
Advanced Programming for Numerical Calculations:

Python & Fortran

*Juan Antonio Hernández Ramos
Miguel Ángel Rapado Tamarit*

*Department of Applied Mathematics
School of Aeronautical and Space Engineering
Technical University of Madrid (UPM)*

Cover:

If you want something, go for it. Cover design Belén Moreno Santamaría

Miguel Ángel Rapado Tamarit
marapadotamarit@gmail.com

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission in writing from the authors.

© 2022 Juan Antonio Hernández Ramos
Miguel Ángel Rapado Tamarit

ISBN 978-1727581539

Contents

I	Foundations	3
1	Overview	5
2	Programming languages and computers	7
2.1	Computers through documentaries and movies	7
2.2	Hidden figures	7
2.3	The imitation game	7
2.4	The triumph of nerds	8
2.5	Micro men: Clive Sinclair	10
2.6	The revolution OS. The code	13
2.7	The thirteenth floor	16
2.8	The colors of infinity	16
2.9	The great hack	16
2.10	The social dilemma	17
2.11	Lo and behold	19
2.12	history of informatics	20
3	Operations and von Neumann style	21
3.1	Introduction	21
3.2	Roots of a second degree equation	22
4	Imperative versus declarative programming	23
4.1	Introduction	23
4.2	Sum of numerical series	24
4.3	First example with imperative paradigm	25
4.4	First example with declarative paradigm	26
4.5	Second example with imperative paradigm	28
4.6	Second example with declarative paradigm	29
4.7	Numerical issues	30
4.7.1	Theoretical framework	30
4.7.2	Mathematical support for Tail function	33

4.7.3	First test for bad convergence sum	34
4.7.4	Second test for bad convergence sum	35
4.7.5	How can I improve this calculation?	36
4.8	Convergence rate	37
5	Operations with vectors and matrices	39
5.1	Introduction	39
5.2	Static size vectors and matrices	41
5.3	Dynamic allocation of vectors and matrices	45
5.4	Memories: Static, Heap, Stack	48
6	Operations with functions	53
6.1	Introduction	53
6.2	Defining piecewise functions	54
6.3	Plotting functions	55
6.4	Integrals and derivatives of functions	57
6.5	Examples of operations with functions	62
7	Series expansion	65
7.1	Introduction	65
7.2	Expansions of functions	66
7.3	Parseval identity	67
8	Read an external file	69
II	Computer operations with integers and reals	73
1	Overview	75
2	Integers representation	77
Overview	77
2.1	Integer overflow example	79
2.2	Two's complement integer representation	81
2.3	Understanding integer overflows	84
2.4	Overflow of constants	87
2.5	Overflow by incorrect assignment	88
2.6	Declaring kind	89
2.7	Two's complement converter	90
2.7.1	TwosCompl_converter_to_decimal function	90
2.7.2	TwosCompl_converter_to_binary function	91
2.7.3	Check_overflowed_int function	92
3	Reals representation and operations	93
3.1	Example of round-off errors	94
3.2	Fixed-point and Floating-point representation	96
3.3	Representation in IEEE 754	99

3.4	Distance between floating point real numbers	111
3.5	Reconstruction from its internal binary representation	115
3.6	Writing floating point expressions	116
3.7	Condition number and stability	121
3.8	Common numerical errors	123
3.9	Catastrophic cancellation	125
3.10	Truncation errors and round-off errors	126
3.11	IEEE exception examples	127
3.12	TO MERGE	127
III	Advanced programming techniques:	129
1	Overview	131
2	Scope	133
2.1	Introduction	133
2.2	Fortran	134
2.3	Python	135
3	First class functions and lexical scoping	137
3.1	Introduction	137
3.2	Fortran	138
3.3	Python	140
4	Overloading: operators and functions	141
4.1	Fortran	142
4.2	Python	143
5	Vector operations	145
5.1	Fortran	145
5.2	Python	146
6	Polymorphism	147
6.1	Example with objects	147
6.1.1	Fortran	148
6.1.2	Python	149
6.2	Example with ODEs integration	149
6.2.1	Python	150
7	Map, filter and reduce	151
7.1	Fortran	151
7.2	Python	152
8	Pointers and targets	153
8.1	Fortran	154
8.2	Python	156

9	Cauchy problem	159
9.1	Cauchy problem solver	159
9.2	Temporal schemes	159
9.3	Stability region	159
9.4	N body problem	159
10	Fractals	161
10.1	Fortran	162
10.1.1	VonKoch	162
10.1.2	Mandelbrot	163
10.2	Python	163
10.2.1	VonKoch	163
10.2.2	Mandelbrot	163
11	Games	165
11.1	Fortran	166
11.1.1	Sudoku	166
11.2	Python	167
11.2.1	Sudoku	167
IV	Software development	169
1	Software development	171
1.1	Software development life cycle	171
1.2	Methodology	173
1.2.1	Names of function, classes and variables	173
1.2.2	Code developing methodology	173
1.3	Imperative or von Newmann style	174
1.4	Functional paradigm	174
	References	175

Introduction

```
write(*,*) "Welcome to Advanced Programming"
write(*,*) "for Numerical Calculations"

write(*,*) " select an option "
write(*,*) " 0. exit/quit  "
write(*,*) " 1. Foundations"
write(*,*) " 2. Integer overflow and floating point arithmetic  "
write(*,*) " 3. Advanced programming techniques  "
```

Listing 1: `main.f90`

Part I

Foundations

Chapter 1

Overview

As a first approach to the use of Fortran to learn applied mathematics a chapter including basic operations will be presented. In it the reader can become familiar to the use of basic sentences in order to perform simple mathematical operations.

```
write(*,*) " select an option "  
write(*,*) " 0. exit/quit  "  
write(*,*) " 1. Hello world"  
write(*,*) " 2. Roots of second grade equation"  
write(*,*) " 3. Sum of series  "  
write(*,*) " 4. Vectors and matrices  "  
write(*,*) " 5. Memory allocation  "  
write(*,*) " 6. Integrals and derivatives  "  
write(*,*) " 7. Taylor expansion  "  
write(*,*) " 8. Read/write data from external files  "
```

Listing 1.1: Foundations.f90

Chapter 2

Programming languages and computers

2.1 Computers through documentaries and movies

2.2 Hidden figures

2.3 The imitation game

3 manchester, he was arrested for homosexuality. Homosexuality was punished at that time in england.

In 1936, Turing had invented a hypothetical computing device that came to be known as the ‘universal Turing machine’.

Whilst working for the National Physical Laboratory (NPL), Turing published a design for the ACE (Automatic Computing Engine), which was arguably the forerunner to the modern computer. The ACE project was not taken forward, however, and he later left the NPL.

World War II, also known as the Second World War, was a global war that lasted from 1939 to 1945.

1942 he developed the bombe machine to decipher.

The first working device to be built was a point-contact transistor invented in 1947 by American physicists John Bardeen and Walter Brattain while working under William Shockley at Bell Labs. The three shared the 1956 Nobel Prize in Physics for their achievement.

ENIAC Introduced to the world on Feb. 14, 1946, the ENIAC – Electronic Numerical Integrator and Computer – was developed by the University of Pennsylvania’s John Mauchly and J. Presper Eckert

was developed under a 1943 contract with the U.S. Army to speed ballistics calculations. It was built under total secrecy and completed only after WWII, however. At 30 feet by 60 feet, weighing 30 tons and using 19,000 vacuum tubes, ENIAC was the epitome of a large system.

The Fleming valve, also called the Fleming oscillation valve, was a thermionic valve or vacuum tube invented in 1904 by English physicist John Ambrose Fleming as a detector for early radio receivers used in electromagnetic wireless telegraphy

2.4 The triumph of nerds

ROBERT CRINGKELY

2.1) Triumph of nerds. Paul allen, bill gates, wozniak, Steve jobs. Mainframe versus personal computer.

Cobol, fortran, basic. First languages. Nerds thought it was a bible for them. For a 10 years old. It was incredible thrilling experience.

Vacuum tubes or valves. Microprocessor That’s the secret of computer. That’s why it’s called silicon valley. Intel invented the Microprocessor. They didn’t realize the potential of the personal computer

1975 Ed Roberts. Invented the first personal computer ALTAIR 8080 Microprocessor. He invented it to play with. He run a calculator in Albuquerque. No monitor, no keyboard. Homebrew computer club. Stanford.

To understand what to do with this useless computer.

Paul Allan began with the Altair and with a tape of paper programming basic. They were 19. They attached keyboards and monitor. At the end of 1975. A revolution. Groovy

The first apple computer was to impress the club. Two years past from 1975 to build apple 2. Wozniak was the Mozart of the digital computer. Success with 21 years old.

Visual calc. The first spread sheets. Profits and expenses. Running numbers by hand. Dan Bricklin. The programmer. They don't make money. They wanted to make a better world.

Obses with numbers. It was crystal ball.

Part II In 1980 apple ii was a success. An IBM appeared into the scene. At the beginning IBM was making mainframes. Tom Watson. Strict dress code. Work 9 to 5 and Saturday wash the car. Entire culture. The songs of IBM. Gay meant something different. Are you an ibmer. Conservative. IBM noticed the explosion of apple II.

A pc needs software. Computer language. Operating system. Bill gates has the basic language. CPM Gary kildall. First OS. He was not a fighter. Bill gates was very competitive. IBM was looking for OS and language. Gary didn't receive IBM and bill gates saw the opportunity. They replicated CPM in four months: they call it pc dos 1.0 . They bought it the OS from Tim Patterson. 50 thousand dollars was the price. Then, Bill and Paul become multi billionaire.

IBM entered in the market. 1981. Lotus 1/2/3. It was the Killer application. Euphoric.

People wanted to copy the IBM. Reverse engineering.

Engineering from Texas computer founded Compaq computer. Micro was available from Intel. Rom bios was patented. You seen the product. You need vigin. Never seen it. After the virgins test. 15 senior engineers. It took one year to accomplish that pc and one million dollars. Terror was in IBM because prices went down because competence.

IBM with OS/2. Tried to steal the business to microsoft. A culture clash between IBM and microsoft. 1990 microsoft with windows broke relations with IBM but the idea came from apple.

IBM gave 1/3 to Intell and 1/3 to Microsoft. Big error.

Part III. A revolution to mage pc friendier s

1995 windows announced its OS. Those ideas was invented 20 years ago. GUI: graphical user interface.

It all began in 1971. 1971 Xerox in Palo Alto. They created the first GUI. Freedom is that company. To work on five years program. Their dreams. The mouse was created there. A terrible mismatch between researchers and managers.

Steve Jobs wanted to change the world not to earn money. 1979 visit to Xerox. Steve Jobs had the vision . Xerox showed steve three things: 1) a network like internet, 2) a GUI and 3) object oriented programming. He was blinded by 2). It was a turning point.

After an hour Steve understood what was going on. Apple developed Lisa with one hundred engineers.

In 1981 apple was in trouble because IBM was selling software that didn't run on MAC. IBM had software. In 1983 they announced a game to create software for mac. Bill Gates was there. He didn't realize that he was going to be his rival Instead of IBM. Macintosh the first affordable pc with a GUI. Very bad sales. It was 1000 dollars more expensive than IBM.

Commercial against IBM tyranny.

Jobs made agreement with bill gates to create software. He thought that the big blue was the enemy.

What you see is what you get. Wysiwyg. Apple needed a killer application. The problem was the dot printer. Agreement with adobe people to create laser images printings. Adobe also developed in Xerox Apple bought those software. they developed a killing application. Steve terrified everyone. So one steve left apple was twofold.

Scotty talked with apple board and decided to follow Scotty plans. Steve said that he hired the wrong guy: Scotty Pepsi cola. Steve left the company.

1984. Microsoft launched windows to challenge macintosh.

Apple sued microsoft. A long a legal battle (6 years). Apple lost. Steve said microsoft have no taste. In the sense that they don't think original ideas.

Society needs 30 years to arrange Advances in technologies.

2.5 Micro men: Clive Sinclair

He asked for national funds

We exist to push barriers.

As inventors we are obliged to dream.

He created sinclair company close to Cambridge university. Clive wanted to develop his electric car and his television but Christopher Curry wanted to create a home computer.

Margaret thatcher (Zacher) was elected.

Curry Christopher began a company with Hermann Hauser an Austrian guy who obtained his ph d in cavendish lab.. Cambridge processor group. They join those guys who did everything for fun.

Radionics 1978.

Clive saw the announcement of 2000 pounds of apple ii. And thought. What is it so expensive ? 99 pounds. Sinclair zx80.

His vision a computing device in every home in Britain. Price is the key.

1980. Sinclair launched zx80 at 99 pounds.

Acorn was founded by Christopher with memory and more power.

They began to fight. They met in a bar to discuss B BBC computer conquest. Join forces

Nerds eating with the tester wires.

BBC announces a computer home initiative They have a computer series.

BBC was expecting something different. Chris said that he would do something in four days. He asked his team to do it. They did it. Acorn computer prototype.

To put a computer in every school of the country. Acorn launched BBC computer.

125 pounds zx spectrum. It was a revolution. Mainly games.

The Zilog Z80 is a software-compatible extension and enhancement of the Intel 8080

What else can I do ? BBC micro from acorn did not have so many games.

Quantum leap. Another powerful computer is 3 months.

1984 sinclair quantum leap 399 pounds. Ready to ship in 20 days.

Clive was concerned with his electric car.

BBC basic language. It was success.

Sinclair clash with Chris in a pub.

The computer boom became to an end.

Acorn designed to ARM microchip. The most famous microchip . As of 2018, over 100.000 million copies of the ARM processor have been manufactured, powering much of the world's mobile computing and embedded systems. Population of the world In 2018: 7600 million persons.

Acorn was considered the british apple.

Sinclair company was sold to amstrad computers.

Z80 Spectrum 1982

Acorn atom: competence.

Quatum leap

1988 amstrad.

The computer boom went to and end.

The future still needs inventing.

Acorn sold 1.5 million BBC micros 250000 unsold computer electrons.

The company was bought by Olivetti in 1985. The ARM chip designed by Acorn team has gone on to be the most popular michochip in history.

Clive Sinclair sold more than 12000 electric mobile cars C5.

The home computer market is now dominated by giant American companies.

2.6 The revolution OS. The code

Free software: Richard Stallman

Open source: ERIC Raymond, Linus Torvald

Bruce Perens: open source definition Colaborating with software.

Donald Knuth 1979. Latex R Stallman 1983 GNU free software. Eric Steven Raymond : The cathedral and the bazar: musing on Linux and open source by accidental revolutionary.

Linus Torvalds says that R Stallman was the great philosopher and he was the engineer.

Linux versus windows

Free software demands freedom. Open source demands quality.

People uses programs instead of OS.

Richard Stallman 1971 in MIT. Founded They didn't like passwords. The security was a joke. Homebrew computer club. Bill Gates defends private software.

Stallman began to write gnu Unix.

He finished in 1991. Compilers, editors. With free software you have freedom

Free has two meanings in English. One Related to money and other related to freedom.

The first company around gnu software wax to sell services around gnu software. In a proprietary software, support is a monopoly. Microsoft.

Stallman gives in his manifesto the keys to make money.

1) support by everyone. Deliver support. cygnus was the first company.

1991-1993 infancy of Linux.

Stallman tried to divide the kernel. Very difficult to debug but very powerful.

Linus tried a monolithic kernel.

The cathedral of the bazaar. He discuss how is possible to develop software by breaking as ll strict software rules of engineers.

They changed free software by open software . Stallman is free software and others open software. Free to redistribution.

Linux open software.

VA Linux company

IPO

People wanted to have a Unix sun sparc station at home. That's why Linux was so important. It was 2 x times and 1/4 the price of a sun sparc. 2000 dollars a IBM PC with Linux in comparison a sun sparc of 7000 dollars.

Linux had a track similar to internet. In 1993 internet became a common commodity. Apache project. HTTP project. Web server.

Red hat sotware was one the most important companies to support Linux.

Linux Torvald created the free kernel.

Gnu / Linux was Stallman proposal.

Apache web server was the Killer application of Linux. It motivates companies to internet.

Apache software foundations.

Raymond thought that

Sharing is the basic of society.

Open software is not comunishm because it does not force people to share.

The revolution goes prime time.

Red hat Linux company became public.

GPL Genral public license.

The mind behind Linux

Linux code.

1998 Netscape was one the first companies. Influenced by the cathedral and the bazaar. Source code was its product. Mozilla product.

They change to open source 1) free redistribution 2) 3) 4) integrity of author code 5) no discrimination

Red hat goes to market. 1999

The code: the story of Linux. You tube . Boring socialism in action.

Open source project is compared to science or knowledge.

Social machine around open source code

Revolution OS . You tube. Free software gnu, Linux . Better than the code but boring.

1991 first Linux 10000 lines of code.

1992. 40000 lines of code and 1000 users.

1993. 100000 lines of code and 20000 users.

1995. 250000 lines of code and 500000 users.

1997. 800000 lines of code and 3.5 million users.

1998. 1.5 million lines of code and 7.5 million users.

1999. 12 million users.

Gnu : main component c compiler.

Richard stallman. Founder of gnu.

Apache web server. 1993. Mass market. Application built with Linux.

The cathedral and the bazaar. Book. Eric Raymond.

1998 Netscape. Release source code. Mozilla. Browser.

1999.

Influential

VA Linux

Free software. Gnu. Collaboration.

Open source. VA Linux. Definition: free redistribution, source code available, derived works permitted, integrity of the author of code. No discrimination against

Cygnus . Free sotware model. First company.

Oracle.

1998. Venture capilist. Enter in open source.

When Stallman recieves the prize in that congress. Torvald ignores him.

VA Linux company multiplied its share values by 10 In one day.

2.7 The thirteenth floor

2.8 The colors of infinity

2.9 The great hack

We are more the commodity. But we are in love with free service. Data attached to our identify. 2016 elections. It was a change.

Project akamo 1million dollars per day in Facebook ads.

Cambridge analytica. The brain. In UK Alexander Nix was its ceo.

An environment of great innovation. Expert data science. 5000 data points in every american. To model personality. It was tied to the brexit champein. They tried to break the world to rebuild new pieces. Ruining our democracy.

Brittany Kaiser. Top executive.

SCL.

Defeat crooked Hillary : dishonest, illegal. Cambridge analytica created that slogan.

The guardian and the observer.

It's about our democracy.

Roger mcnamee. Investor facebook. I feel really guilty. He was not able to sleep at night.

Brittany says thay obama did not pay her to his champein and she needed money.

Fear and hate. Divide and conquer. Facebook was created to connect people.

Mi

Carole Cadwalladr. TED talk. She was one of the jurnalist. Driving us apart. Is this what you want.

You have to understand how you data is affecting your life. Our dignity is at stake.

Donald Trump 5.9 million versus 66000 dollars Hillary in Facebook ads.

David Carroll continues to teach and advocate for data rights to be recognized as the new human rights.

2.10 The social dilemma

They are very concerned. When you look around you is like the world is going crazy.

In 2006 facebook was looking at Google with admiration. They created something good and the same time a machine to make money.

