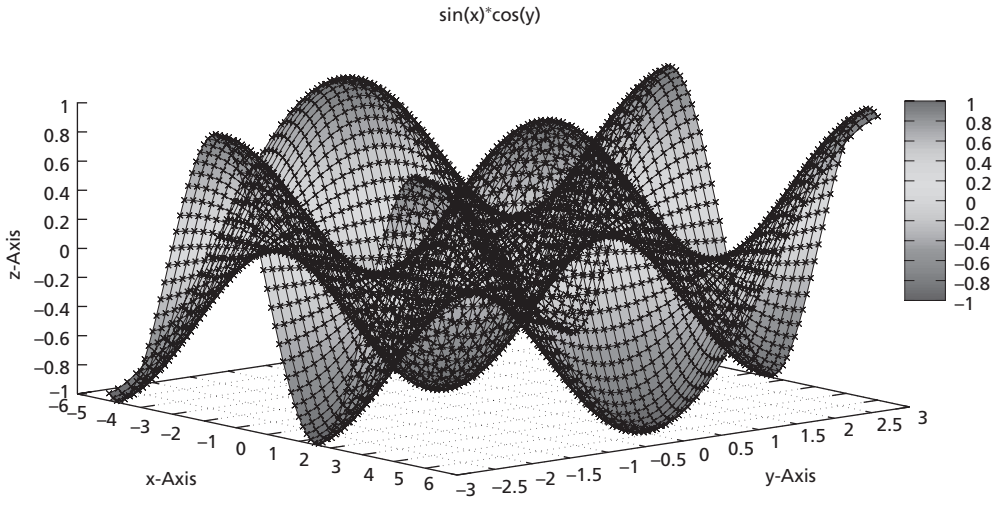


Figure 1.8 Plotting a surface with GNUPlot (2)

the function as can be seen in Figure 1.7. The next statement calls the subroutine `plot3d` again, but this time we provide all possible input arguments by which we can alter the appearance of the plot, see again the website for details on all these input arguments. The corresponding output is shown in Figure 1.8.

Finally, GNUPlot can not only plot the surfaces of functions, but is also able to represent simple lines in the three-dimensional space. One example of such a line is a spiral. A spiral can be represented by the data points $(a, b, c) = (\sin(c), \cos(c), c)$. We generate these data points in the last part of Program 1.18. The corresponding plot data are three arrays, `a`, `b`, and `c`, of the same length. When we feed these arrays into the subroutine `plot3d`, it generates a three-dimensional spiral as can be seen from Figure 1.9.

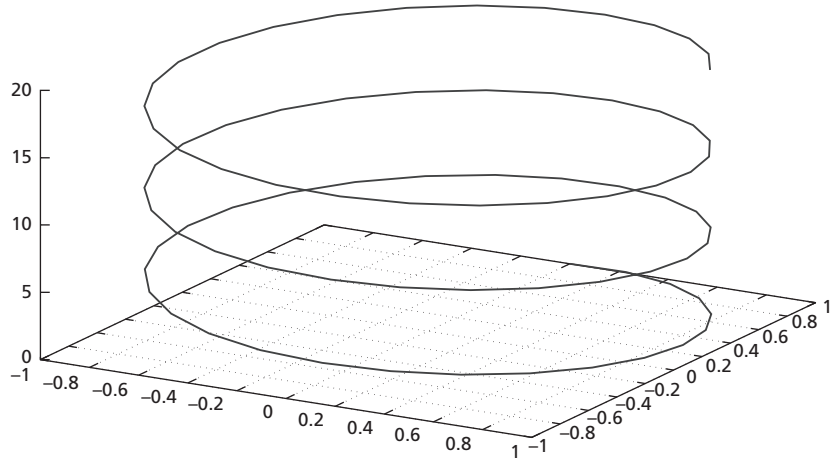


Figure 1.9 Plotting a line in 3D with GNUPlot

Note that this feature of three-dimensional plotting could also be used to generate a three-dimensional scatter plot of some data. To achieve this, we should set the `noline` option to `.true.` to avoid that GNUPlot connects data points with a line and specify a `marker` that generates a mark for each of the data-point combinations.