Book: Jaron Lanier. Ten arguments for deleting your social media accounts. Right now.

Robert macnameee: selling their users.

If you product is free, you are the product.

What do they do with our data ? They build models that predict our actions.
Hi

Facebook techniques: engagement, growth adds

And then give you back the dopamine hit.

Social media is a drug. That directly affects the release of dopamine in the reward pathway.

Around 3 hours on average. Social media.

A pacifier. Social media.

Hundreds of algorithms running simultaneously

Few people working on those algorithms. But machine learning has nothing to do with precise instructions. They don't know the programmers what is going to happen . Because the algorithm is designed to maximize the profit. AI makes the rest. It learns online. And modifies the way of interactions with human beings. In that sense, we don't have the control.

You are playing with AI. It knows everything about you and you don't know anything about him. It's not a fair fight.

Two time scenarios.

1) when AI will be smarter than human

2) right now . The root of addiction. Polarization, radicalization, outrage-ification, vanity-ification. This is checkmate on humanity.

Example. Imagine wikipedia saying different definitions for different people. If wikipedia could know your profile and it was paid to change a little bit your idea, it would change your information to conquer its objectives.

Each person has its own reality with its own story based on his profile. You think what they don't realize what is going on. Your question. Because they are not seeing the same information than others.

Fake news makes companies to earn more money than true news.

Social media amplifies gossips and rumors not allowing us s to distinguish between true and fake.

Conspiracy theories. Very easy.

Technology is the existential threat.

Whether it is to be utopia or oblivion will be a touch-and-go relay race right up to the final moment. Humanity is in his final exam as to whether or not it qualifies for continuance in universe. Buckminster Fuller.

Technology is Simultaneously utopia and distopia.

The business model has a problem.

These markets undermine democracy and undermine freedom.

Based on religion: profits at all costs. Corrosive business model.

Not everyone recognize that we have a problem.

Reform facebook and Google.

Qwant in instead of google. Block notifications.

Three basic rules:

1) out of your bedroom 2) no social media until high school. 3) limit you social media time.

Delete your social media accounts. To have more people to have a conversation.

Check mate on humanity

1) drug addict to social media 2) when superintelligence will surpass human beings. 3) these markets undermine democracy.

Bunkminster Fuller

Three rules to convact social media : 1) avoid phone in your bedroom 2) don't allow kids to enter in social media until they are at least in high school. 3) discuss the time you devote to social media.

2.11 Lo and behold

Reveries of the connected world. Director: Herzog, Werner.

Ucla, 1969 internet was born. First node for decades. Stanford first communication. To log in: connected by a telephone. Lo ang g crashed.

Arpanet. 197–. Protocol and technology.

II. The glory of the net. Bio and automobiles.

III. The dark side. Email with decapitated girl in an car accident. A manifestation of evil.

IV. Life without the net. Telescope. Restart. Avoid addiction. In south korea they use diapers no to loose points in the game.

V. The end of the net. Big flares from the sun.

2.12 history of informatics

1812. Charles Babbage. Differential machine. 1872. Ada Lovelace. Daughter of The poet Lord Byron. She invented the first language eith perforated cards. 1889. Herman Hollerith. Automatization census. Then 1924, IBM. 1907 vacuum valve 1936. Universal Turing machine. 1939. Bombe machine. To decipher enigma. 1946. ENIAC. Electronic Numerical Integrator and Computer. Ballistic calculations. 1947. Transistor. First assembly language with von Neumann. 1960 NASA. Fortran. 1977. John Backus. Alan Turing award. Can machines be liberated from the von Neumann style ?

Chapter 3

Operations and von Neumann style

3.1 Introduction

machines and style. instructions.....

Example of code 3*2

Turing Universal machine Neumann Architecture Church lambda Backus Fortran and its paper

3.2 Roots of a second degree equation

In this section, a program to obtain the roots of a second order equation is presented:

$$ax^2 + bx + c = 0, \quad \forall a, b, c \in \text{Re}.$$

The fundamental theorem of algebra states that every Nth order polynomial has N complex roots. If the coefficients are real, then the roots are complex conjugate. Dividing the above equation by a and looking for a perfect square, the following equation allows is obtained:

$$\left(x + \frac{b}{2a}\right)^2 - \frac{b^2}{4a^2} + \frac{c}{a} = 0.$$

Solving the unknown x , the well known formula for the roots is obtained:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (3.1)$$

If the discriminant $d = b^2 - 4ac$ is less than zero, roots become complex. In the following code, complex solutions given by (3.1) are implemented. Note that the discriminant d was defined as a complex variable to avoid math problems when the discriminant is negative. Whereas the root of a real negative number is not defined, the root of a negative number has a value when considering working with complex numbers.

```
subroutine Roots_2th

  real :: a, b, c      ! coefficients
  complex :: x1, x2, d ! roots and discriminant

  a = 1.; b = 0; c = 1

  if (abs(a)==0) then
    if (abs(b)==0) then
      write(*,*) "There is no solution "
    else
      write(*,*) "There is only one solution x1 =", -c/b
    end if
  else
    d = b**4 - 4*a*c
    x1 = ( -b + sqrt(d) )/( 2*a )
    x2 = ( -b - sqrt(d) )/( 2*a )

    write(*,*) "There are two solutions "
    write(*,*) " x1 = ", x1
    write(*,*) " x2 = ", x2
  end if

end subroutine
```

Listing 3.1: Roots.f90

Chapter 4

Imperative versus declarative programming

4.1 Introduction

A big number of programming languages exist nowadays, all of them covering different features in order to fit diverse needs. While a programming language is focused on the way the code is organized, other language gives importance to the dataflow through operations. Consider as an example the object-oriented paradigm, one of his main features is that the data is organized into objects that are modified by methods also specified in the object.

A group of these features define the conceptual framework of the language, its programming paradigm. Think of it as a group of ideas that describe how to write a program. Some languages are based only on one paradigm while others are multi-paradigm so their programs are too.

A big amount of programming paradigms are commonly grouped into two types: imperative and declarative. This chapter covers the difference between both supported on examples of numerical sum series.

In an imperative style, the programmer tells the machine **how** to perform each task step by step, hence, an statement describes how to change the state of the program. However, in a declarative paradigm, the programmer tells the interpreter/compiler **what** to obtain and its properties but not the control flow of the

process.

One of the paradigms grouped into the declarative programming is the functional paradigm, which is treated and used throughout this book. Its properties are covered later in the Advance Programming chapter. Try now to glimpse with these examples some of its advantages comparing it to the imperative programming. Ease of code reuse for future projects and the possibility of independent validation of pieces of code are just two examples of the potential of this kind of programming.

4.2 Sum of numerical series

One of the main tasks of numerical calculus is to sum numerical series which appear from approximation of functions to numerical integration.

The algorithm to sum these series can be done in an iterative way by a sequential machine or von Neumann machine or concurrently by a bunch of processors. This first idea allows to introduce the two different paradigms mentioned when trying to write or to implement algorithms in different programming languages: imperative paradigm and declarative paradigm.

Imperative programming is related to an ordered sequence of sentences that allows to obtain the solution whereas declarative programming is concerned with the specification or the statement of the problem no matter the way it is accomplished.

The following sum series are treated in the Fortran project included with this book:

$$S = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{2^n}, \quad S = \sum_{n=1}^{\infty} \frac{1}{n!}, \quad S = \sum_{n=1}^{\infty} \frac{1}{n^2}$$

The first two examples have a good convergence, this means that the general term an tends to zero quickly with $n \rightarrow \infty$ and few terms must be added to get a correct result. Then, it is easy to obtain the result of the infinite sum up to a given finite number of digits. However, the third example has a poor convergence and some issues must be review to sum it in a correct way. The following chapter is dedicated to this third example and the particularities that appear when bad convergence series are calculated with a finite precision machine.

4.3 First example with imperative paradigm

Let's consider the following summation as an example to show how the imperative paradigm can be implemented.

$$S_N = \sum_{n=1}^N \frac{1}{n^2}$$

In the following code, the summation is done in an iterative way.

Listing 4.1: `Sum_series.f90`

The result for $N = 10$ is obtained by calling the function in the following way:

```
write(*,*) " S = ", SummationN_n2( 10 )
```

Notice how the main task to develop (sum a value to the older value of `S`) is specified to the compiler with a loop. Since this function is simple, the imperative style used is easy to read and easy to learn for beginners. However, with more complex tasks, the code quickly becomes extensive and difficult to follow and maintain.

If you are curious about the need of including `real(i)` in the denominator of the sum, this issue is treated in the section 4.7 and also you can take a look at the chapter 2.3. Just forcing the number `1.` of the numerator to be real, the operations are performed in the reals field so the result is properly calculated, converting the denominator to real should not be needed. However, there is another interest in doing that. As a clue, overflow of integers must be avoided.

4.4 First example with declarative paradigm

Now the same sum is coded with declarative programming. On one hand, the abstraction of summation is used for the function `Sigma_N`, an easily reusable code has been created and once validated it will be used for future similar projects. On the other hand, the general term of the sum is independently declared in a different function, hence, if the program specifications change (a different sum is needed), updating this code is extremely easy.

Listing 4.2: `Series.f90`

Listing 4.3: `Series.f90`

Listing 4.4: `Sum_series.f90`

The result for `N = 10` is obtained by calling the function in the following way:

```
write(*,*) " S = ", Sigma_N( a1, 1, 10 )
```


The essence of the declarative programming in this example are the lines:

```
S = sum( [ (a(i), i=i0, N ) ] )
```

```
Sigma_N( a1, 1, 10 )
```

Both lines tell the compiler the properties of the solution to be obtained, no matter how the process is done. In the first case the intrinsic function `sum` performs the summation of some elements by means of an implicit loop from one value to another. The properties of the solution sought are of course the limits of the sum (`i0` and `N`) and the type of term to be added. In the second case, thanks to the created function `Sigma_N`, the sum is performed just by declaring the general term to add and the limits, the programmer does not need to tell the compiler how to do the summation step by step.

Another ingredient is needed for this example, since the function `Sigma_N` admits any general term of a series, it needs to know how to communicate with the function that defines that general term `a1(n)`. Through the interface block (intentionally created defining a function from integers to reals) `Sigma_N` knows everything about the dummy arguments used inside the referenced procedure `a1(n)`. The same procedure could be recycled next time a function of the following form is used:

$$\begin{aligned} \mathbf{f_N_R}: \mathbb{Z} &\rightarrow \mathbb{R} \\ n &\rightarrow \mathbf{f_N_R}(n) \end{aligned}$$

4.5 Second example with imperative paradigm

Let's consider another series summation example with imperative paradigm. In this case the result sought is the summation of the infinite terms of the series.

$$S = \sum_{n=1}^{\infty} \frac{1}{n^2}$$

In the following code, the summation is done in an iterative way.

Listing 4.5: `Sum_series.f90`

This piece of code performs the summation of as many terms as the variable **S** can significantly sum to itself. Once the **nth** term of the sum is not going to modify the value of **S** due to its finite number of digits, the result obtained is the closest value to the infinite sum that **S** can hold (with this specific algorithm). Then, **n** is returned indicating the last term calculated.

The sum must be stopped once the result of **an** is smaller than the smallest number that can be added to **S**. Then, the result of adding **S + an** is equal to **S**. Notice that the calculus of **S** from one step to the next one is the same to the first example. However, now **Sn** is used to compare the value before and after the **nth** sum. If both values are the same, it means that **an** has not contributed to **S** and the process is stopped. Also, take into account the need of initializing **S** and **Sn** with different values.

The result of the sum is obtained by calling the function in the following way:

```
integer :: n

write(*,*) " S = ", Summation_n2( n )
```

4.6 Second example with declarative paradigm

For this example a modification of `Sigma_N` is needed: `Sigma`. Nevertheless, notice the simplicity of the code when the existing function `Sigma_N` is reused.

For an infinite sum it is more interesting to declare the minimum term to add to the sum (`eps`) instead of the number of terms to sum (`N`). The programmer does not normally know how many terms are needed to reach a finite number of digits in the result. However, the allowed error of the result is usually a specification. For this reason, `N` is automatically found by specifying the general term and the minimum term to sum with the function `Tail(a, eps)`. Then, `Sigma_N` is used to perform the sum as seen in the previous example.

Once again, the operation is performed by means of a desired properties bunch and not declaring the control flow step by step. This strategy increases the capabilities of solving many different problems with the same tools by creating more and more complex functions always based on lower level functions (in a hierarchical structure). In this case for example, a new degree of freedom (`eps`) is obtained by using `Sigma`, which is based on `Tail` and `Sigma_N`. This programming paradigm also leads to a reuse strategy, where the next time that a similar task is performed in another program, the same functions can be used thereby saving time.

Listing 4.6: `Series.f90`

Listing 4.7: `Series.f90`

Do not forget to declare the same interface block of the first example `f_N_R(x)` and define the general term to sum (`a1` for example), then, the following statements obtain the infinite sum for a minimum general term of `eps = 1e-10`:

```
real :: eps

eps = 1e-10
write(*,*) " S = ", Sigma( a1, 1, eps )
```

The mathematical support for the function `Tail` is treated in the section 4.7. It is essential to notice three issues:

1. This function is written for good convergence series since its definition is based on obtaining N such as the integral from $i = N+1$ to infinity of a_i is equal to ϵ . Read the following section for more information.
2. The stop condition is established on the last two terms to be added with the line:

```
abs( a(i) + a(i+1) ) > 2*eps
```

This is done so alternating sums also reacts to this criteria. If this were not done, an alternating sum with very different odd and even terms could stop prematurely.

3. Also, a maximum allowed sums are permitted with the line:

```
.and. i < Nmax
```

4.7 Numerical issues

4.7.1 Theoretical framework

From the numerical point of view, there are some important issues to revise. Let's try to explain these issues with an example. Consider the previous sum of infinite terms:

$$S = \sum_{n=1}^{\infty} \frac{1}{n^2}.$$

The difficulty in this case comes with the infinite number of terms to be added. It is clear that the program will try to yield an approximate number for this problem and the difficulty will be associated to those series with low convergence rate.

1. Convergence of the series.

The rate of convergence of this series depends on the velocity the general term $a_n = 1/n^2$ tends to zero with $n \rightarrow \infty$. In this special case, n should be very big to approach a_n to zero and many terms of the series should be added to obtain the right result. This issue would slow down the process of the summation. Besides, another issue appears associated to the finite precision of computers.

2. Overflows associated with integer variables.

Another interesting issue that we can encounter when running a loop is overflow of variables. A real variable can hold a huge number and an overflow is unlikely reached. However, since the maximum of an integer variable of 4 bytes is around 2×10^7 , a simple operation such as $(100)^5$ can give rise to an

overflow if the operation is done with integers. Taken into account that the maximum value of a real variable of 4 bytes is around 3×10^{38} and to avoid this issue, the operation $(100)^5$ can be done with real numbers. This is done in the summation example by the following line:

Listing 4.8: `Sum_Series.f90`

3. Round-off errors associated to significant digits.

Since the numbers in the computer are operated with a finite precision arithmetic, real numbers do have an approximate representation. This means that they are approximated by a finite number of digits. Namely, single precision variables are approximated with 7 digits and double precision variables with 16 digits. Hence, even real number can be very small, the smallest real number ϵ that can be added to 1 should be greater 10^{-15} . It is interesting to bound the error committed due to leaving this infinite number of terms out of the sum. This forces to stop the summation process when round-off is encountered as it is done by the following line in the summation example:

Listing 4.9: `Sum_Series.f90`

4. Round-off errors associated to the summation process.

Another source of error appears due to the same fact explained above, the finite precision floating point arithmetic (each of the sums is performed and stored with a finite number of digits; 7 or 15). When only one sum is made, this error remains bounded, however, after millions of operations it can grow. The total error of the result can be divided into truncation error E_N and round-off error E_{fl} as follows:

$$E = E_N + E_{fl}.$$

Since only N terms are added, and when considering exact precision arithmetic, the error is due to those terms from $N + 1$ to infinity which are not included. However, when floating point precision is considered, round-off errors are accumulated after a huge amount of operations to yield an additional round-off error E_{fl} . An analytical effort is needed to bound the truncation error E_N . It is important to discuss the magnitude of those two errors to optimize the computational effort associated to N .

5. Truncation error.

The number of terms N that the algorithm will sum depends mainly on the convergence rate of the general term a_n . To know the error associated to the finite sum S_N , the remaining terms should be evaluated.

$$S = S_N + \sum_{n=N+1}^{\infty} a_n$$

To obtain an upper and lower bound of this error the integral test is used. Let's consider the real function $f(x)$ that verifies $f(n) = a_n$ for all n . This function is represented in figure 4.1. In the same figure, the image of $f(x)$ for different natural values n is plotted. These images allow to draw rectangles of height $f(n)$ and base one. From figure 4.1, the integral from n to $n+1$ can be bounded by the area of two rectangles of height $f(n+1)$ and $f(n)$ by the following expression:

$$f(n+1) \leq \int_n^{n+1} f(x) dx \leq f(n) \quad (4.1)$$

Since the error $S - S_N$ is given by:

$$S - S_N = \sum_{n=N+1}^{\infty} f(n)$$

and using equation (4.1), a lower bound for the error is obtained:

$$S - S_N \geq \sum_{N+1}^{\infty} \int_n^{n+1} f(x) dx = \int_{N+1}^{\infty} f(x) dx.$$

In the same manner, an upper bound is given by:

$$S - S_N = \sum_{n=N}^{\infty} f(n+1) \leq \sum_N^{\infty} \int_n^{n+1} f(x) dx = \int_N^{\infty} f(x) dx$$

With these bounds, the rest $S - S_N$ can be estimated by the following expression:

$$\int_{N+1}^{\infty} f(x) dx \leq S - S_N \leq \int_N^{\infty} f(x) dx \quad (4.2)$$

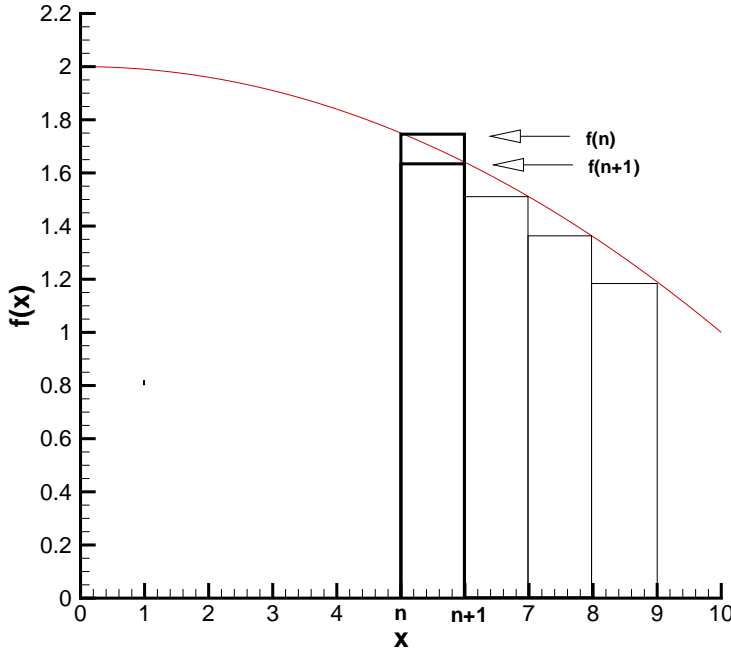


Figure 4.1: Relation between the integral of $f : \mathbb{R} \rightarrow \mathbb{R}$ and a sum of a numerical series with general term $f : \mathbb{N} \rightarrow \mathbb{R}$.

4.7.2 Mathematical support for Tail function

Notice that the function `Tail` is specially created for good convergence series. It is based on the premise that:

```
! Obtain N such as:
! integral from i=N+1 to infinity a_i = eps
```

For a good convergence sum (the general term tends to 0 quickly), stopping the sum when `an` reaches a value `eps` leads to a situation where the truncation error committed is not important compared to the value `eps` itself. The infinite terms not included in the sum would not appreciably modify the result. For a sum series with a poor convergence the situation is the opposite, stopping the sum when the general term is less than `eps` leaves without adding a huge number of terms whose sum is comparable to `eps` or even much more higher. Then, the total error committed could be impermissible and there resides the difficulties of calculating

these series.

Said in other words, the value `N` that `Tail` calculates is based on reducing the truncation error for good convergence series, not considering that it can be much higher than expected if round-off is reached before the truncation error is permissible.

4.7.3 First test for bad convergence sum

For the example we are treating here the bounds of the error are:

$$\frac{1}{N+1} \leq S - S_N \leq \frac{1}{N} \quad (4.3)$$

Being `S1` the result of the sum in the computer and using its analytical result, the total error of the sum is $\pi^2/6 - \text{S1}$. Then, with the above bounds, the round-off error of the summation gives:

$$E_{fl} = E - E_N = \left(\frac{\pi^2}{6} - \text{S1} \right) - 1/N$$

Let's execute now the following example to test this bad convergence sum series:

Listing 4.10: `Sum_Series.f90`

First, take a look at the result of the computer if the summation process is stopped according to the round-off errors associated to significant digits. Notice that this result is obtained after 94906265 sums.

```
Summation 1/n**2
S = 1.64493405783458    N = 94906265 Error = 9.013651380840315E-009
```

Now force the computer to calculate the same sum up to 200000000 terms, around twice the number of operations than before, notice that the total error is exactly the same than in the previous case.

```
S = 1.64493405783458    N = 200000000 Error = 9.013651380840315E-009
```

If both errors are balanced thanks to the bounds calculated, the following results are obtained for the truncation and round-off errors (for $N = 94906265$ terms added):

```
Truncation error S =    1.053671197498475E-008
Round--off error   =   -1.523060594144434E-009
```

Two interesting conclusions are obtained for this example. Firstly, the round-off error is quite smaller than the error associated to the truncation. Secondly, the error is much higher than the finite number of digits that the computer is handling with the sums (in this case around 15 decimal digits). An infinite number of non-negligible terms are omitted due to a bad convergence rate.

4.7.4 Second test for bad convergence sum

Now, the same sum is performed with declarative paradigm, calculate the result for a value `eps = 1e-10` and calculate the total error as usual:

```
real :: S, eps, E

eps = 1e-10
S = Sigma( a1, 1, eps )
E = PI**2/6 - S

write(*,*) " Summation 1/n**2 "
write(*,*) " S = ", S, "Error = ", E
```

The following result is obtained, a total error of around $1e-5$ is committed when the computer is performing sums with terms around $1e-10$.

```
Summation 1/n**2
S =    1.64492406689824      Error =    9.999949984074163E-006
```

A question may appear now; which `eps` should be introduced to `Sigma` so the result is calculated until round-off is obtained? The answer is `eps = epsilon(S)/2.`, however, some notions of floating-point representation are essential to deepen in this topic. An extensive discussion regarding the consistency of using `epsilon(S)` for this example and the generalization for any other sum is treated in the section 3 of Part II of this book. Also, why dividing it by 2 is needed becomes clearer.

4.7.5 How can I improve this calculation?

There is a different way of calculating this sum with better accuracy in the computer:

1. Using higher precision for the variables, for example quadruple precision for the variables involved.
2. Change the algorithm to perform the sum. A non-intuitive way of calculating the sum comes from the fact that adding in floating-point arithmetic is not an associative operation.

Let's perform the same sum backwards, also with double precision and from the lower terms to the higher ones using 1000000000 terms, more than 10 times the maximum terms used for the forward sum.

```
Summation 1/n**2
S = 1.64493406584823    N = 1000000000 Error = 1.000000082740371E-009

Truncation error S = 1.000000000000000E-009
Round --off error = 8.274037093680878E-017
```

Surprisingly, just performing the same sums from the lower terms to the higher ones the limit of terms found before can be exceeded and the result is ten times more accurate than before. As a general rule, when adding numbers with a big difference in magnitude order, try to sum first the lower values and then add the result to the higher values. Why this improve the result is treated in the section 3 of Part II of this book.

4.8 Convergence rate

One of the main focus of the numerical approximation techniques is to attain expansions with high rate of convergence. When dealing with numerical series, it is desirable to obtain a precise result by adding the smallest number of terms to reduce CPU time. However, the number of terms to obtain the precise accuracy depends on the convergence rate of the series. In other words, the behavior of a_n with $n \rightarrow \infty$ defines the number of terms to be added.

In figure 4.2, a_n versus n is represented in log-scale. In the same context of discretization techniques, if $a_n = O(1/n^q)$ for a given number q , the convergence is algebraic. If $a_n \ll O(1/n^q)$ for all q , the convergence is spectral. This is shown schematically in figure 4.2.

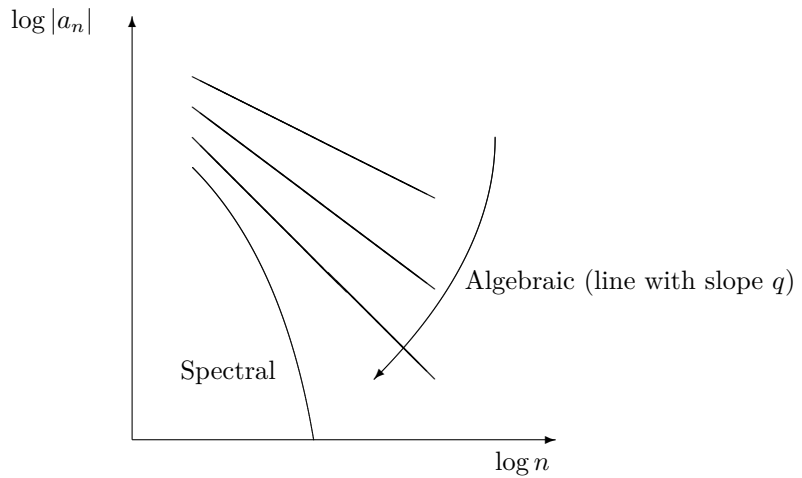


Figure 4.2: Convergence of a_n with n . Algebraic and spectral convergence.

Chapter 5

Operations with vectors and matrices

5.1 Introduction

This chapter covers two independent topics through examples. In the first place, some essential operations of matrix arithmetic are introduced, an operation like the product of two matrices becomes extremely simple in programming languages oriented to vector programming. In the second place, the concepts of static and dynamic data objects are covered. Notice that all the examples are written according to a declarative programming, consider reproducing the same results with an imperative programming to compare.

Matrix arithmetic

In the context of scientific programming some features like a syntax close to mathematics or functional oriented programming are desirable for a programming language. In addition, array programming is an essential characteristic. It makes reference to the possibility of operating a whole group of values at the same time, hence, the operations applied to a number can also be applied to vectors, matrices and higher dimension arrays in a easy to write (near to maths) expression greatly simplifying maths computations.

Languages like Fortran, MATLAB, R or the NumPy extension of Python support array programming. Matrix arithmetic are built-in in these languages and

expressing the mathematical language in a natural way is feasible.

Take note of the difference between array programming and array processors. The first makes reference to how the programmer codes the mathematical operations in its program (a big advantage is obtained from this feature for Scientific Computing as mentioned before). The second feature is related to how the processor operates that group of numbers, by performing all the operations together under the same instruction given to the processor in a considerably increase of speed. Both features suppose an increase of performance for the coding and executing of scientific programs, however, this chapter is oriented to the aspects of the first feature mentioned.

It is relatively new the development of array programming languages being Fortran the first one to include it in the 90's with the ISO/IEC standard 1539:1991.

The following concepts regarding arrays constructions and operations are covered:

1. Type, rank and dimension of arrays.
2. Constructors to initialize an array.
3. Sectors or slices of arrays.
4. Operations among array.
 - (a) Addition.
 - (b) Dot product.
 - (c) Matrix multiplications.
 - (d) Hadamard product.

Static/Dynamic data objects

Each data object declared in a program (variables, constants, pointers, arrays, etc.) are either static or dynamic which means that the memory storage to hold that piece of data is reserved during compilation time or during the execution of the program respectively. Consider the variable `real :: A(N, N)` declared at the beginning of a code, its memory storage is reserved when the program is compiled and this space is not liberated until the program has finished the execution.

Another option to declare and manage the memory storage of a object is dynamic allocation. In this case, the storage of a variable like `real, allocatable :: Ak(:, :, :)` is not reserved until the program orders it (during execution) and can be changed or freed at any moment.

5.2 Static size vectors and matrices

For the following example some basic matrix arithmetic is performed with static data objects. Since the sizes of the arrays are known at compile-time, the memory storage can be declared statically. The main properties of a static allocation are:

1. The memory address and the size are assigned during the compilation of the code in the executable image of your program.
2. These address and size can not be changed during execution.
3. Once the program finishes the execution the space is freed.
4. It is a simple and quick allocation process.

Consider the vectors $V, W \in \mathbb{R}^N$ and the matrices $A, B \in \mathcal{M}_{N \times N}(\mathbb{R})$ defined in the following way:

$$V = \left[v_i = \frac{1}{i^2}, \quad i = 1 \dots N \right],$$

$$W = \left[w_i = \frac{(-1)^{i+1}}{2i+1}, \quad i = 1 \dots N \right].$$

$$A = \left[a_{ij} = \left(\frac{i}{N} \right)^{j-1}, \quad i = 1 \dots N, \quad j = 1 \dots N \right].$$

$$B = A^T$$

Some basic notions are summarized now:

- An array is properly declared when it has type, rank and dimension (or extent). All this information comes from the mathematical definition of the vectors and matrices.
 1. The data type in this case is **real** since the examples are built with real vectors/matrices.
 2. Rank is the number of dimensions in the array; a rank-one array represents a column vector (**V** or **W**), a rank-two array represents a matrix organised into columns and rows (**A** or **B**), etc.
-

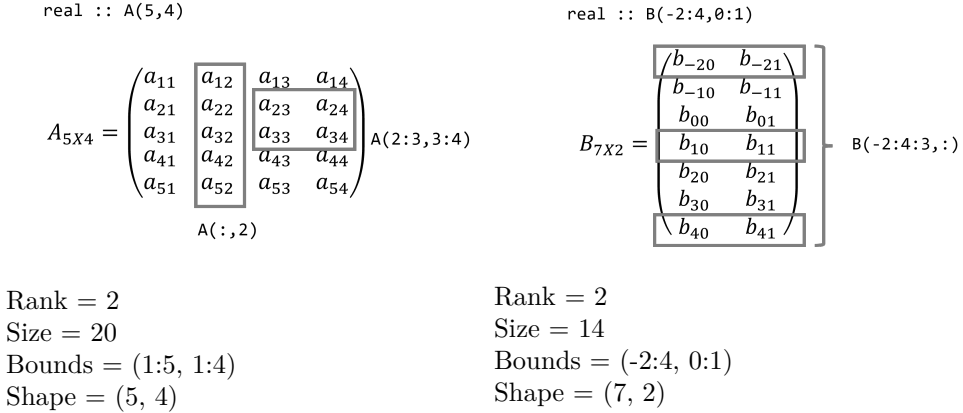


Figure 5.1: Properties of a couple of arrays.

3. The extent of each particular dimension is its length, which means, the number of elements in that dimension. All dimensions involved in these examples are equal to N . The bounds of a dimension does not have to start with index 1, later some examples with different bounds are shown.
- Once declared, the initialization of the arrays is performed with constructors. In the case of Fortran the constructors are only used for rank-one arrays so functions like **reshape** are needed for higher ranks. Three ways are normally used to manually construct an array:
 1. By a list of values: `(/ list /)` or `[list]` where 'list' is full of values separated by commas (of the same type declared for the array). For example: `V = [3.4, 5.2, 4.5, 2.1]` is a column vector of dimension 4 with those values in each position.
 2. An array expression can also be used in the initialization: `V = [B(2, 3:5)]` which stores in `V` the list of values in the second row of `B` from the column 3 to 5.
 3. Using an implicit loop so a list of elements is computed from a loop controlled by a DO variable. Lots of examples of this are used throughout this book: `V = [(1./i**2, i=1, N)]`.
 - The sectors of an array are already introduced in this text. Notice that the initialization of `V` in this example: `V = [B(2, 3:5)]` use a sector of the whole array `B`. This can also be extended to a whole dimension of an array, i.e. the column 5 of an array `C` is referred like `C(:, 5)`. Furthermore, alternate values can be selected by specifying a lower and upper bound and the jump between values (see matrix `B` of Figure 5.1): `B(-2:4:2, :)` being the second value 2 the jump between rows.

- Some operations to be used with real matrices are introduced now. Notice that these examples are limited to numeric arrays, reals or even integers, but arrays can be constructed with a different data type. Furthermore, different operations could be created for new data types also created by the programmer.

Consult the specifications of the following matrix operations to become familiarized with the purpose, inputs and optional arguments allowed. As a quick reference: use `sum` to add all elements of an array, whether it is across one dimension, all the dimensions or only those elements that accomplish with some condition (e.g. $V_i > 0$). Use `dot_product` to calculate the scalar product of two vectors, `matmul` for a matrix multiplication and `transpose` to calculate the transpose of a matrix or vector. In addition, `maxval` and `maxloc` return the maximum value of all the elements of an array and its position, whether it is across one specific dimension, the whole array or only for those values that accomplish with a specified condition.

It is mandatory that rank and dimension of the arguments of these intrinsic functions must agree with the mathematical definitions of the operations, consider for example how the scalar product of the vector V with a column of A is calculated. The colon symbol `:` is used as an implicit loop for all the elements of column N in the matrix.

```
subroutine Matrix_operation_examples()

integer, parameter :: N = 10
real :: W(N), V(N), A(N,N), B(N,N)
integer :: i, j

V = [ ( 1./i**2, i=1, N ) ]
W = [ ( (-1)**(i+1)/(2*i+1.), i=1, N ) ]
A = reshape( [ ( ( i/real(N))**(j-1), i=1, N ), j=1, N ], [N, N] )

write(*,*) " 1. Sum ( V ) = ", sum(V)
write(*,*) " 2. Sum ( A ) = ", sum(A)
write(*,*) " 3. Sum ( V, V>0 ) = ", sum(V, V>0)
write(*,*) " 4. Sum ( A, A<0 ) = ", sum(A, A<0)
write(*,*) " 5. dot product ( V, W ) = ", dot_product(V, W)
write(*,*) " 6. dot product V and A(:,N) = ", dot_product( V, A(:,N) )
write(*,*) " 7. mat multiply A times V = ", matmul( A, V )

write(*,*) " 8. my transpose (A) = "
B = transpose(A)
do i=1, N
    write(*,'(100f8.3)') (B(i, j), j=1, N)
enddo

write(*,*) " 9. maxval (A) = ", maxval(A)
write(*,*) " 10. maxloc (A) = ", maxloc(A)
```

end subroutine

Listing 5.1: `Matrix_operations.f90`

Take some interesting notes of the previous example. First of all, `N` is declared as a named constant thanks to the `parameter` attribute. Then, its value is fixed once for all the execution and a try to change it will end in a compilation error. Its value is automatically used to declare the size of the vectors and matrices involved in the example.

Secondly, to initialize both `V` and `W` the mathematical definition of each component is written by means of an implicit loop (declared with parentheses). Two implicit loops are used in the definition of the matrix `A`, the nested loop `i` calculates the rows while `j` jumps from one column to the next one. Both loops define a $N*N$ rank-one vector that needs to be reshaped to a $\mathcal{M}_{N \times N}$ matrix. The function `reshape` organizes the components of the vector into columns.

Thirdly, take into account the difference between these operations and the element wise operations that can be performed with vectors and matrices. Some operations like the addition of two matrices are performed by adding the corresponding elements of both, also, many elemental operations on numbers can also be applied to matrices, so the result is an equal shaped array with the results of applying the operation to each element. Test the following expressions for matrix addition, Hadamard product (multiplying the corresponding elements of both matrices), cosine or square root of all elements of `A`:

```
A + B
A*B
cos( A )
SQRT( A )
```

Finally, take a look at the way the matrix `B` is written in the screen, by using two loops each row of the matrix is represented element by element. Compare this way with the simple output of the line `write(*,*) B` where all the elements are represented by columns and not by rows. This is due to the fact that programs store multidimensional arrays in a linear storage and Fortran is a column-major order language where the consecutive elements of a column reside next to each other.

5.3 Dynamic allocation of vectors and matrices

Consider now that the size of the matrix involved in your code is not known at compile-time, maybe it comes from an user input, an external file or it is the result of a previous operation. In this case, dynamic data objects are used so the memory storage (address, size, etc.) for the object is allocated or modified during the execution of the program. The essential statements to manage the memory for this data objects are **allocate** and **deallocate** while the attribute for the data object is **allocatable**. The main properties of dynamic data objects are:

1. The program decides how much memory reserve for a data objects so it can accommodate any size with no need of re-compiling the code.
2. Modifications for the memory size are allowed.
3. It becomes the programmer responsibility to liberate the memory reserved for non-used arrays in order not to run out-of-memory.
4. It is generally slower than static allocation.