1.7 **Further reading**

The best way to learn a programming language is to read, write, and run programs immediately. Kendrick and Amman (1999) provide some guidance through the most widely used programming languages of economists. Aruoba and Fernandez-Villaverde (2015) compare some modern programming languages (including Fortran 2008) with respect to their speed and coding requirements. As it turns out, Fortran has a considerable speed advantage and the coding is fairly simple and compact. There are many good books on Fortran programming. Chivers and Sleightholme (2012) provide a very gentle introduction for complete beginners with little or no programming background, but also offer a lot of advanced material for experienced Fortran programmers who want to update their skills. Each chapter contains many example programs and a list of problems to solve. While Chapman (2004) covers almost the same material, the main advantage is the very student friendly design. Throughout the book various boxes highlight good programming practices and programming pitfalls and specific sidebars provide additional information of potential interest to the student. Students are especially motivated by various quizzes that are answered in the appendix and many exercises at the end of each chapter.

1.8 Exercises

1.1. (a) Write a program that reads two scalars, say `x` and `y` of type **`real*8`** from the console. Print an error message when the numbers are not readable. Otherwise the program should compute `x+y`, `x-y`, `x*y` and `x/y` and print the answers in a readable fashion. Test your program with the numbers `x=2` and `y=4`.

(b) Change your program so that the two scalars `x` and `y` are of type **`integer`**. Test the program again with `x=2` and `y=4` and explain the difference.

1.2. Write a program that adds the numbers 55,555,553 and 10,000,001 and stores the result in variables `sum1` and `sum2` which are of type **`real`** and **`real*8`**, respectively. Print the values of `sum1` and `sum2` to the console, compare the two results and explain the difference.

1.3. (a) Store the value of 10^9 in a variable of type **`real*8`** in three ways: `10**9`, `10d0**9`, and `10**9d0`. Then repeat the same for the value 10^{10}. Print out the variable values to the console and check the result.

(b) Extend the program and define two **`real*8`** variables of value 0.000000000003. The first definition ends with `d0` while in the second definition the `d0` is omitted. Print the values of the two variables with format `f30.25` and explain the difference.

(c) Finally define two **`real*8`** variables of value 3.1415926535. Again, the first definition ends with `d0` while in the second definition the `d0` is omitted. Print the values of the two variables with format `f15.12` and explain the difference.

1.4. Write a program that evaluates the logical expression

```
x >= 3.and.y <= 4.and.z == 5.or.x <= y.and.y < z
```

for `x = 4`, `y = 6`, and `z = 8`. Explain the result. Evaluate again for `x = 4`, `y = 6`, and `z = 2` and explain.

1.5. The German income tax function has four brackets:

If taxable income y is		
more than	but not more than	then income tax $T(y)$ is (in €)
0 €	8130 €	0
8131 €	13469 €	$(933.70x + 1400)x$
13470 €	52881 €	$(228.74z + 2397)z + 1014$
52882 €	250730 €	$0.42y - 8196$
250731 €		$0.45y - 15718$

$x = (y - 8130)/10000$, $z = (y - 13469)/10000$.

Write a program that reads some taxable income y of type `real*8` from the console and then computes the resulting tax burden $T(y)$, the average tax rate $\frac{T(y)}{y}$, and the marginal tax rate $\frac{\partial T(y)}{\partial y}$ and prints it to the console.

1.6. The Fibonacci-series is defined as follows: The first two elements of the series are $a_1 = 1$ and $a_2 = 1$. Each of the following elements is computed as the sum of the two previous elements, i.e. $a_n = a_{n-1} + a_{n-2}$. The first ten elements of the Fibonacci-series are therefore

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

(a) Write a program that computes the n-th element of the Fibonacci-series. The main program should first read the value of an **integer** scalar n which determines the requested element a_n of the series. Then the main program should print out the result to the console using a function `fib(n)`. This function calculates a_n using a **do**-loop.