Given the square Vandermonde matrix of $M \times M$:

$$A_M = \left[a_{M_{ij}} = \left(\frac{i}{M} \right)^{j-1}, \quad i = 0, \dots, M-1, \quad j = 0, \dots, M-1 \right],$$

determine the following operations:

$$S = \sum_{M=1}^{10} \text{Tr}(A_M), \quad S = \sum_{M=1}^5 \text{Tr}(A_M^2), \quad S = \text{Tr} \left(\sum_{k=0}^5 A_M^k \right), \quad M = 8$$

The following examples are supported by the functions: **Vandermonde(M)** matrix of a given dimension **M**, **trace(A)** trace of a matrix and the recursive function **power(a, k)** to obtain the kth power of a matrix. Once each function is coded, general operations like these proposed here are easy to implement, remember that each abstraction can now be used in any program where this algebra is needed from now on.

```
subroutine Matrices_allocation()

  real :: S                                ! sum
  integer :: M                             ! Vandermonde dimension
  real, allocatable :: Ak(:, :, :)        ! rows, columns, kth power
  integer :: k                             ! index k of power
```

```

S = sum( [ ( trace( Vandermonde(M) ), M=1, 10) ] )
write(*,*) "1. sum from M=1 to 10 of trace ( A_M ) = ", S

S = sum( [( trace(matmul(Vandermonde(M), Vandermonde(M))), M=1, 5) ])
write(*,*) "2. sum from M=1 to 5 of traces ( A_M **2 ) = ", S

allocate( Ak(8, 8, 0:5) )
do k=0, 5
  Ak(:, :, k) = power( Vandermonde(8), k)
end do

S = trace( sum( Ak, dim=3) ) ! trace( I + Ak + Ak**2 +... Ak**5 )
write(*,*) "3. trace ( sum from k=0 to 10 of A_5**k )=", S

end subroutine

```

Listing 5.2: `Dynamic_allocation.f90`

Let's take a deeper look into the program. First of all, notice that the 3-rank object `Ak` is declared with the attribute `allocatable` and colons `:` instead of the dimension specifications. Later, when the dimensions are known, the `allocate` function is used to reserve the appropriate memory for the variable.

Consider now the first operation to perform. The expression `trace(Vandermonde(M))` is coded inside an implicit loop from 1 to 10. This results are constructed in a vector with squared brackets and `sum` performs the summation of the components of that vector. The mathematical abstraction summation is already implemented in many programming languages.

The use of many variables is avoided here thanks to a declarative programming. In an imperative programming the Vandermonde matrix would have been stored in a variable `AM(M,M)`, its trace in a real vector variable used as an input for `sum`. Do not think that these objects are not used now, the compiler needs them exactly the same in order to operate, however, it automatically reserves memory, stores the intermediate results, returns only the single result needed `S` and, when the calculation is finished, frees all that temporary memory used in an efficient way.

The second operation is similar with the exception that now the trace is calculated on the product of two matrices. Fortran, like any common scientific language already has implemented the product between two matrices of reals. As it has been mentioned before, the vector/array programming does not only have advantages in the coding of algebra computing, it has also a better performance when calculating those multiple operations when vector processors are used, so the same operation among a bunch of numbers is performed efficiently.

For the third operation the variable `Ak` is dynamically allocated. This 3-rank array is used to store each power of the Vandermonde matrix with bounds 0 to 5

in the third dimension. While normally the indices of any vector or matrix start in 1, is decision of the programmer to modify it so the index automatically responds to a mathematical sense, in this case writing $Ak(:, :, 3)$ is a reference to the third power, with no need of remembering in which index starts and how many powers are we calculating. If the needed powers of Vandermonde would only have been 4, 5 and 6, we could have allocated the matrix like: `allocate(Ak(8, 8, 4:6))`.

Take note of the full array assignation performed inside the explicit loop. Once the matrix Ak is properly allocated, writing $Ak(:, :, k) =$ is enough to store the result of the k th power of Vandermonde matrix since it is known that the result is an 8×8 matrix.

```
real function trace( A )
  real, intent(in) :: A(:, :)

  integer :: i           ! row of matrix A
  integer :: N           ! dimension of matrix A

  N = size(A, dim=1)
  trace = sum( [ (A(i,i), i=1, N) ] )

end function
```

Listing 5.3: Dynamic_allocation.f90

```
recursive function power(A, k) result(B)
  real, intent(in) :: A(:, :)
  integer, intent(in) :: k ! kth power
  real :: B( size(A,dim=1), size(A,dim=1) )

  integer :: i, N

  N = size( A, dim=1)

  if (k==0) then
    B = Identity(N)
  else
    B = matmul( power(A,k-1), A )
  end if

end function
```

Listing 5.4: Dynamic_allocation.f90

For the functions `power(A, k)` and `trace(A)` two concepts should be extracted. Firstly, both must be used with square matrices, where both mathematical operations are defined. Secondly, the concept of recursion is used to calculate the k th power of a matrix. Essentially, the k th power of a matrix is the multiplication of that matrix with his $k - 1$ power. A `recursive` statement is used for the function declaration so the compiler knows that this function can call to itself in order

to calculate smaller instances of its main functionality, which means, it is going to call itself to calculate the $(k - 1)$ power once it tries to compute the k th power and so on until the power needed is 0 so the result is the identity matrix.

5.4 Memories: Static, Heap, Stack

It is clear that during the execution of the program certain amount of memory is needed. Data objects or the source code must be stored somewhere. The compiler, the linker and the operating system of the machine decides where each piece of data is stored. Three regions of the memory can be distinguished for a program (see Figure 5.2): static, heap and stack, each one related to one type of allocation. Notice that the last two memories are dynamic in nature.

Static and dynamic allocation have been already treated. A third way to allocate memory size for data objects is used: stack (or automatic) allocation. The compiler is constantly using it in order to store temporary arrays used inside sub-routines/functions or needed for array expressions.

Let's revise some properties, advantages and problems associated to these memories:

- **Static:** it is usually located in the low end of the memory reserved for the application. The compiler creates a list of variables to be allocated during compilation and gives them a fixed address so the whole program can use these variables at any time and faster.

This static allocation requires knowing the amount of memory needed at compile-time.

- **Heap:** when a data object is allocated, the application requests an amount of memory to the system, if there is space available the system answers with the starting address where storing the data. Once that memory is not needed any more, the program can ask the system to free that memory under the `deallocate` statement (in the case of Fortran) so that space becomes available for the next time. Notice that the amount of space for a program is bigger than the stack, but not unlimited, here concepts like virtual memory and swap play an important role.

For the heap a memory leak can happen if no deallocation is performed (the memory continue being used with not useful data) and it is normally slower than static allocation. However, the programmer manages this space and the program decides how much memory to use for each purpose.

- **Stack:** it is composed by a limited amount of memory and a pointer that holds the current position where routines can store local variables. This
-

space is filled from the top to the bottom of the memory so any time that a subroutine or function (let's say **function A**) needs from temporary storage this space is used, the pointer is decremented and the stack space is reduced. If this function calls a nested **function B**, then its local variables are stored below variables of **A** and the pointer is decremented once again. Once the **routine B** finishes its operations, the pointer is incremented again below the spaced filled by **routine A** so that memory is available for the next routine. This structure is called LIFO; 'Last-In,-First-Out'.

The process is efficient and fast and the space is automatically freed when the routine returns to its host. This allocation is performed by the machine but the programmer usually has the option to declare some variables as **AUTOMATIC** in subprograms so they reside in the stack area. On the contrary, the amount of space is limited by the linker in the case of Windows and the programmer must be careful with allocating more space than it is available in order to avoid stack overflow.

Stack overflow, when the memory allocated on the stack overflows into other memory regions, usually happens with two situations; extremely deep (or infinite) recursion and large array variables.

The second example dynamically allocates a $400 \times 400 \times 400$ of 4-bytes reals array in the heap (**R**) but then the function `StackOverflow_size(n)` tries to locate a similar automatic array (**S**) in the stack leading to a Stack Overflow error.

```
real, allocatable :: R(:, :, :)  
integer :: n
```

```
n = 400  
allocate( R(n, n, n) )
```

```
R = StackOverflow_size( n )
```

with the function:

```
function StackOverflow_size( n ) result(R)  
integer, intent(in) :: n  
real :: R(n, n, n)  
  
    R(:, :, :) = 1.  
  
end function
```

Some strategies can be used to avoid this error:

1. Try to reduce the abuse of stack; allocate automatic arrays that usually goes to stack so they are located in heap. Then, they are automatically deallocated at the end of the routine.
2. The size of the Stack memory can be increased through a linker option. In Visual Studio for example it can be written: `/STACK:100000000` specifying the size in bytes desired.
3. A compiler option can be used to change the default storing place for automatic arrays and temporary arrays so they are automatically located in the heap: `/heap-arrays`. If a kilobytes size is specified, only larger arrays are allocated to heap (i.e. `/heap-arrays100` to only affect arrays larger than 100 kilobytes). Use the value 0 to apply the behaviour to all arrays.

Now try to execute the second example with the option `/heap-arrays0` specified in the compilation options. For the case of Visual Studio it can be done in the project properties by clicking on Configuration Properties/Fortran/Command Line and adding that line.

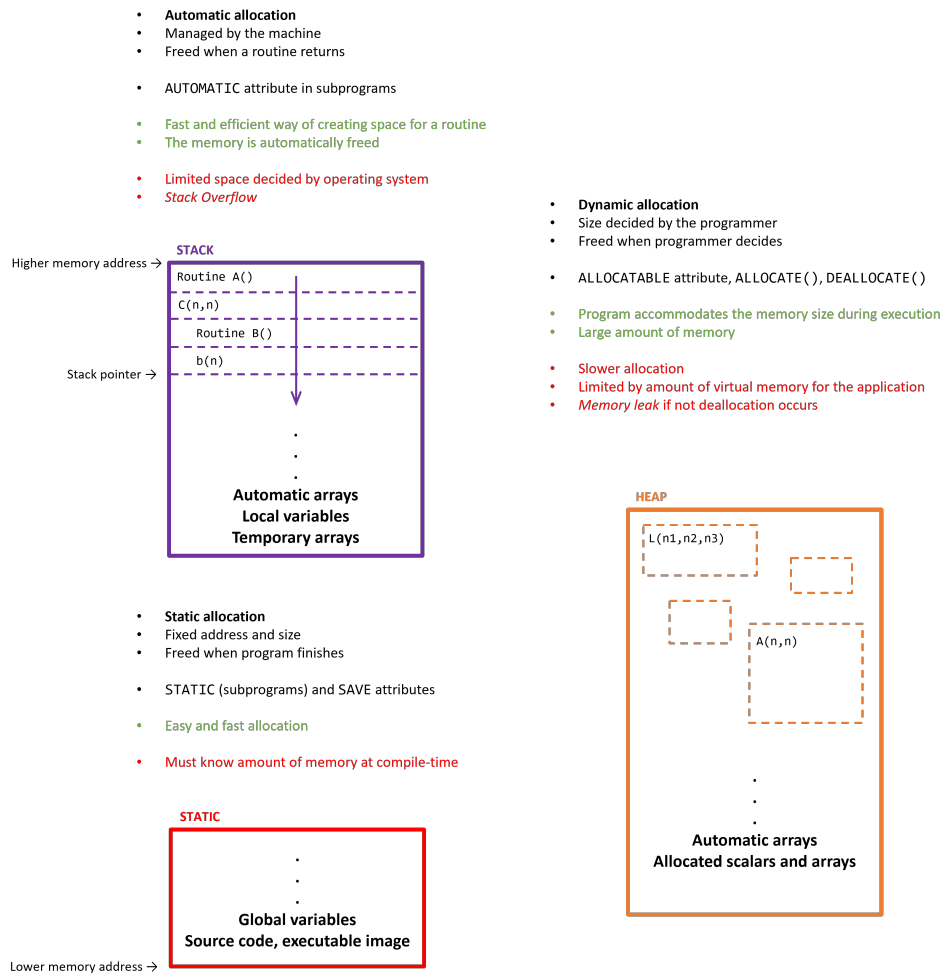


Figure 5.2: Three memory regions used by a program.

Chapter 6

Operations with functions

6.1 Introduction

This chapter covers some essential concepts that arise when the tools of mathematical analysis are used in scientific programming. The use of piecewise-defined functions, plotting functions as a means of debugging codes or the use of integrals and derivatives are the first things that a scientific code programmer will have to deal with to solve problems.

The examples presented here are developed with Fortran, however, the concepts are cross languages. The ideas around the approximation of a derivative or integral with a computer are the same in other languages and of course the need of using finite precision to represent real numbers becomes once again a matter to consider.

6.2 Defining piecewise functions

A function can be defined by means of one or more formulas, a list of values, a recurrence rule, etc. In this example, the following piecewise mathematical function is defined in Fortran using three formulas for different intervals of its domain:

$$\begin{aligned}
 f: \mathbb{R} &\rightarrow \mathbb{R} \\
 x &\rightarrow f(x)
 \end{aligned}$$

$$x \rightarrow f(x) = \begin{cases} 0, & x < -\frac{\pi}{2} \\ \cos(x), & x \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right) \\ 0, & x \geq \frac{\pi}{2} \end{cases} \quad (6.1)$$

It can be easily done through a conditional expression on the variable x so, for each point of the domain the proper result is returned. From the numerical point of view two precautions must be taken into account:

- In the first place, make sure that the entire domain of the function is represented in the code, do not forget to consider all the intervals. It is possible that the domain of the function is not the whole set $\{\mathbb{R}\}$, however, the function in the code may be called with values x out of the domain. A big topic is opened under this premise; how can be the output of an out-of-domain value managed.
- In the second place, for the algorithm the order in which we define the intervals is important. Consider what function we would be representing if the conditional ($x < \text{PI}/2$) is written in the first place and the conditional ($x < -\text{PI}/2$) in the second place for this same example.

```

function Piecewise_f(x) result(f)
  real, intent(in) :: x
  real :: f

  if ( x < -PI/2 ) then

    f = 0

  else if ( x < PI/2 ) then

    f = cos(x)

  else

```

```

        f = 0
    end if
end function

```

Listing 6.1: Integrals_derivatives.f90

6.3 Plotting functions

For a mathematical function its graph is one of the ways to represent it. Notice that for a given function $f(x)$ the set of all pairs $\{(x, f(x)) \mid x \in D\}$ (where D is the domain of the function) are unique.

This is specially useful when the domain and codomain of the function are subsets of \mathbb{R} or maybe the domain is a subset of \mathbb{R}^2 since the elements (x, y) or $((x, y), z)$ can be associated to points in a coordinate system resulting in a curve in 2 dimensions or a surface in 3 dimensions. The graph of functions, results, variables, etc. is used for “graphic debugging”, an essential way to debug scientific programs like simulations in physics. It makes much easier to find all kind of bugs in a program by means of plotting final or intermediate results obtained by the computer.

The following example is based on DISLIN, a plotting library for Fortran and C languages created by the Max Planck Institute. It is a high-level plotting library for displaying data that can be called from the main program or subroutines.

```

subroutine plot( f )
procedure (f_R_R) :: f

    integer, parameter :: M = 200
    integer :: i
    real :: x(0:M), y(0:M)

    x = [ ( xmin + (xmax-xmin)*i/real(M), i=0, M ) ]
    y = [ ( f( x(i) ), i=0, M ) ]

    call curve(x, y, M+1)
end subroutine

```

Listing 6.2: plotting.f90

The previous subroutine allows to plot a 201 points curve from a generic function declared (by means of an interface) as a:

$$\begin{aligned} \mathbf{f_R_R}: \mathbb{R} &\rightarrow \mathbb{R} \\ x &\rightarrow \mathbf{f_R_R}(x) \end{aligned}$$

It is used in combination with an initialization subroutine for the graph and a subroutine that displays the result (the Fortran solution that accompanies this book shows some examples of how to use this subroutines and DISLIN libraries):

```
subroutine plot_ini( xmin_, xmax_, ymin_, ymax_ )  
  
subroutine plot_show( )
```

Notice how this subroutine makes use of the **parameter** **M** that needs to be converted to **real** when the grid **x** is calculated to avoid operating in the integer field. Also, the use of implicit loops greatly simplifies the reading of the code.

6.4 Integrals and derivatives of functions

Let's build a simple approximation to the derivative of a function $f(x)$ by means of its definition:

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (6.2)$$

In calculus, the existence of this limit (derivable function) implies that both lateral limits exist, are finite and equal (at least when the function is defined in both sides of the point x):

$$f'(x) = \lim_{h \rightarrow 0^-} \frac{f(x+h) - f(x)}{h} = \lim_{h \rightarrow 0^+} \frac{f(x+h) - f(x)}{h} \quad (6.3)$$

To obtain an approximation to this value $f'(x)$ in a point x with the computer, finite differences can be used. Instead of reproducing the limit in the point, a fixed value of $h \neq 0$ is used. Notice that both $h > 0$ or $h < 0$ are possible so the approximation can be found like a forward difference or a backward difference. Actually, also central differences can be used if the expression is slightly modified. What must be considered is that now each expression leads to a different approximation while in calculus the existence of the derivative implies that all values are equal.

For this case let's consider $h > 0$ with h a small value:

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h} \quad (6.4)$$

The question that arises; is it possible to bound the error of the approximation?

$$E = f'(x) - \frac{f(x+h) - f(x)}{h} \quad (6.5)$$

If the function $f(x)$ is sufficiently smooth near x (for a forward-difference twice differentiable is needed), then a Taylor expansion can be used so the order of the scheme used is obtained. Intuitively, it is clear that by lowering the value of h a better approximation of the derivative is obtained. Finding the order of the scheme means precisely finding how fast the error tends to zero when $h \rightarrow 0$. In addition, round-off error appears in this approximation depending on the value of h too.

Let's bound the truncation error for a forward difference. Starting with the Taylor series of $f(x)$ at the point x particularized in $x + h$:

$$f(x + h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \mathcal{O}(h^3)$$

Notice that (h^3) means that the following terms are lower than a constant times h^3 . Re-arranging the expression:

$$\frac{f(x + h) - f(x)}{h} = f'(x) + \frac{h}{2}f''(x) + \mathcal{O}(h^2) = f'(x) + \frac{h}{2}f''(x) + \mathcal{O}(h^2)$$

$$\left| \frac{f(x + h) - f(x)}{h} - f'(x) \right| \leq Ch$$

The error committed by replacing the derivative $f'(x)$ by the finite difference approximation is of order h and the same result is obtained if backward-difference is used. Of course this is not the only way to approximate the value of a derivative. First of all, central differences could be used and, if the function is three times differentiable the truncation error of that approximation can be demonstrated that is of order h^2 . Secondly, higher order finite differences exist so the value of a first derivative can be approximate with order h^4 , h^6 , etc. considering more points of the domain, while the example shown here comes from the derivative definition or the Taylor series expansion similarly, higher order approximations can be inferred from the interpolation theory. Thirdly, other strategies that also comes from the interpolation theory can be used so spectral convergence can be obtained (see section 4.8).

A different source of discrepancy between the real value and the approximation appears; round-off error, also dependent on the value of h taken. Notice that as h gets smaller, the values of $f(x + h)$ and $f(x)$ become more similar. This issue, with infinite precision arithmetic would not pose any problem, however, performing the subtraction with the computer's finite precision involves the problem of numerical cancellation. This topic is broaden in the chapter 3.9 since the basis of real numbers representation are explained. For now, just keep in mind the following:

- Every real numbers in the computer is stored with a rounded value that fits in the finite precision used by the computer. Consider for example the numbers 1.2345651 and 1.2345649, in a finite precision machine they could be stored only with 6 significant digits and they would become: 1.23457 and 1.23456,

they have been rounded. Of course, both values $f(x+h)$ and $f(x)$ in our derivative computation suffer from this issue.

- When two near values are subtracted the significant digits are reduced, i.e. from 6 significant digits in the following expression we end up with just 1:

$$1.23457 - 1.23456 = 0.00001 = 1 \cdot 10^{-5}$$

- If both values are approximated to the sixth digit and the result of the subtraction is a value in the order of magnitude imposed by that digit, then it is dominated by the round off made to the numbers! In this example the infinite precision operation would result in $1.2345651 - 1.2345649 = 2 \cdot 10^{-7}$ quite smaller than the result obtained by the finite precision machine.

Our expression for the derivative $\frac{f(x+h)-f(x)}{h}$ will have an error in the numerator divided by h so, with no deeper explanations let's say now that this rounding error can be bounded by the following expression. Notice how it depends inversely on h so it is a barrier for the reduction of h as a way to increase the accuracy of the derivative. The decision and meaning of writing 2ϵ for the numerator error is deeply treated in the chapter .

$$E_{round} = \frac{2\epsilon}{h}$$

Since we could be interested in making h as small as possible to obtain better approximations to the derivative, it is not difficult to find that round-off barrier in our computations. Both sources of error can be dominant and it is essential to understand the origin of both and how to relate the precision desired for a simulation (needs of the project) and the way to approximate derivatives with the computer.

The following codes recycles the procedure interface used along this chapter of `f_R_R(x)` for the function `f` and defines the `Derivative` function with inputs the function to derive and the value where obtaining the approximation. Notice that this new function itself also responds to the interface:

$$\begin{aligned} \text{f_R_R}: \mathbb{R} &\rightarrow \mathbb{R} \\ x &\rightarrow \text{f_R_R}(x) \end{aligned}$$

```
function Derivative( f, x ) result(df)
  procedure (f_R_R) :: f
  real, intent(in) :: x
  real :: df
```

```

real :: h = 1e-5

    df = ( f(x+h) - f(x) ) / h  ! lim h->0 ( f(x+h) - f(x) )/h
end function

```

Listing 6.3: Calculus.f90

```

interface
  function f_R_R(x) result(f)
    real, intent(in) :: x
    real :: f
  end function
end interface

```

Listing 6.4: Calculus.f90

Now let's compute the definite integral of a function $f(x)$ in the closed interval $[a, b]$ and let's do it in a functional way as it is common during this book. In this case we are going to obtain an approximation to the integral value through a Riemann sum, which means that the area of the curve under the function in the interval is obtained as a sum of small rectangles along the interval and carrying that process to the limit.

$$\int_a^b f(x)dx = \lim_{m \rightarrow \infty} \sum_{i=0}^{m-1} f(x_i)\Delta x$$

Each value x_i is taken from a partition of $[a, b]$:

$$x_i = a + i\Delta x \quad \text{with} \quad i = 0, \dots, m-1$$

and with:

$$\Delta x = \frac{|b - a|}{m}$$

However, we need an approximation so instead of evaluating that limit, we choose a value m big enough to accomplish with the accuracy required.

$$\int_a^b f(x)dx \simeq \sum_{i=0}^{m-1} f(x_i)\Delta x$$

In this example, the leftmost value of each subinterval is taken to approximate the rectangle area, however, the rightmost value or another point contained in the interval could be used.

```
function Integral( f, a, b ) result(I)
  procedure (f_R_R) :: f
  real, intent(in) :: a, b
  real :: I

  integer :: k, M
  real :: dx

  dx = 1e-3
  M = abs(b-a)/dx
  dx = (b-a)/M

  I = dx * sum ( [ ( f(a+dx*k), k=0, M-1 ) ] )
end function
```

Listing 6.5: Calculus.f90

Some notes should be taken from this example: firstly, notice that the initial value given to `dx` is not the actual value used, it is just an approximation to the expected value (0.001). The operation `abs(b-a)/dx` is used to calculate the approximate number of subintervals that fit in our interval $[a, b]$ and then the integer part is retained. Then, the exact value of `dx` is calculated for an integer number of subintervals. Secondly, notice that the sum must not stop in the last value of the interval (b) but in the previous one since the area of that rectangle is computed by $dx \cdot f(a + dx(m - 1))$. As usual, the whole addition can be compacted in the sum of the components of a vector defined by an implicit loop.

6.5 Examples of operations with functions

The following subroutine gives some examples of the functions seen during this chapter. `DISLIN` is used for all the graphs as mentioned before. Firstly, the plot intervals must be initialize so the `plot_ini` subroutine is used and the plotting boundaries are declared, all the graphs will be calculated and represented in $x \in [-2\pi, 2\pi]$, $y \in [-2.5, 2.5]$. Then, four functions are represented; a sine, the piecewise function defined in 6.1, its derivative and its integral from -1 to x .

To graph those functions with `plot(f)`, the single input argument needed is the function itself. Also, the already explained `procedure (f_R_R) :: f` is used for `f`. Hence, the functions `Derivative_f(x)` and `Integral_f(x)` are needed to build the following mathematical expressions:

$$\text{Derivative_f}(x) = \frac{df(x)}{dx}$$

and:

$$\text{Integral_f}(x) = \int_{-1}^x f(x)dx$$

Remember that $f(x) = \text{Piecewise_f}(x)$. Notice that integrating the function with these limits implies finding one of the primitive functions of $f(x)$, that one where the not defined constant has a specific value decided by the lower integration limit $x = -1$. The function treated in this example is positive (or zero) in the whole domain of definition so its integral as well. However, since it is plotted starting in $x = -2\pi$ and the function is integrated from $x = -1$, we expect the result to be negative until it reaches zero in $x = -1$ according to this property of definite integrals:

$$\int_{-1}^x f(x)dx = - \int_x^{-1} f(x)dx$$

```
subroutine Integral_and_derivative_examples()
```

```
    call plot_ini( -2*PI, 2*PI, +2.5, -2.5 )
```

```
    call plot( sine )                ! plot sine function
```

```
    call plot_show()
```

```
call plot( Piecewise_f )    ! plot piecewise function
call plot_show()

call plot( Derivative_f )  ! plot derivative of piecewise function
call plot_show()

call plot( Integral_f )    ! plot integral of piecewise function
call disfin

contains

function Derivative_f(x) result(Df)
  real, intent(in) :: x
  real :: Df

  Df = Derivative( Piecewise_f, x )
end function

function Integral_f(x) result(I_f)
  real, intent(in) :: x
  real :: I_f

  I_f = Integral( Piecewise_f, a = -1., b = x )
end function

end subroutine
```

Listing 6.6: Integrals_derivatives.f90

Chapter 7

Series expansion

7.1 Introduction

One of the main focus of numerical methods is to approximate functions by means of different expansions:

1. Polynomial expansion: Taylor, Lagrange
2. Trigonometric series: sine and cosine expansions.
3. Expansion by means of known basis: Chebyshev, Legendre,...

The same function can be expanded or approximated with different basis. It goes without saying that the computer only allows to sum a finite number N of terms.

Hence, the best election is related to the rate of convergence of the expansion. In other words, given a tolerance error between the approximation and the exact function, the best expansion allows to obtain the approximation with a minimum number of terms N .

7.2 Expansions of functions

$$f(x) = \sum_{k=0}^N a_k x^k, \quad a_k = \frac{f^{(k)}(0)}{k!},$$

```

real function TaylorE( df, x0, x, eps)
  procedure (f_RxN_R) :: df
  real, intent(in) :: x0, x, eps

  TaylorE = Sigma( a, 0, eps)

contains

real function a(k)
  integer, intent(in) :: k

  a = df(x0, k) * ( x - x0 )**k / factorial(k)

end function

```

Listing 7.1: Taylor_expansions.f90

```

interface

  function f_RxN_R(x, k) result(f)
    real, intent(in) :: x
    integer, intent(in) :: k
    real :: f
  end function

end interface

```

Listing 7.2: Taylor_expansions.f90

```

real function TaylorN( df, x0, x, N)
  procedure (f_RxN_R) :: df
  real, intent(in) :: x0, x
  integer, intent(in) :: N

  TaylorN = Sigma_N( a, 0, N)

contains

real function a(k)
  integer, intent(in) :: k

  a = df(x0, k) * ( x - x0 )**k / factorial(k)

end function

```

Listing 7.3: Taylor_expansions.f90


```

interface Taylor
  module procedure TaylorN, TaylorE
end interface

```

Listing 7.4: Taylor_expansions.f90

7.3 Parseval identity

$$f(x) = \sum_{k=-\infty}^{\infty} \hat{c}_k e^{ikx} \quad (7.1)$$

$$f(x) = \sum_{k=0}^{\infty} \hat{a}_k \cos kx + \sum_{k=1}^{\infty} \hat{b}_k \sin kx \quad (7.2)$$

$$\hat{c}_k = \frac{1}{2} (\hat{a}_k - i \hat{b}_k), \quad k = 1, \dots, N/2. \quad (7.3)$$

$$\hat{c}_{-k} = \bar{\hat{c}}_k, \quad k = 1, \dots, N/2. \quad (7.4)$$

$$\int_{-\pi}^{\pi} f(x)^2 dx = 2\pi \left(|\hat{c}_0|^2 + 2 \sum_{k=1}^{\infty} |\hat{c}_k|^2 \right) \quad (7.5)$$

Let $f(x) = x \forall x \in [-\pi, +\pi]$ and extend $f(x)$ periodically to the right and to the left. Once $f(x)$ is defined in this way, it can be approximated by a sine expansion:

$$f(x) = \sum_{k=1}^{N/2} \hat{b}_k \sin kx, \quad (7.6)$$

where the coefficients \hat{b}_k are given by:

$$\hat{b}_k = \frac{(-1)^{k+1}}{\pi k}, \quad k = 1, \dots, N/2. \quad (7.7)$$

Using Parseval identify with $f(x)$, the following summation is obtained:

$$\sum_{k=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6} \quad (7.8)$$

Chapter 8

Read an external file

```
subroutine Test_load_matrix()

  real, allocatable :: A(:, :)
  integer :: i, j

  A = load_matrix('./sources/Foundations/data.csv')

  ! print row ith
  do i=1, size(A, dim=1)
    write(*, '(10f8.3)') ( A(i,j), j=1, size(A, dim=2) )
  end do

  A = load_matrix('./sources/Foundations/data2.csv')

  do i=1, size(A, dim=1)
    write(*, '(10f8.3)') ( A(i,j), j=1, size(A, dim=2) )
  end do

end subroutine
```

Listing 8.1: read_files.f90

```

function load_matrix( filename ) result(A)
  character(len=*) , intent(in) :: filename ! string of characters
  real, allocatable :: A(:, :)

  ! Local variables
  integer :: i_file = 8           ! file unit to read
  character(len=400) :: header ! text header row
  integer :: M, N                 ! rows(M) and columns(N)
  integer :: i, j

  ! it associates 8 to filename
  open( unit = i_file, file = filename)
  read( unit = i_file, fmt = '(A)') header

  N = columns( string = header, s = ", " )
  M = 0

  ! if end of file then, go to 10
  do
    read(unit = i_file, fmt = *, end = 10)
    M = M + 1
  end do
10 write(*,*) " N =", N, "M = ", M

  allocate( A(M, N) )

  rewind(unit = i_file)
  read(i_file, '(A)') header           ! header
  do i=1, M
    read( i_file, *) ( A(i, j), j=1, N)
  end do

  close(i_file) ! to allow others to read filename

end function

```

Listing 8.2: read_files.f90

```
integer function columns( string, s) result(M)
  character(len=*), intent(in) :: string
  character, intent(in) :: s           ! delimiter

  integer :: i, L ! length

  L = len_trim(string) ! length without blanks

  M = 1
  do i=1, L
    if (string(i:i) == s ) then
      M = M + 1
    end if
  end do

end function
```

Listing 8.3: read_files.f90

Part II

Computer operations with integers and reals

Chapter 1

Overview

Being the base of any numerical calculation that a computer scientist want to perform, both, integers and reals, must be controlled. This part of the book covers the particularities of these two data types giving special attention to the common errors that appear when programming maths in the computer.

How the numbers are treated by computers is similar for most machines in the world. Although Fortran is used for the examples, any programming language will carry with the same issues in a similar way. Here you will find some useful notions to acquire a better understand of the behaviour of integers in a computer, the arithmetic behind different data types or the phenomena associated to the approximation of numbers by a finite precision.

Make use during the reading of the chapter of the attached programs, there you will find the same examples explained here and the possibility to change and write your own programs using the existing codes. The following is the menu implemented in the program and also serves as an overview of the topics covered in the next pages.

```
write(*,*) " Enter option ? "  
write(*,*) " 0. Exit "  
write(*,*) " 1. Integer overflow "  
write(*,*) " 2. Understanding integer overflows "  
write(*,*) " 3. Overflow by incorrect assignment "  
write(*,*) " 4. Two's complement converter "  
write(*,*) " 5. IEEE reals representation "  
write(*,*) " 6. Round off errors in operations "  
write(*,*) " 7. Loss of precision "  
write(*,*) " 8. Catastrophic cancellation "  
write(*,*) " 9. Summation errors "  
write(*,*) " 10. IEEE exceptions "
```

Listing 1.1: Foundations.f90

Chapter 2

Integers representation

Overview

It is well known that the programs are written with numbers in decimal form. However, the computer, in order to work with these constants and variables, is going to convert numbers to binary form. Using two voltage levels (representing 0's and 1's) as unique states the computer can perform calculations that, in order to show to the user the results obtained, will be translated again to decimal form.

The computer translates decimal values to binary similarly to the way humans typically do (with some exceptions to be noticed). For positive integers the way to translate a value uses the principles of positional numeral systems, the value of each digit is totally determined by the digit itself, the position in the number and the base of the numeral system (10 and 2 for decimal and binary). In the case of negative integer values, while a person could write a sign (-) before the number, the machine uses only 0's and 1's and how this is done is treated in the chapter.

On one hand, not all the numbers that can be written in a program (actually, much less numbers than the infinite possibilities) are exactly represented in the binary translation that the computer performs. It is essential for a programmer to understand this issue. On the other hand, integers do have an exact representation in binary form with an integer variable but its maximum and minimum value depends on the memory size of this variable. The main problematic that motivates the study of the integer representation in computers is related to overflow, values out of the representable range.

The following topics are covered in this chapter:

1. Integer overflow example.
2. Two's complement integer representation.
3. Understanding integer overflows.
4. Overflow of constants.
5. Overflow by incorrect assignment.
6. Declaring kind
7. Two's complement converter

Throughout this chapter the reader will be able to understand and justify the corrupted values stored in a variable when an out-of-range integer is assigned by mistake. When this error occurs during compilation time, in many cases, the compiler does not abort the compilation, it only prints a warning. In the worst situation, the error appears during execution, the program is not working properly and this source of error could be a nightmare to trace.

2.1 Integer overflow example

The range of integer variables is limited by its memory size representation. When programming and by mistake is possible to assign a value out of the range of some integer variable. It is very important to understand how the programming language treats this exception to understand what is going on in our execution. In the following example, the integer variable `i` is stored in 1-byte of memory whereas `k` is stored in 2-bytes.

```
subroutine Integer_overflow

  integer (kind=1) :: i
  integer (kind=2) :: k

  write(*,*) " Integer overflow "
  do k=0, 257
    i = k
    write(*,*) "k = ", k, " i= ", i
  end do

end subroutine
```

Listing 2.1: Example of Integer overflow. `Integer_representation.f90`

A single loop prints on screen the successive values of `k` and `i`. Once the value `+127` of `i` is reached, negative numbers of `i` appear even when we are incrementing step by step the variable `k`. The successive values of `i` after `+127` are `-128, -127, ...`

To understand this fact, we should consider that integer numbers of 1-byte size are treated as numbers modulo 2^8 which means that if integer numbers were unsigned, their range would be from 0 to 255. If this unsigned integer variable were assigned with a number `k` greater than 255, the resulting value would be `k mod 255` (remainder of `k` divided by 255). Additionally, to hold signed integer variables (positive and negative values), a two's complement notation is generally used that restrict the integer values from -128 to +127 in case of 1-byte size variables.

Another classical example that disconcerts to the novel programmer is the overflow with classical mathematical operations such as the factorial of an integer value:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \dots \cdot 3 \cdot 2 \cdot 1$$

In the following code, the factorial of 20 is obtained by multiplying the integer variable `f` sequentially from 1 to 20. The partial results are shown on screen. After the variable `k` has reached 13, the variable `f` shows wrong results.

```
subroutine Factorial_overflow

  integer :: k, f

  write(*,*) " Factorial overflow "
  f = 1
  do k=1, 20
    f = f * k
    write(*,*) "k = ", k, " f= ", f
  end do

end subroutine
```

Listing 2.2: Factorial overflow. Integer_representation.f90

Once the result has reached the maximum value the integer variable `f` can hold, even negative numbers appears being the result of the modular arithmetic with two's complement. It is important to notice, that in this case, the integer variable `f` is stored in 4 -bytes which is the default storage of an integer value.

When an integer overflow occurs, the execution is not aborted and the programmer is not alerted of the malfunction. This is the main problems of integer overflow occurrences. Generally, and integer overflow is associated to the following situations:

1. Counters to measure time or repetitive processes. If these counters are used in a perpetual control system, they are eventually overflowed.
2. Mathematical operations in the set of integers \mathbb{Z} such as the factorial of a number.
3. Careless assignments between reals and integers.

When program codes are used as controlling units in embedding systems, the overflow could origin terrible accidents such as the crash of the 1996 maiden flight of the Ariane 5 rocket.

On May 2015, the European Aviation Safety Agency discovered that the embedded software of the Boeing 787 suffered after 248 days an integer overflow of a counter of a signed variable of 32-bits. This overflow required to reset periodically the system to avoid loss of electrical power and ram air turbine deployment.

The two's complement integer representation is explained in the next section.

2.2 Two's complement integer representation

An integer variable declared as 4-bytes (32 bits) size has 2^{32} possible different values. Some languages allow to work with unsigned integers. In that case, all bits are reserved for positive values and its range goes from 0 to $2^N - 1$. When specifying signed integers, one bit has to be reserved for the sign and then the range is divided by two. Hence, an integer variable stored in 4-bytes has the range $[-2\ 147\ 483\ 648, 2\ 147\ 483\ 647]$. The Table 2.1 summarizes the main parameters for signed integers.

Name	Minimum (-2^{N-1})	Maximum ($2^{N-1} - 1$)
Kind = 1	-128	127
Kind = 2	-32768	32767
Kind = 4	-2147483648	2147483647
Kind = 8	-9223372036854775808	9223372036854775807

Table 2.1: Main properties of the different signed integer kinds used (two's complement used for negative values).

In general, the range of an integer variable stored in N bits is:

$$[-2^{N-1}, 2^{N-1} - 1]. \quad (2.1)$$

The Table 2.2 represents an integer storage with N-bits where b_i denotes the possible values 0 or 1.

b_{N-1}	b_{N-2}	b_{N-3}	b_2	b_1	b_0
-----------	-----------	-----------	-----	-----	-------	-------	-------

Table 2.2: Binary representation of an integer number stored in N bits.