(b) In the second step add to this program a function `binform(n)` which uses Binet's formula

$$F_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}} \qquad \text{with} \quad \phi = \frac{1+\sqrt{5}}{2}$$

to compute the n-th Fibonacci number, where ϕ is the *golden ratio*.

(c) Define a maximum number `plotmax` and compute the relative difference between the two approaches for each element up to `plotmax`. Plot this difference to the console.

1.7. Write a program that simulates rolling a pair of six-sided dice. The outcome we are interested in is the sum of the values the two dice show. We know that with probability 1/36 this sum equals 2, with probability 2/36 it equals 3, and so on. The goal of this task is to simulate this probability distribution using random number generation. Therefore define a number `iter` of times for which you simulate rolling the dice. In each iteration, use the intrinsic random number generator subroutine **random_number**`(x)`, which sets the argument x to a pseudo random real number x with $0 \leq x \leq 1$. Use this number to simulate the outcome of one roll of one dice. Do this twice and sum up the result to a variable `d`. To store the results, create an array `Dsum(2:12)` and add a value of 1 to the entry `d` of this array. The simulated probability distribution is then **dble**`(Dsum)/`**dble**`(iter)`. Print this probability distribution to the console.

Then generalize your program to rolling n dice, each with k sides. Again, plot the results for the sums between n and nk to the console.

Hint: Use the intrinsic subroutine `random_seed()` at the beginning of the program. What is the effect of this subroutine?

1.8. Two six-sided dices are rolled until their sum equals a value $x > 2$ or until their sum is equal to a value $y > 2$. Write a simulation program to determine the percentage of how often the game will be stopped by the first condition instead of the second condition. To generalize the simulation procedure of rolling the dice, create a subroutine `random_int(res, int1, inth)`. This subroutine should generate a random number of type **`integer`** between the values `int1` and `inth`. Thereby assume that each of the potential integer values {`int1`, `int1+1`, …, `inth` } occurs with equal probability. The simulated random number should be stored in the value `res`.

Plot the percentages and the largest number of rolls before the game stops to the console. Set $x = 4$ and $y = 10$ and test the program. Then set $x = 4$ and $y = 7$. Explain your results!

Now use three dice to simulate the game. First set $x = 4$ and $y = 17$ and then set $x = 4$ and $y = 5$. Again explain your results.

1.9. (a) Write a function `utility(c, gamma)` that computes for an input variable `c` and an intertemporal substitution elasticity `gamma` the value of the utility function

$$u(c) = \frac{c^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}}.$$

Function `utility(c, gamma)` should first check whether `c` is positive. In case `c<0` an error message should be written to the console and the program should stop (use the **`stop`** command).

(b) Now write a program that tests the function `utility(c, gamma)` with alternative values for `c` and `gamma`. Then plot the function using the toolbox for alternative parameter values $\gamma \in [0.25, 0.5, 0.75, 1.25]$.

1.10. Write a subroutine `utility_int(a, b, u)` where the input values `a` and `b` define the endpoints of an interval `[a, b]`. Within this interval the subroutine should compute the values of the function $u(c)$ from the previous exercise at `n` nodes. The function values should be stored in the array `u(:)` which is of assumed size and will be given back to the main program. The value of `n` can be calculated from size of `u(:)`. Therefore proceed via the following steps:

(a) At the beginning of the subroutine check whether `0 < a < b`. If this is not the case, write an error message and stop the program.

(b) If `0<a<b` holds then determine `n` by setting `n = size(u)`.

(c) Compute the array entries in `u(:)` as $u(c_j)$ using the function `utility(c)` with a specific value for γ. Define

$$c_j = a + \frac{(j-1)}{n-1}(b-a) \quad \text{for} \quad j = 1, \ldots, n.$$

Don't forget the **`dble`** command.

Now set $\gamma = 0.5$ and write a program that tests the subroutine using `a = 1` and `b = 2`, and an array length of 11. Write your results to the console.