The general expression to convert this binary number to decimal when it is expressed in two's complement is given by:

$$x = -b_{N-1} 2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i. \quad (2.2)$$

As an example, let's consider an integer value equal to 425_{10} stored in 2-bytes of memory. The binary form comprises 16 different bits to hold the integer number. As a signed positive integer it is represented with common binary numeral system:

$$\begin{aligned} N = \ 0000000110101001_2 &= 1 \cdot 2^8 + 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + \\ &+ 0 \cdot 2^1 + 1 \cdot 2^0 = 2^8 + 2^7 + 2^5 + 2^3 + 2 = 425_{10} \end{aligned}$$

where the leading zeroes are not displayed when printing that number on your screen. The same number stored in a variable of 4-bytes (32bits) of memory size has the same binary representation but with more leading zeroes. Notice that a signed integer variable of 1-byte size can not store this number since its range is $[-128, 127]$.

The following two questions arise:

1. How are negative integer numbers stored in the computer?
2. How is overflow treated with integer numbers?

To answer these two questions, we should understand how signed integer numbers are stored in the computer. There are two main ways to store signed integer variables in memory:

1. One's complement integer representation.
2. Two's complement integer representation.

In the following table, a signed integer variable of 3-bits of memory size is represented in binary form, unsigned integer, two's complement and one's complement.

Binary	unsigned	two's comp.	one's comp.
000	0	0	0
001	1	1	1
010	2	2	2
011	3	3	3
100	4	-4	-3
101	5	-3	-2
110	6	-2	-1
111	7	-1	-0

The two's complement notation is, generally, used in computers. If the leftmost bit is 1, the integer variable holds a negative number. From the above table, a procedure is deduced to obtain the binary form of a negative number. For example, the binary form of -2 is obtained from the binary form of 2 which is 010. Then, these bits are inverted to give 101. An finally, 001 is added to the resulting number to give: 110 which is -2 in the two's complement.

Important Notice

To obtain the binary form in two's complement of a negative number, carry out the following steps:

1. Obtain the binary form of the integer number without sign.
2. Invert all bits by using the bitwise NOT operation.
3. Add 1 to the resulting value.

The signed negative number -425_{10} , using two's complement, is translated in the machine like:

$$N = 1111111001010111_2 = -425_{10} \quad (2.3)$$

Notice that this technique to encode signed numbers defines the 'two's complement' of a specific number with N bits as its complement with respect to 2^N which means that adding a number to its complement results in 2^N . From the different ways to represent signed numbers, this method performs addition, subtraction, and multiplication in the same way for signed and unsigned binary numbers and it lacks from the negative zero.

Once the treatment of negative numbers is understood, the next step is to understand how the computer treats an overflow with integers. The most common result of an overflow is that the least significant representable digits of the result are stored. Imagine that integer value 5 is stored in a 4-bits x -variable:

$$x = 0101.$$

If the content of this variable is assigned to a y -variable of 3-bits (which can only store the range $[-4,3]$), only the least significant bits are stored and the resulting variable will store the number

$$y = 101,$$

which represents the value -3 in a two's complement representation with 3 bits. The original value of x has value 5. Since the overflow of an integer variable never shows as an error, it is very important to avoid overflows of integer variables not to have mysterious programming problems very difficult to be identified.

Important Notice

Overflow of integer variables are not usually identified as explicit errors and they can originate strange behaviors very difficult to discover in a programming code.

2.3 Understanding integer overflows

Now, the examples at the beginning of this chapter are easily interpreted. In the first case, the values $+128, +129, \dots$ stored in `k` (2-bytes memory) size are assigned to the variable `i` with 1-byte memory size. Then, the binary forms of $+128_{2-bytes}$ or $+129_{2-bytes}$:

$$+128_{2-bytes} = 00000000 \ 10000000, +129_{2-bytes} = 00000000 \ 10000001$$

are stored keeping the 8 least significant bits like:

$$10000000, 10000001$$

which are the binary two's complement form of -128_{1-byte} and -127_{1-byte} respectively.

In the second example the variable `f` does not overflow in one or two units but in a much higher quantity. The real result of $13!$ is 6227020800 whose binary representation with enough memory size to store it (40 bits for example) is:

$$6227020800_{40-bits} = 00000001 \ 01110011 \ 00101000 \ 11001100 \ 00000000$$

Once this variable is stored with the 32 least significant bits the result is:

$$01110011 \ 00101000 \ 11001100 \ 00000000$$

which is the binary two's complement representation of $+1932053504_{4-bytes}$, the result printed by the computer. From that value, all the following results of the factorial are wrong because of the overflow of the 4-bytes variable.

In the following example, an integer variable of 1-byte size is overflowed with the value 130 and with -130. The resulting values are compared with an integer variable of 2-bytes size. The internal representation in binary form according to two's complement is also shown. The following subroutine determines the internal binary form of an integer of any kind. Depending on the `kind_type` of the integer variable, the maximum and minimum range is calculated by means of the

subroutine `integer_parameters`. Later, the integer variable `xr` is written in binary form by means of a `write` sentence. And finally, the integer variable `xr` is reconstructed by means of equation (2.2) to show on screen and to check the binary form representation.

```

subroutine Integer_representation_twos_complement( kind_type, x )
  character (len=*), intent(in) :: kind_type
  integer (kind=8), intent(in) :: x

  integer, parameter :: N = 64
  character (len=N) :: bits          ! binary form char
  integer :: b(0:N-1)                ! binary form vector
  integer (kind=8) :: m, xr, xb, xmax, xmin ! integer parameters
  integer :: i

  call integer_parameters(kind_type, x, m, xr, xb, xmax, xmin )

  write(bits, '(B64)' ) xr
  b = 0
  do i=1, N;
    if (bits(i:i) == ' ') bits(i:i) = '0';
    if (bits(i:i) == '1') b(N-i) = 1;
  end do

  xr = - b(N-1) * 2**(N-1)
  do i=0, N-2
    xr = xr + b(i) * 2**i
  end do

  open( 6, FILE='CON', STATUS='UNKNOWN', RECL=200 )
  write(6, '(a)' ) "-----"
  write(6, '(a)') " Internal representation of integers " // kind_type
  write(6, '(a35, i20)') "Integer kind=8          x = ", x
  write(6, '(a35, i20)') "Same integer "//kind_type// "          x = ", xb
  write(6, '(a27, 2i20)') "Integer range "//kind_type// " = ", xmin, xmax
  write(6, '(a35, a)') "Internal representation of x = ", bits(N-m+1:N)
  write(6, '(a35, i20)') "-b_{N-1}2 ** (N-1)+sum b_i 2**i = ", xr
  write(6, '(a)' ) "-----"

end subroutine

```

Listing 2.3: Binary form representation. `Integer_representation.f90`

To understand the binary internal representation of an integer overflow, the following lines gives the results:

```

call Integer_representation_twos_complement("kind=1", 130)
call Integer_representation_twos_complement("kind=2", 130)
call Integer_representation_twos_complement("kind=1", -130)
call Integer_representation_twos_complement("kind=2", -130)

```

```

Internal representation of x=130 with x integer kind=1
  Integer kind=8      x =      130
  Same integer kind=1  x =     -126
  Range integer kind=1  =     -128      127
  Internal representation of x = 10000010
  Reconstruction sum b_i 2**k =     -126

```

```

Internal representation of x=130 with x integer kind=2
  Integer kind=8      x =      130
  Same integer kind=2  x =      130
  Range integer kind=2  =    -32768      32767
  Internal representation of x = 000000000100000010
  Reconstruction sum b_i 2**k =      130

```

```

Internal representation of x=-130 with x integer kind=1
  Integer kind=8      x =     -130
  Same integer kind=1  x =      126
  Range integer kind=1  =     -128      127
  Internal representation of x = 01111110
  Reconstruction sum b_i 2**k =      126

```

```

Internal representation of x=-130 with x integer kind=2
  Integer kind=8      x =     -130
  Same integer kind=2  x =     -130
  Range integer kind=2  =    -3276      32767
  Internal representation of x = 1111111101111110
  Reconstruction sum b_i 2**k =     -130

```

It's shown that the value 130 overflows an integer variable of kind=1. When this happens, no alert is shown by the compiler and non desirable results could destroy the performance of our programming code.

2.4 Overflow of constants

The issues and limits treated above also appear when, instead of integer variables, the program works with integer constants. The assignment of a constant which is out range of a `kind=8` integer variable will give rise to a compilation error.

In the following example an integer variable of size 8 bytes is assigned with a huge number:

```
integer (kind = 8) :: P

!Example variable assignation out-of-range 8 bytes
P = 9223372036854775808
```

When trying to compile the above program, the compiler returns the following message:

```
Compilation Aborted (code 1)
error #6901: The decimal constant was too large when
      converting to an integer, and overflow occurred.
      [9223372036854775808]
```

The compiler tries to store the number 9223372036854775808, which is equal to `huge(P)+1`, in an internal register of 8-bytes with two's complement representation. Since the constant is greater than the maximum allowed value of the maximum memory size available in the computer the compiler gives the above error.

2.5 Overflow by incorrect assignment

If valid huge constants are assigned to lower size integer variables, only compiler warnings could alert of possible malfunctions.

In the following example, an integer variable `Q` of 4-bytes is considered. The value 2147483648, which is equal to `huge(Q) + 1`, is assigned to `Q`.

```
subroutine Assignment_overflow

  integer (kind=4) :: Q

  !If the assignment of the value is out-of-range,
  ! no error is detected by the compiler
  ! only a warning could alert of possible malfunctions

  Q = 2147483648
  write(*,*) "Intention: assign Q <- 2147483648"
  write(*,*) "Result:  Q =", Q, "huge(Q) = ", huge(Q)

end subroutine
```

Listing 2.4: Assignment overflow. Integer_representation.f90

When compiling the above program, no errors are obtained but the following warning appears:

```
warning #6384: The INTEGER(KIND=4) value is out-of-range.
```

Hence, the program could be executed without taking care of warning messages giving the following result:

```
Intention: assign Q <- 2147483648
Result:  Q = -2147483648 huge(Q) =  2147483647
```

If warnings are not taken into account, non desirable results could be obtained. In this case, variable `Q` stores the value `-2147483648` and the programmer by an improper assignment tried to store the value 2147483648 which is greater the maximum value that `Q` can store.

2.6 Declaring kind

A good practice is to write codes that could be used with different integer sizes: `kind=1, 2, 4, 8`. This can be done by not specifying the kind type of any integer variable. In this case, the compiler assumes default integer kind for all integer variables. Generally, this default can be easily changed with a proper compilation option.

The opposite approach is to declare specifically the kind type of any integer variable by means of sentences such as:

```
integer (kind=1) :: i1
integer (kind=2) :: i2
integer (kind=4) :: i4
integer (kind=8) :: i8
```

When declaring constants, the situation is different. A specific kind type is declared by underscoring the constant with its kind type:

```
integer (kind=8) :: i

i = 123_1
i = 123_2
i = 123_4
i = 123_8
```

In the above example, the constant `123` is stored in a register of 1, 2, 4, or 8 bytes. Later, it is assigned to an integer variable of 8 bytes. When no kind type specifier is selected for integer constants (`i = 123`), then the compiler reserves an internal appropriate register of variable size depending on the value of the constant.

Important Notice

- **Constants.** The compiler reserves memory size depending on the magnitude of the constant.
- **Variables.** The memory size of any variable is specified by its kind type. If no kind type is specifically declared, the default integer kind is considered by the compiler.

2.7 Two's complement converter

A binary-to-decimal and decimal-to-binary converters are included in the codes that accompany this book. A third function is also included, it checks if an integer value is not properly stored with an specific number of bits because of an overflow error. The converters will return the two's complement conversion for the introduced value, whether is an integer value or a string characters of 0's and 1's. The specifications of each function are presented now.

2.7.1 TwosCompl_converter_to_decimal function

Returns the decimal integer value for the string of binary characters introduced according to the two's complement encoding.

Notice that the two's complement encoding is always referred to an specific number of bits which means that it make sense in a pre-defined memory size. Due to that, the conversion is performed according to the total length of bits introduced in the string, if your binary expression has leading zeros do not forget to include them. In addition, blank spaces are also treated as zeros so the string ' 01101' is the same as '001101'.

If more than 64 bits or wrong characters are introduced the result will be automatically 0. The table 2.3 shows the specifications of the input arguments of the converter.

`TwosCompl_converter_to_decimal(bits)`

Name	Type	Limits
bits (input)	character	Up to 64 characters with 0's, 1's or blank spaces (treated as 0's)
Result	integer(kind=8)	[−9223372036854775808, 9223372036854775807]

Table 2.3: Specifications of the `TwosCompl_converter_to_decimal` function.

Example:

```
write(*,*) "Two's complement conversion of binary: "
write(*,*) "010101010110111001110 = "
write(*,*) TwosCompl_converter_to_decimal( '010101010110111001110' )
```

```
Two's complement conversion of binary:
010101010110111001110 =
                        699854
```


2.7.2 TwosCompl_converter_to_binary function

Returns the minimum length binary string that can store the decimal integer introduced according to the two's complement encoding.

Notice that the two's complement encoding is always referred to an specific number of bits which means that it make sense in a pre-defined memory size. This result is printed with the minimum bits that can store the number.

Out-of-range `kind=8` integer values will not be compiled (see table 2.4). In addition, take into account that the algorithm used for this converter is the one described in the section 2.2. This algorithm will not return a proper value for the case $x = -\text{huge}(P) - 1$ where P is a `kind=8` variable. While that number can be perfectly stored in the computer, its absolute value can not:

$$|x| = |-\text{huge}(P) - 1| = \text{huge}(P) + 1$$

`TwosCompl_converter_to_binary(x)`

Name	Type	Limits
x (input)	integer (all kinds)	$[-9223372036854775807, 9223372036854775807]$
Result	character	Up to 64 binary characters

Table 2.4: Specifications of the `TwosCompl_converter_to_binary` function.

Example 1:

```
write(*,*) "Two's complement conversion of integer: ", -89544563
write(*,*) TwosCompl_converter_to_binary( -89544563 )
```

```
Two's complement conversion of integer:      -89544563
1010101010011010100010001101
```

Example 2: This example shows that the number 1 must be represented with at least two bits. The following section explains this result.

```
write(*,*) "Two's complement conversion of integer: ", 1
write(*,*) TwosCompl_converter_to_binary( 1 )
```

```
Two's complement conversion of integer:      1
01
```

2.7.3 Check_overflowed_int function

For an integer introduced it returns the actual decimal integer stored in the computer in a specific memory size according to the two's complement encoding. If the number is stored in more (or equal) binary digits than the minimum length needed, the result is the same as the input.

Notice that an out-of-range `kind=8` integer value will not be compiled (see table 2.5).

```
Check_overflowed_int( int, n_bits )
```

Name	Type	Limits
<code>int</code> (input)	<code>integer</code> (all kinds)	<code>[-9223372036854775808, 9223372036854775807]</code>
<code>n_bits</code> (input)	<code>integer</code> (default kind)	<code>[1, 64]</code>
Result	<code>integer(kind=8)</code>	<code>[-9223372036854775808, 9223372036854775807]</code>

Table 2.5: Specifications of the `Check_overflowed_int` function.

A non-intuitive result is obtained from the experiment of representing the decimal integer 1 with just one bit. In a pure binary conversion (unsigned integer for example) it is clear that the number 1 is represented through the binary digit 1. However, it can be checked that the two's complement conversion of the number 1 can only be performed with 2 or more binary digits (01), being the result of the following line -1.

```
Check_overflowed_int( 1, 1 )
```

Example:

```
write(*,*) "Check overflow integer of 13! = ", 6227020800
write(*,*) "with 4-bytes memory size = ", 32, "bits"
write(*,*) "The actual value stored is: "
write(*,*) Check_overflowed_int( 6227020800, 32 )
```

```
Check overflow integer of 13! =          6227020800
with 4-bytes memory size =          32 bits
The actual value stored is:
1932053504
```

Chapter 3

Reals representation and operations

Overview

Real numbers can be thought of as points with a infinite decimal representation on an infinitely long line called the real line. However, since the internal representation of a real number in a computer is stored on a finite amount of memory, only some real numbers can be represented exactly. In other words, working with a finite precision machine, real numbers are approximate.

This idea is extended to real operations and allows to understand that most calculations are approximate and the resulting errors are called "round-off" errors. The origin and the consequences of these errors will be discussed in this chapter.

The following topics are covered in this chapter:

1. Example of round-off errors.
2. Fixed-point and Floating-point representation.
3. Representation in IEEE 754.
4. Distance between floating point real numbers.
5. Reconstruction from its internal binary representation.
6. Writing floating point expressions.
7. Condition number and stability.
8. Catastrophic cancellation.
9. Summation example.
10. IEEE exception examples.

3.1 Example of round-off errors

To introduce unexpected behavior when working with real variables and round-off errors, two examples are presented. In the first example, a real variable S stored in 4 bytes of memory is used to sum $N=100000$ times the value $dX = 0.3$. The expected result is $S = 0.3 \times 100000 = 30000$. However, the execution gives $S = 3027.90$. Later, in the same subroutine, the magnitude $dX = 0.3$ is subtracted N times. Again, the expected result is $S = 0$ but the computer result is $S = 26.1582$.

```
subroutine errors_in_operations

  real (kind=4) :: S, dX = 0.3
  integer :: i, N = 1000000

  S = 0
  do i=1, N
    S = S + dX
  end do

  write(*,*) "-----"
  write(*,*) " do i=1, 100000; S = S + 0.3 "
  write(*,*) " S = ", S ! result S = 3027.90
  write(*,*) " instead of S = 30000 "
  write(*,*)

  do i=1, N
    S = S - dX
  end do
  write(*,*) " do i=1, 1000000; S = S - 0.3 "
  write(*,*) " S = ", S ! result S = 26.1582
  write(*,*) " instead of S = 0 "
  write(*,*)

  write(*,*) " do i=1, 1000000; S = S + 1./1000000 "
  S = 0;
  do i=1, N
    S = S + 1./N
  end do
  write(*,*) " S = ", S ! result S = 26.1582
  write(*,*) " instead of S = 1 "
  write(*,*)

end subroutine
```

Listing 3.1: Round_off.f90

In this second example, an infinite loop is written to check the smallest value ϵ that being added to the unity gives a result different to the unity. Fortran implements this value by means of the intrinsic function `epsilon(x)`.

```

subroutine loss_of_precision

  real :: x, eps = 1

  do while (eps>0)

    write(*,*) " Enter eps =? (0:exit) "; read(*,*) eps
    x = 1 + eps

    write(*,*) " eps = ", eps, " epsilon = ", epsilon(x)
    write(*,'(a, e35.25)') " x = 1 + eps = ", x

  end do
end subroutine

```

Listing 3.2: Round_off.f90

The above examples illustrate two main problems when working with real variables and real constants:

1. Real constant numbers do not have an exact representation in the computer.
2. Operations with reals give rise to round-off errors that could be important.