2 Numerical solution methods

In this chapter we develop simple methods for solving numerical problems. We start with linear equation systems, continue with nonlinear equations and finally talk about optimization, interpolation, and integration methods. Each section starts with a motivating example from economics before we discuss some of the theory and intuition behind the numerical solution method. Finally, we present some Fortran code that applies the solution technique to the economic problem.

2.1 Matrices, vectors, and linear equation systems

This section mainly addresses the issue of solving linear equation systems. As a linear equation system is usually defined by a matrix equation, we first have to talk about how to work with matrices and vectors in Fortran. After that, we will present some linear equation system solving techniques.

2.1.1 MATRICES AND VECTORS IN FORTRAN

The general structure of a matrix A and a vector b in mathematics is given by

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

A matrix consists of several columns, where a vector only has one. We call A a $m \times n$ matrix, and b a n-dimensional vector. A natural way to store matrices and vectors in Fortran is via the concept of arrays. We thereby store matrices in a two-dimensional array of lengths m and n and a vector in a one-dimensional array of length n. There are a number of intrinsic Fortran functions that were especially written for operation with matrices and vectors. A summary of these is given in Program 2.1. Most of these functions should be self-explanatory, however they are also described on our website.

Program 2.1 Matrix and vector operations

```
program matrices

    implicit none
    integer :: i, j
    real*8 :: a(4), b(4)
    real*8 :: x(2, 4), y(4, 2), z(2, 2)

    ! initialize vectors and matrices
    a = (/(dble(5-i), i=1, 4)/)
    b = a+4d0
    x(1, :) = (/1d0, 2d0, 3d0, 4d0/)
    x(2, :) = (/5d0, 6d0, 7d0, 8d0/)
    y = transpose(x)
    z = matmul(x, y)

    ! show results of different functions
    write(*,'(a,4f7.1/)')'          vector a = ',(a(i),i=1,4)
    write(*,'(a,f7.1)')  '            sum(a) = ',sum(a)
    write(*,'(a,f7.1/)') '        product(a) = ',product(a)
    write(*,'(a,f7.1)')  '         maxval(a) = ',maxval(a)
    write(*,'(a,i7)')    '         maxloc(a) = ',maxloc(a)
    write(*,'(a,f7.1)')  '         minval(a) = ',minval(a)
    write(*,'(a,i7/)')   '         minloc(a) = ',minloc(a)
    write(*,'(a,4f7.1)') '     cshift(a, -1) = ',cshift(a,-1)
    write(*,'(a,4f7.1/)')'    eoshift(a, -1) = ',eoshift(a,-1)
    write(*,'(a,l7)')    '        all(a<3d0) = ',all(a<3d0)
    write(*,'(a,l7)')    '        any(a<3d0) = ',any(a<3d0)
    write(*,'(a,i7/)')   '      count(a<3d0) = ',count(a<3d0)
    write(*,'(a,4f7.1/)')'          vector b = ',(b(i),i=1,4)
    write(*,'(a,f7.1/)') ' dot_product(a,b) = ',dot_product(a,b)
    write(*,'(a,4f7.1/,20x,4f7.1/)') &
                        '          matrix x = ',((x(i,j),j=1,4),i=1,2)
    write(*,'(a,2f7.1,3(/20x,2f7.1))')&
                        '     transpose(x) = ',((y(i,j),j=1,2),i=1,4)
    write(*,'(a,2f7.1/,20x,2f7.1/)') &
                        '       matmul(x,y) = ',((z(i,j),j=1,2),i=1,2)

end program
```

2.1.2 SOLVING LINEAR EQUATION SYSTEMS

In this section we would like to show how to solve linear equation systems. There are two ways to solve these systems: by factorization or iterative methods. Both methods will be discussed in the following.