The first problem is illustrated with the following example:

```
write(*,'(a,f20.15)') "Constant 1.1 with single precision = ", 1.1
```

which gives the result:

```
Constant 1.1 with single precision = 1.100000023841858
```

In this case, the default real kind option of the compiler is single precision. There are two options to improve the precision for the constant 1.1

1. Specify the real kind of the constant by writing 1.1d0
2. Modify the default real kind of the compiler.

When executing the following code with default double precision for real variables and constants:

```
write(*,'(a,f20.15)') "Constant 1.1 with double precision = ", 1.1
```

the result gives:

```
Constant 1.1 with double precision = 1.1000000000000000
```

The second problem related to round-off errors of real operations is even more complicated but the same criteria applies to increase the accuracy of operations. To assure seven significant digits in a real operation, single precision is enough. To assure fifteen significant digits, double precision is needed. In the same manner, this double precision can be attained by specifying the real kind of variables or by configuring the default real kind by means of a compiler option. The explanation of seven or fifteen significant digits associated to single or double precision is explained in the next sections.

3.2 Fixed-point and Floating-point representation

There are two main ways to represent real numbers in computers:

- Fixed-point representation. The real number is represented by its integer part and its decimal part (e.g. 55.88).
- Floating-point representation. The real number is represented by its mantissa and its exponent (e.g. 0.5588e02).

Fixed-point representation fixes the position of the binary point that separates the integer part from the decimal part. Then, the bits reserved for both parts are prefixed. This method has the advantage that processing the numbers is simpler but has strong limitations in the available range to work with.

Nowadays, computers use the floating point representation through the standard IEEE 754. The numbers are stored similarly to the way standard scientific notation is written. Hence, a number of bits store the mantissa of the number and another part of bits store the exponent of that number. Notice that the exponent is telling how many positions the decimal point allows to position the decimal point to the left or to the right in the mantissa, huge numbers and tiny numbers can be represented. As an inconvenient, for the same number of bytes, this representation is slower to process and has lower resolution than fixed-point representation.

As an example, consider a fixed point 1-byte precision number with 5 bits the integer part and 3 bits for the decimal part as it is shown in Figure 3.1.

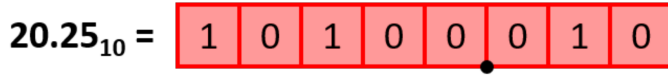


Figure 3.1: Representation of a couple of numbers with fixed point and floating point formats using only 1 byte (8 bits).

Giving a binary fixed point representation with 8 bits (b_7, \dots, b_0) , the decimal number is obtained by:

$$x = b_7 2^4 + b_6 2^3 + b_5 2^2 + b_4 2^1 + b_3 2^0 + b_2 2^{-1} + b_1 2^{-2} + b_0 2^{-3}$$

Hence, the number 20.25 is 10100.010 in binary fixed point representation. Notice that the smallest number that can be represented would be 00000.001 which is only 0.125 and also notice that the biggest number available with this method is 11111.111 which is 31.875. This representation allows numbers from $0.125 = 2^{-3}$ to 31.875 with jumps of 0.125. If 2, 4, 8 or 16 bytes are used to represent this number the range increases and the distance among numbers decreases but the range is still very limited.

In order to compare with floating-point representation, let's consider a mini-float of 8 bits with the same memory size that the above fixed point representation. Consider a mini-float with 4 bits for the exponent (1 for its sign and 3 for its value) and 4 bits for the mantissa. Giving a binary floating point representation with 8 bits (b_7, \dots, b_0) , the decimal number is obtained by:

$$x = m 2^e, \quad m = b_7 2^{-1} + b_6 2^{-2} + b_5 2^{-3} + b_4 2^{-3}, \quad e = \pm(b_2 2^2 + b_1 2^1 + b_0 2^0)$$

The maximum value with this representation is:

$$1111 \ 1111 = 15 \times 2^7$$

and the smallest number is:

$$0001 \ 0111 = 2^{-7}.$$

The distance between the maximum number and its closest number is: 2^7 and the distance between the minimum number and its closest number is: 2^{-7} .

To conclude, for the same number of bits, fixed-point numbers are equal-spaced along the whole range but with a smaller range (31.875 vs 15×2^7). The distance among real numbers in the fixed-point representation is constant and equal to 0.125. However, in the floating-point representation, the distance among real numbers changes from 2^7 to 2^{-7} depending on the exponent of the real number. It is also important to notice that the total amount of real numbers in fixed point or floating point representation is the same but their distribution and range are different.

Important Notice

1. Fixed-point format has constant distance among all representable real numbers and a small range.
2. Floating-point format has variable distance among all representable real numbers with a huge range.

3.3 Representation in IEEE 754

The best way to understand the differences between both representations is with an example.

Important Notice

Before the example, revise these concepts in the context of numerical calculations:

- **Precision/Resolution:** The smallest change that can be represented in floating point representation, which means, how exactly we can specify a number we want to represent. Sometimes it is used the word “precision” for the number of bits used (remember that a change in the least significant bit is the smallest available change) and “resolution” for that specific quantity changed. For every value in the exponent, the resolution is fixed by the value of the least significant bit of the mantissa in floating-point arithmetic, or the number bits in the mantissa which is similar.
- **Accuracy:** How close a value is to what it is meant to be or the closeness of floating point representation to the actual value. This can be used in the result of an operation or the assignment of a constant to a variable, the result should be a value while the computer gives a nearer but not equal number. The accuracy is governed also by the number of bits used in the mantissa. The obligation to round numbers (rounding error) is related to accuracy. Dedicating more bits for the mantissa increases the resolution and the precision.
- **Range:** Highest and tiniest number representable with the number of bits available. This concept is governed by the number of bits of the exponent so for a fixed number of bits (precision), dedicating more bits to the exponent and less bits to the mantissa extends the range but decreases the accuracy and resolution.

Important Notice

There are some basic concepts that you can revise to confront this section with all the tools. First of all, take a look at the positional numeral systems and how to convert a decimal number to a pure binary number either integer or fractional and either positive or negative. Then, take a look at the standard scientific notation in decimal system used to express large and tiny numbers, at this point revise the scientific notation in binary system which is similar. Once this is done, some conclusions can be obtained:

1. The computer can not use the symbol (-) for negative values and it neither uses an explicit decimal point to represent the fractional part. It needs a method to express reals (and specially negative reals) just using 0's and 1's.
2. Except for the number $0_{10} = 0_2$, all the numbers to be expressed in binary system are going to have the digit 1 as first significant digit. Exactly like in a decimal number the first significant digit must be a digit from 1 to 9.
3. Exactly like the case of a decimal number expressed in standard scientific notation ($r = c \cdot b^e$ with $b = 10$), where the exponents is chosen to accomplish $|c| \in [1, 10)$, for a binary number expressed in standard scientific notation ($r = c \cdot b^e$ with $b = 2$), the value of $|c| \in [1, 2)$ (except for the 0, which is an special value). That condition is expressed in decimal form, while it is referred to a binary number, the meaning is not other than the mantissa of a binary number expressed in binary standard scientific notation is going to be contained between 1 and 2 while expressed in decimal form. In binary form, this means that the mantissa of the number always has the digit 1 followed by the fractional point and a series of binary digits. Well, notice that standard IEEE 754 for floating point numbers is really similar to scientific notation in base 2.

Just as an example, a binary number in standard scientific notation could be $1.01101 \cdot 2^3$. In this case, the mantissa is written in binary form while the base and exponent are written in decimal form, it does not change the meaning. We could write all the number in binary form like: $1.01101_2 \cdot 10_2^{11_2}$

Now the conversion between decimal and floating-point number in the standard IEEE can be treated.

The current standard implemented in most machines to work with floating-point Arithmetic is the IEEE 754 Standard (see table 3.1). Let's take a look at the basics of a representation in this standard. Similarly to the scientific notation, this

representation assumes that the binary point is located immediately at the right of the first binary digit, the sign bit as will be seen later. The exponent then will float the binary point to the right or to the left depending on the value.

This information can also be accessed by code, take a look at the useful intrinsic functions that can be used in Fortran.

```
real(kind=4) :: x
real(kind=8) :: y

write(*,*) 'Declaration of x with - real(kind = 4):: x'
write(*,*) 'Maximum value', huge(x)
write(*,*) 'Minimum value', tiny(x)
write(*,*) 'Round_off', epsilon(x)
write(*,*) 'Significant digits', precision(x)

write(*,*) 'Declaration of y with - real(kind = 8) :: y'
write(*,*) 'Maximum value', huge(y)
write(*,*) 'Minimum value', tiny(y)
write(*,*) 'Round_off', epsilon(y)
write(*,*) 'Significant digits', precision(y)
```

which results in:

```
Declaration of x with - real(kind = 4):: x
Maximum value  3.4028235E+38
Minimum value  1.1754944E-38
Round_off  1.1920929E-07
Significant digits          6

Declaration of y with - real(kind = 8) :: y
Maximum value  1.797693134862316E+308
Minimum value  2.225073858507201E-308
Round_off  2.220446049250313E-016
Significant digits          15
```

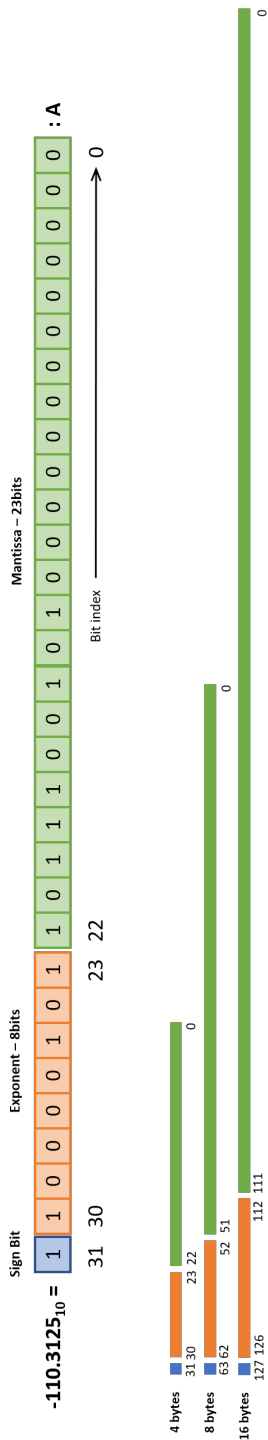


Figure 3.2: Main formats in the standard IEEE 754: single, double and quadruple precision with an example in single precision.

Name	Sign	Exp.	Mantissa	Exp. Bias	Bits precision	Normalized range	Approximate decimal	Precision
Single precision (binary32)	1	8	23	+127	24	$\pm 2^{-126}$ to $\pm 2^{127+1}$	$\pm 1.18 \cdot 10^{38}$ to $\pm 3.4 \cdot 10^{38}$	~7.2 digits
Double precision (binary64)	1	11	52	+1023	53	$\pm 2^{-1022}$ to $\pm 2^{1023+1}$	$\pm 2.23 \cdot 10^{308}$ to $\pm 1.80 \cdot 10^{308}$	~15.9 digits
Quadruple precision (binary128)	1	15	112	+16383	113	$\pm 2^{-16382}$ to $\pm 2^{16383+1}$	$\pm 3.3621 \cdot 10^{-4932}$ to $\pm 1.1897 \cdot 10^{4932}$	~19.2 digits

Table 3.1: Main properties of the different precisions covered by the IEEE 754 standard.

The explanation and example is done with single precision, the 4-bytes (32bits) format. Check in the table 3.1 that 23bits are reserved for mantissa, 8 bits for the exponent and 1 bit for the sign. Actually, since the mantissa always omit the value 1 before the binary point in the representation (because it is always a 1 in normalized notation) the actual precision of the mantissa is 24 bits and not 23. Although it is not represented, when the computer needs to process any number it adds that value 1 to operate properly with reals. In the memory of the machine the standard says that the first bit is the sign bit, followed by the bits of the exponent and followed by the 23 mantissa bits. This order is the same for any precision but with different quantity of bits.

Let's convert the number -110.3125 to a 32bits IEEE 754 floating point explaining step by step.

1. First of all give a value to the sign bit, which is decided by the sign of the mantissa. If your number is negative use a 1, if it is positive, a 0. In this case the example is negative so our first bit in the memory is a 1.
2. Convert the number to pure binary, notice that pure binary is similar to the fixed-point representation but in pure binary there are not limits on the number of bits. Since we have defined the sign in the previous step, consider now the number as positive. The whole part is $110_{10} = 1101110_2$ and the decimal part is $0.3125_{10} = 0.0101_2$. Hence, the complete number in pure binary is $110.3125_{10} = 1101110.0101_2$.
3. Put the binary point in the first position, according to the scientific notation and then find the unbiased exponent:
 $1101110.0101_2 = 1.1011100101_2 \cdot 2^6$.
4. Omit the first significant digit, which is always a 1, there is no need to waste a representable digit covering this information: $1.1011100101_2 \cdot 2^6 \rightarrow .1011100101_2 \cdot 2^6$.
5. Calculate the biased exponent. For 4-bytes precision (8 bits reserved for exponent) the bias is +127. We are not covering here the advantages or disadvantages of the biased exponent against two's complement or one's complement. All the options allow to store positive and negative exponents in a more or less proximate way to the pure binary representation, in the case of the bias all the exponents have an offset from the smallest value. In this precision, the smallest available value is -126 and the highest is $+127$. Hence, our exponent of 6 is covered by the value $6 + 127 = 133_{10} = 10000101_2$.
6. Fill in the rest of mantissa digits with zeros at the right to complete the whole number representation. The result is:

1 10000101 1011100101000000000000_{2,IEEE754}

Exponent	Mantissa	Value represented
All 0's	All 0's	± 0 depending on the sign bit, they are equal
All 1's	All 0's	$\pm \infty$ depending on the sign bit
All 1's	NOT all 0's	Not a Number (NaN)
All 0's	NOT all 0's	Denormalised numbers

Table 3.2: Special values covered by the IEEE 754 standard.

In some cases, instead of filling with zeros maybe is necessary to truncate or round the excess of digits obtained in the conversion to pure binary, either because the result has more than 23 decimal digits or because, for example, we have obtained a periodic binary number. The normal way to do it is that one called “round to the nearest, ties to even”: if the 24th bit is a zero we chop the digits, if it is a 1 we add one to the 23th bit. The special case of a 1 in the 24th bit followed by zeros involves (similarly to the way of rounding in decimal system) that, if the 23th is a 1, we add one to the mantissa, if the 23th is a zero, we chop the digits.

In order to convert from the IEEE standard to decimal system we have to cover the same steps in the opposite order. In that case, do not forget to add 1 to the mantissa and always take into account that some rounding could has happened in the process of converting the value to binary IEEE 754 representation. Hence, is possible that after converting the number to binary and again to decimal system, the result is not exactly the same.

Finally, the standard reserves some exponent values to special situations (see Table 3.2). It is important to always consider that there are two numbers equal to 0, ± 0 . Infinity and NaN's are essential to denote whether the result of a computation is too large to be represented in IEEE-754 (then infinity is used, for example, when the maximum exponent is exceeded in an operation, also called overflow) or a variable didn't obtained a known value or it is illegal (then NaN is used). The operations between those special values are totally defined in the standard (see Table 3.3). The denormalised numbers are used when underflow occurs, which means, a value is obtained in the gap that exists between the smallest normalised number representable by the standard and the same negative value. That gap is many orders of magnitude larger than the machine epsilon (the distance between two representable values outside the gap, this is treated in the section 3.4). With the denormalised numbers (or subnormal numbers) a gradual underflow is achieved. Said in other words, the numbers too small to be represented (and then forced to be replaced by zero) are gradually decreased. In this case the number does not have an assumed leading one before the binary point, it is a zero. The range of the mantissa is then $|c| \in [0, 1)$.

Notice that the exponent with all 0's would correspond to 0_{10} for biased expo-

Operation	Result
$n/(\pm\infty)$	0
$\pm\infty * (\pm\infty)$	$\pm\infty$
$\pm \text{nonZero} / \pm 0$	$\pm\infty$
$\pm\text{finite} * \pm\infty$	$\pm\infty$
$\infty + \infty$ or $\infty - (-\infty)$	$+\infty$
$-\infty - \infty$ or $-\infty + (-\infty)$	$-\infty$
$\pm 0 / \pm 0$	<i>NaN</i>
$\pm\infty / \pm\infty$	<i>NaN</i>
$\pm\infty * 0$	<i>NaN</i>
$\text{NaN} == \text{NaN}$	<i>False</i>

Table 3.3: Special operations covered by the IEEE 754 standard.

nent while the exponent all 1's would be equal to 255_{10} for the biased exponent in simple precision.

An essential concept to manage when programming is the relation between the precision used in a number and his decimal significant digits, which means, the number of reliable digits. This concept is broaden in the next section, however, notice now that when a decimal value is written in a program to be assigned to a variable, used as a constant or as a named constant, the binary value stored is the nearest IEEE 754 floating point value to the expected value. We always have to work with a rounding error, hence, the main question is how many digits are are reliable in the assignment of a variable? Well, it depends on the precision used, in the case of simple precision normally the first 7 digits are reliable but remember that the resolution is not the same everywhere. In some places the binary representation is dense and extra digits can be considered, in other places less digits.

As a quick approximation, consider that $\log_2(10) \approx 3.32$ ($2^n = 10$) binary digits are needed in order to represent one single decimal digit and with simple precision we count on 24 binary digits for mantissa (23 + omitted leading 1) so: $24/3.32 = 7.2$ decimal digits are represented. Notice that this does not mean that the first 7 digits of the represented number are equal to the 7 first digits of the expected value, it means that the relative error between both numbers is in the order of magnitude of $1e-7$.

Just consider the number 1.3_{10} , in the standard IEEE 754 in simple precision this value is stored as

$$0 \ 01111111 \ 01001100110011001100110_2 = 1.2999999523162841796875_{10}$$

which means that the error caused by the conversion is $-4.76837158203125E-8$ while only the first digit is the same. Furthermore, this is only the error caused

by the conversion to the binary value, more errors can appear when the program operates with that value.

Important Notice

Notice that not all the integers contained in the representable range of real numbers have exact representation in this set. In the case of simple precision for example, at least all the integers with 6 or less significant decimal digits can be converted to a IEEE 754 value and not lose precision in the conversion. Some integers with 9 digits can also be converted but more than 9 digits is inevitably related to loss of precision. As an example, the number 899565_{10} is exactly transformed, but the number 45962178_{10} is converted to 45962176_{10} with an error of -2 units.

With the following subroutines and functions you can follow the deconstruction of a real number through his internal bits representation. This representation is charged in a string of bits and once extracted the different parts of the string (sign bit, exponent and mantissa), the value is reconstructed according to the standard. Notice that this code covers the normalized numbers and not the special values.

Either single, double or quadruple precision, the number is converted to a quadruple precision number. Notice that this conversion carries the error made in the first assignation (simple, double or quadruple) and the assignment to quadruple just simplifies the rest of the code since it is developed only for one kind. If a simple precision value is introduced in the program, where only 32bits are used, at the moment of assigning to the 128bits, lots of values will be filled with zeros. Later, the program extracts sign, exponent and mantissa. While sign and exponent are charged to an integer variable, the mantissa is stored in a quadruple precision real value (already adding the omitted 1). Taking into account the bias for the exponent and the basic conversion from binary to decimal the number can be reconstructed. While representing in the screen the bits, do not forget that leading zeros are not displayed, however, they are there and blanks are left instead.