Example Consider the supply and demand functions for three goods given by

$$\begin{aligned} q_1^s &= -10 + p_1 & \qquad q_1^d &= 20 - p_1 - p_3 \\ q_2^s &= 2p_2 & \qquad q_2^d &= 40 - 2p_2 - p_3 \\ q_3^s &= -5 + p_3 & \qquad q_3^d &= 25 - p_1 - p_2 - p_3 \end{aligned}$$

As one can see, the supply of the three goods only depends on their own price, while the demand side shows strong price interdependencies. In order to solve for the equilibrium prices of the system we set supply equal to demand $q_i^s = q_i^d$ in each market which after rearranging yields the linear equation system

$$\begin{aligned} 2p_1 + p_3 &= 30 \\ 4p_2 + p_3 &= 40 \\ p_1 + p_2 + 2p_3 &= 30. \end{aligned}$$

We can express a linear equation system in matrix notation as

$$Ax = b \tag{2.1}$$

where x defines an n-dimensional (unknown) vector and A and b define a $n \times n$ matrix and a n-dimensional vector of exogenous parameters of the system. In the above example, we obviously have $n = 3$ and

$$A = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 4 & 1 \\ 1 & 1 & 2 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} 30 \\ 40 \\ 30 \end{bmatrix}.$$

Gaussian elimination and factorization We now want to solve for the solution x of a linear equation system. Of course, if A were a lower triangular matrix of the form

$$A = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix},$$

the elements of x could be easily derived by simple forward substitution, i.e.

$$\begin{aligned} x_1 &= b_1/a_{11} \\ x_2 &= (b_2 - a_{21}x_1)/a_{22} \\ &\vdots \\ x_n &= (b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{nn-1}x_{nn-1})/a_{nn}. \end{aligned}$$

Similarly, the problem can be solved by backward substitution, if A was an upper triangular matrix. However, in most cases A is not triangular. Nevertheless, if a solution

to the equation system existed, we could break up A into the product of a lower and upper triangular matrix, meaning there is a lower triangular matrix L and an upper triangular matrix U, so that A can be written as

$$A = LU.$$

If we knew these two matrices, equation (2.1) could be rearranged as follows:

$$Ax = (LU)x = L(Ux) = Ly = b.$$

Consequently, we first would determine the vector y from the lower triangular system $Ly = b$ by forward substitution and then x via $Ux = y$ using backward substitution.

Matrix decomposition In order to factorize a matrix A into the components L and U we apply the *Gaussian elimination method*. In our example we can rewrite the problem as

$$\begin{aligned} Ax &= \begin{bmatrix} 2 & 0 & 1 \\ 0 & 4 & 1 \\ 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{L} \times \underbrace{\begin{bmatrix} 2 & 0 & 1 \\ 0 & 4 & 1 \\ 1 & 1 & 2 \end{bmatrix}}_{U} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \underbrace{\begin{bmatrix} 30 \\ 40 \\ 30 \end{bmatrix}}_{b}. \end{aligned}$$

We now want to transform L and U in order to make them lower and upper triangular matrices. However, these transformations must not change the result x of the equation system $LUx = b$. It can be shown that subtracting the multiple of one row from another satisfies this condition. Note that when we *subtract* a multiple of row *i* from row *j* in matrix U, we have to *add* the same multiple to the same cell in matrix L which is eliminated in matrix U.

In the above example, the first step is to eliminate the cells below the diagonal in the first column of U. In order to get a zero in the first column of the last row, one has to multiply the first row of U by 0.5 and subtract it from the third. Consequently, we have to add 0.5 to the first column of the third row of L. After this step matrices L and U are transformed to

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 4 & 1 \\ 0 & 1 & 1.5 \end{bmatrix}.$$

In order to get a zero in the second column of the last row of U, we have to multiply the second row by 0.25 and subtract it from the third line. The entry in the last line and the second column of L then turns into 0.25. After this second step the matrices are

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.5 & 0.25 & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 4 & 1 \\ 0 & 0 & 1.25 \end{bmatrix}.$$

Now L and U have the intended triangular shape. It is easy to check that $A = LU$ still holds. Hence, the result of the equation systems $LUx = b$ and $Ax = b$ are identical. The above approach can be applied to any invertible matrix A in order to decompose it into the L and U factors. We can now solve the system $Ly = b$ for y by using forward substitution. The solution to this system is given by

$$\begin{aligned} y_1 &= 30/1 = 30, \\ y_2 &= (40 - 0 \cdot 30)/1 = 40 \quad \text{and} \\ y_3 &= (30 - 0.5 \cdot 30 - 0.25 \cdot 40)/1 = 5. \end{aligned}$$

Given the solution $y = [30\,40\,5]^T$, the linear system $Ux = y$ can then be solved using backward substitution, yielding the solution of the original linear equation, i.e.

$$\begin{aligned} x_3 &= 5/1.25 = 4, \\ x_2 &= (40 - 1 \cdot 4)/4 = 9 \quad \text{and} \\ x_1 &= (30 - 0 \cdot 9 - 1 \cdot 4)/2 = 13. \end{aligned}$$

The solution of a linear equation system via LU-decomposition is implemented in the toolbox that accompanies this book. Program 2.2 demonstrates its use. In this program, we first include the `toolbox` module and specify the matrices A, L, U and the vector b. We then initialize A and b with the respective values given in the above example. The subroutine `lu_solve` that comes with the toolbox can now solve the equation system $Ax = b$. The solution is stored in the vector `b` at the end of the subroutine. Alternatively we could solely factorize A. This can be done with the subroutine `lu_dec`. This routine receives a matrix A and stores the L and U factors in the respective variables. The output shows the same solution and factors as computed above.

The so-called L-U factorization algorithm is faster than other linear solution methods such as computing the inverse of A with determinants and then computing $A^{-1}b$ or using Cramer's rule. Although L-U factorization is one of the best general methods for solving a linear equation system, situations may arise in which alternative methods may be preferable. For example, when one has to solve a series of linear equation systems which all have the same A matrix but different b vectors, $b_1, b_2, \ldots, b_m$ it is often

Program 2.2 Linear equation-solving using the toolbox

```
program lineqsys

    use toolbox

    implicit none
    integer :: i, j
    real*8 :: A(3, 3), b(3)
    real*8 :: L(3, 3), U(3, 3)

    ! set up matrix and vector
    A(1, :) = (/ 2d0,  0d0,  1d0/)
    A(2, :) = (/ 0d0,  4d0,  1d0/)
    A(3, :) = (/ 1d0,  1d0,  2d0/)
    b       = (/30d0, 40d0, 30d0/)

    ! solve the system
    call lu_solve(A, b)

    ! decompose matrix
    call lu_dec(A, L, U)

    ! output
    write(*,'(a,3f7.2/)')'    x = ', (b(j),j=1,3)
    write(*,'(a,3f7.2/,2(5x,3f7.2/))') &
        ' L = ',((L(i,j),j=1,3),i=1,3)
    write(*,'(a,3f7.2/,2(5x,3f7.2/))') &
        ' U = ',((U(i,j),j=1,3),i=1,3)

end program
```

computationally more efficient to compute and store the inverse of A and then compute the solutions $x = A^{-1}b_j$ by performing direct matrix vector multiplications.

Gaussian elimination can be accelerated for matrices possessing special structures. If A was symmetric positive definite, A could be expressed as the product

$$A = LL^T$$

of a lower triangular matrix L and its transpose. In this situation one can apply a special form of Gaussian elimination, the so-called *Cholesky factorization algorithm*, which requires about half of the operations of the Gaussian approach. L is called the Cholesky factor or square root of A. Given the Cholesky factor of A, the linear equation

$$Ax = LL^T x = L(L^T x) = b$$

may be solved efficiently by using forward substitution to solve $Ly = b$ and then backward substitution to solve $L^T x = y$.

Another factorization method decomposes $A = QR$, where Q is an *orthogonal* matrix and R an upper triangular matrix. An orthogonal matrix has the property $Q^{-1} = Q^T$. Hence, the solution of the equation system can easily be computed by solving the system