This program is not only useful to understand the conversion but also to make tests with the same number in different precisions. Call the program with the same number declared as simple, double and quadruple precision and take a care look at the big differences in the reconstructed number, specially the significant digits for all the precisions.

```

subroutine mantissa_exponent_base_2( x, m, e, s )
  real (kind=16), intent(in) :: x
  real (kind=16), intent(out) :: m      ! mantissa
  integer, intent(out) :: e, s          ! exponent and sign

  character (len=128) :: bits
  real (kind=16) :: xr
  integer :: i

  write(bits, '(B128)' ) x
  if (bits(1:1) == '1' ) then
    s = -1
  else
    bits(1:1) = '0' ! sign positive
    s = +1
  end if

  m = normalized_mantissa( bits(17:128) )
  e = biased_exponent( bits(2:16) )
  xr = s * m * 2.**e

  open( 6, FILE='CON', STATUS='UNKNOWN', RECL=200 )
  write(6, '(a)' ) "
  -----
  write(6, '(a35, ES40.32)' ) " x = ", x
  write(6, '(a35, a128)' ) "Internal representation of x = ", bits
    (1:128)
  write(6, '(a35, a112)' ) "Mantissa of x = ", bits(17:128)
  write(6, '(a35, a15)' ) "Biased exponent of x = ", bits(2:16)
  write(6, '(a35, ES40.32)' ) "Normalized mantissa x = ", m
  write(6, '(a35, i10)' ) "Exponent of x = ", e
  write(6, '(a35, ES40.32)' ) "Reconstruction x = ", xr
  write(6, '(a)' ) "
  -----
end subroutine

```

Listing 3.3: Subroutine to reconstruct the number in IEEE_representation.f90

```
elemental function normalized_mantissa( string ) result(mantissa)
  character(len=*) , intent(in) :: string
  real (kind=16) :: mantissa

  integer :: i, M

  M = len_trim(string)
  mantissa = 1
  do i=1, M

    if (string(i:i)=='1') then
      mantissa = mantissa + 1./2.**i
    end if

  end do
end function
```

Listing 3.4: normalized_mantissa function in IEEE_representation.f90

```
function biased_exponent( string ) result(e)
  character(len=*), intent(in) :: string
  integer :: e

  integer :: i, M
  integer :: bias

  M = len_trim(string)
  e = 0
  do i=M, 1, -1

    if (string(i:i)=='1') then
      e = e + 2**(M-i)
    end if

  end do

  ! bias
  ! if (M==8) then
  !   bias = 127
  ! else if (M==15) then
  !   bias = 16383
  ! end if

  e = e - bias

end function
```

Listing 3.5: biased_exponent function in IEEE_representation.f90

3.4 Distance between floating point real numbers

It has been introduced in the previous section that there are two reasons why the number you may want to represent in the computer can not be exactly represented and has to be rounded (to an IEEE floating-point value). First and the less common, the number is out of the representable range in the specific precision (either for an overflow or underflow) and second, while the decimal number has finite number of digits, the exact binary representation has infinite digits or just more significant digits than the allowed by the precision (see [2]).

In a previous section the concepts of range and resolution of the numbers that are represented are explained. Also, how both concepts are opposed due to the finite precision. A visually way to understand this is thinking of the exponent as a window between two values, and the mantissa as the offset of an specific number from the initial value of the window [1]. In the decimal system the window tells which two consecutive power-of-ten are we treating: $[0.1,1]$, $[1,10]$, $[10,100]$, etc. and in binary system the windows jump with powers-of-two so $[0.5,1]$, $[1,2]$, $[2,4]$, etc. This window then is divided in equidistant divisions according to the number of bits used for the mantissa. However, notice that from one window to the next one the number of divisions is the same and the length of the window much bigger (each window doubles the previous), then the resolution in that window is lower and the programmer is loosing capability to represent numbers since the absolute value of the number is bigger. Said in other words, the density of IEEE 754 floating-point values near the zero is bigger than the density of values with growing values.

Let's see this with an example, with 23 digits (the omitted leading 1 is always a 1) we have 23bits precision to divide every window, this means $2^{23} = 8688608$ possible values. In the window $[1, 2]$ the resolution is $\frac{(2-1)}{2^{23}} \sim 0.000000119$ while in the window $[32768 = 2^{15}, 65536 = 2^{16}]$ the resolution is $\frac{(65536-32768)}{2^{23}} \sim 0.003906$. Notice the essential consequences of this, the simple step of writing a number in your program to be used as a constant or stored in a variable is going to have an error associated, and this error is going to be higher when working with big numbers than small numbers. For simple precision, like in the previous example, when the window corresponds to $[16777216 = 2^{24}, 8388608 = 2^{23}]$ the absolute error committed is 1.

The density of representable numbers can be represented in the number line, however, for the representation let's consider a lower precision. For example, as it is broaden in [2], consider the case where the base of representation is 2, there are only 3 significant digits in the mantissa and only 4 available exponents $(-1, 0, 1, 2)$ without sign bit (see Figure 3.3). A general number in this system is written: $d.dd \times 2^e$ with $e \in [-1, 2]$ Take a look at the possible decimal values that this system could represent (see table 3.4).

Mantissa/Exponent	-1	0	1	2
1.00	0.5	1	2	4
1.01	0.625	1.25	2.5	5
1.10	0.75	1.5	3	6
1.11	0.875	1.75	3.5	7

Table 3.4: Possible values (shown the decimal value) covered by the example system treated (read text for explanation).

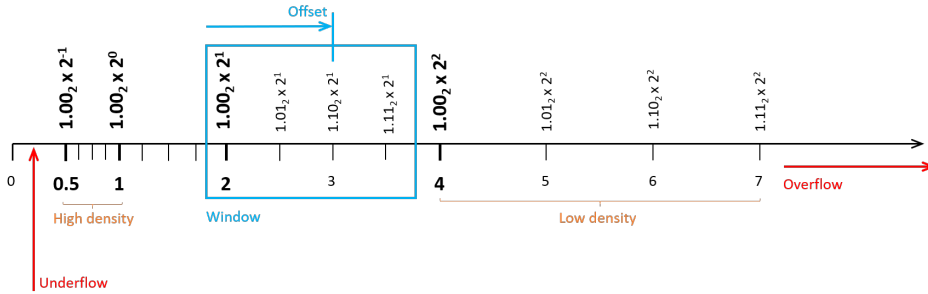


Figure 3.3: Representable numbers in the binary numbers system defined by $p = 3$ and $e \in [-1, 2]$ (read text for more details).

In those cases where the exact number to be represented is approximated by the nearest IEEE 754 number, the absolute error committed is bounded. This happens in the declaration of a variable or the use of a constant value. On the opposite, in the case of operating with variables and constants the result of the calculation could be different from the nearest IEEE value to the exact result.

As an example, consider the binary system explained above, imagine that the program needs to use the value $3.1875_{10} = 11.0011_2 = 1.10011_2 \times 2^1$. This number will be represented as $3_{10} = 1.10_2 \times 2^1$ in the system proposed so the error committed is $0.1875_{10} = 0.0011_2$. Notice that this is 0.011_2 units in the last place (*ulps*). The absolute error in this system, when the exact number is approximated by the nearest floating-point value, is bounded by $0.1_2 = 0.5_{10}$ ulps and the exact absolute error committed for a number z will be $|d.dd - (\frac{z}{2^e})| 2^{3-1}$ ulps. It seems that this absolute error is small for every number to be represented, but take into account that the concept “unit in the last place” depends on the exponent, said in other words, depends on the window of the number to be represented, so 0.5 ulps in the number $z = 0.5625$ means an error of 0.0625_{10} units, but if the number $z = 5.5$ is represented by the number 6_{10} , then the error committed is 0.5_{10} units, which is quite bigger.

Another way to calculate and measure the error when the number is approximated by the nearest value is the relative error. This is the difference between both the exact number and the approximation and divided by the exact value. Since

Definition	Expression
ulp	$\beta\beta^{-p}\beta^e = 0.0000...\beta'x\beta^e$ with $\beta' = \frac{\beta}{2}$ digits
Absolute error of z	$ d.dddd...x\beta^e - z = d.dddd... - \frac{z}{\beta^e} \beta^e$
Maximum absolute error	$\left(\frac{\beta}{2}\right)\beta^{-p}\beta^e = \frac{1}{2}\text{ulp}$
Relative error	$\frac{z - d.dddd...x\beta^e}{z}$
Relative error of the maximum absolute error	$\left[\frac{1}{2}\beta^{-p}, \frac{\beta}{2}\beta^{-p}\right]$

Table 3.5: Different errors committed when the real number z is approximated by the nearest floating-point value.

this value is divided by the real number, the relative error is going to have the same bounds for all the representable numbers, while the absolute error grows for growing exponents, the denominator also grows and the relative errors maintains “constant”. Actually, the relative error does not vary from one window to the next one, it repeats the same behaviour, but it does change inside the window, just take a look at his expression. What it is interesting is to calculate the relative error that is related to the maximum absolute error. In this case, for every exponent (window), the real number varies in the range $z \in [\beta^e, \beta\beta^e)$ while the maximum absolute error is constant $\frac{1}{2}\text{ulp}$ so the relative error ranges in: $\left[\frac{\beta}{2}\beta^{-p}, \frac{1}{2}\beta^{-p}\right]$

Important Notice

Use the absolute error, whether expressed absolutely or with ulps, to understand the rounding error and use the relative error to analyse the results of calculations in the computer.

For a general number system where β is the base of the system, p is the number of digits reserved for the mantissa and e is the exponent, the table 3.5 shows the errors and their bounds. Notice that the definition of the ulp is composed by a part that depends number system (β and p) and another part that depends on the specific exponent that is represented (the window), e .

A good way to understand the expressions and behaviour of these errors is to represented them in the same number line shown above, using that representation system. Because of the high density of the representable number in the IEEE-754 system for simple precision, in that case only the boundaries of the errors are represented, while in the simpler case of only 3 significant digits, is possible to show the exact error for every real number (see Figure 3.4).

For the relative error, take into account that the each number has its own

value, exactly like the absolute error. However, we are more interested in the general behaviour of this error. Since it is calculated dividing the absolute error for the real number (z) notice that for each window the relative error is smaller in the higher values (in the right part of the window) while it is higher in the left part. Furthermore, the relative error is exactly the same for all the windows (it does not vary with the exponent of the number represented). Said that, the relative error is always bounded by $\frac{\beta}{2}\beta^{-p}$ in the left part of each window and by $\frac{1}{2}\beta^{-p}$ in the right part (see Figure 3.5).

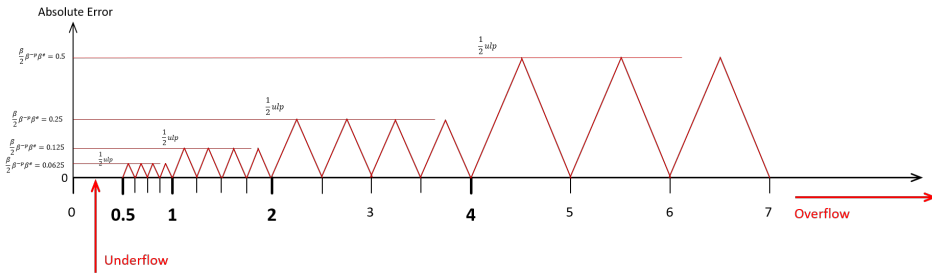


Figure 3.4: Graphic of the absolute error committed when the real numbers are approximated by the nearest floating-point value with the maximum value represented for each window (exponent).

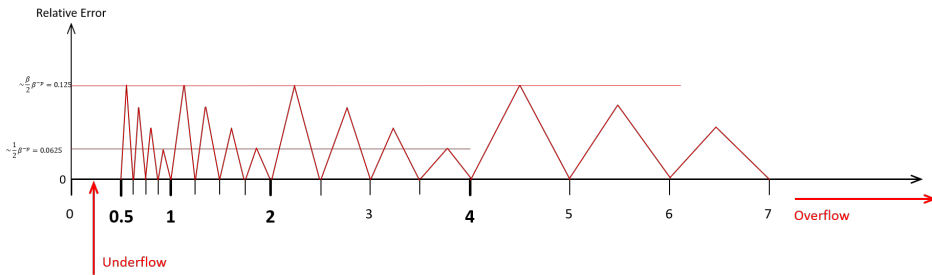


Figure 3.5: Graphic of the relative error committed when the real numbers are approximated by the nearest floating-point value with the maximum value represented for all windows (exponent).

Important Notice

The value $\frac{\beta}{2}\beta^{-p} = \epsilon$ is called machine epsilon, always take into account this magnitude because it bounds the relative error of any number when is rounded to the closest floating-point value. In the case of binary floating-point it can be expressed as $\epsilon = 2^{-p}$ taking the value of $2^{-24} \sim 5.96e - 8$ in simple precision (23 bits plus 1 implicit bit), $2^{-53} \sim 1.11e - 16$ in double precision and $2^{-113} \sim 9.63e - 35$ in quadruple precision. In other conventions, the machine epsilon is considered the value $\beta^{1-p} = \epsilon$ which is the ulp for the value 1.0 and in this case it is defined as: *machine epsilon is defined as the difference between 1 and the next larger floating point number.*

The example above helps to understand the behaviour of both errors, for a real situation where simple or double precision of IEEE standard is used, the density of numbers is much higher but the curve is the same. For real applications a programmer is more interested in the upper bound of the absolute and relative errors already expressed in decimal system since it is the numbers system used for a numerical simulation. This values are represented in the Figures ?? and ??.

Now, with all the explanations above, the following examples are easy to understand based on converting the reals to floating-point values from the IEEE standard.

It can be concluded from this chapter that the density of numbers that can be represented near 0 is enormous, while this density decreases as we move away from 0. This property of the floating point representation can be used by the numerical programmer through the normalization of the operations to perform. For example, consider an ODE solution where the whole equation is dimensionless with all the terms divided by the maximum values that the variables are going to reach. In this case, the solution will vary between 0 and 1 where the IEEE 754 floating point set is more dense and the absolute error is lower. Anyway, do not forget that the relative error performed in a operation is going to be constant.

3.5 Reconstruction from its internal binary representation

Use scientific notation with ES format (normalized mantissa).

$$x = (-1)^s \sum_{i=0}^{M-1} b_i 2^{-i} 2^e$$

with $b_0 = 1$.

$$\epsilon = m_1 2 - m_2 2^e = 2^{e-M}$$

3.6 Writing floating point expressions

Exactly like in the case of integers, in a code, the programmer works with real variables and real constants, and it is interesting to manage properly the precision reserved for both. Specially in strong typed languages like Fortran, the programmer must understand how assignments are performed and the typical errors that could appear when types of variables are not properly defined. If the type of a variable is defined in the declaration of the variable it is typically used `real(kind = n)` :: or `real*n` :: where `n` is 4, 8 or 16. For each situation the strategy can vary, however, it can be interesting to write the program independently of the precision, this means that the declaration of variables is made with no kind specification, just `real :: x`. If the “kind” parameter is not specified, the kind of the variable is the default value, which can be modified in the compiler options. The main advantage of this strategy is that the code can be executed with all the precisions by changing an option of the compiler. Thus, the program is not dependent on the precision imposed when it was written.

In the case of constants the precision can be also defined when used, just add `_k` besides the value of the constant with `k` is 4, 8 or 16 and that kind will be imposed to the constant. However, once again, this is not the common way to do it, consider not declaring the kind for the constants neither so it automatically adopts the default real kind value and, in case different precision is needed for the whole program, that option is changed in the compiler options. Unless you have changed it, the default real kind is simple precision (`kind = 4`).

When the constant has an exponent part, for example, `1.35E+9` which means `1.35 * 10**9` the constant is automatically a simple precision value because of the “E”, unless the kind parameter besides specifies anything different (`1.35E+9_8` would be double precision). If a “D” or a “Q” is used, then the constant is double or quadruple precision instead, in this case no optional parameter `_k` can be used since the precision is already chosen. All these different ways to declare the same thing should be used in those cases the programmer needs it, consider not declaring the precision for each real value in the program and make it independent of the precision. When the constant does not have an exponent and does not have decimal part (for example the real value `8`.) the number written must have a decimal point to tell the compiler that is a real value, whether the kind parameter is imposed or not. Otherwise, the compiler will consider it an integer value. Later in the text is treated the possible errors that appears commonly in a program when operations

between different types of variables and constants are performed.

Important Notice

As a conclusion, get used to think first the needs of your program regarding the precision of the real variables and constants so you can define it properly in their declarations. In case you prefer to write a code that does not depend on the precision do not specify values for precisions and choose the proper value in the compiler options for each compilation and execution. Anyway, do not forget to always specify that a constant is a real value, so it is not confused with an integer. Use one of the following ways:

1. Use a decimal point after the integer part of the number so it is clear that the number is treated as a real. For example `3. * 78.` is an operation between reals.
2. If the real constant has exponent use any of the symbols “E”, “D” or “Q” depending on the precision. Use “E” if you do not want to specify precision and let the compiler use the default value. In this case the decimal point is optional. For example `45E-3 * 7.E2` is also an operation between reals with the default precision imposed by the compiler. In this case, after the symbol it must be an integer number, but you can use the value zero, for example `7e0 * 34.`
3. If you need to work with an integer variable but operate it as a real value, then use the intrinsic function `real (N)`. In this situation the variable `N` which may be declared as an integer value could be used in operations with reals properly.

Unicamente tener cuidado con la asignacion de una constante o variable de doble a simple que no estas aprovechando toda la precision y de simple a doble, que se hace la asignacion con el error que lleve el numero en simple precision.

Take a look at the following example of simple arithmetic operations between different data types (whether different type or kind in the same type):

```
write(*,'(a20, f17.15)') '1.1/2.'           ', 1.1/2.
write(*,'(a20, f17.15)') '1.1e0/2e0'         ', 1.1e0/2e0
write(*,'(a20, f17.15)') '1.1d0/2d0'         ', 1.1d0/2d0

write(*,'(a20, f17.15)') '1.1/2'             ', 1.1/2
write(*,'(a20, f17.15)') '1.1/2d0'           ', 1.1/2d0
```

The result when the compiler has default real kind defined as simple precision is the following, try to understand why those results:

```
1.1/2.      0.550000011920929
1.1e0/2e0   0.550000011920929
1.1d0/2d0   0.550000000000000

1.1/2       0.550000011920929

1.1/2d0     0.550000011920929
```

Let's analyse each example and obtain some conclusions from them.

The first three examples performs the same operation, it divides the number 1.1 (which does not have exact representation in the standard IEEE 754) by two, which is exact. They perform the operation with no precision imposed the first two of them and in double precision the third one. However, if we change the default real kind in the compiler options and impose to treat constants and variables as double precision by default, the result is:

```
1.1/2.      0.550000000000000
1.1e0/2e0   0.550000000000000
1.1d0/2d0   0.550000000000000
```

The conclusion to obtain from this is: write codes that do not depend on a precision, just use the first case (1.1/2.) and change the compiler options whether you need simple precision or double precision result. It is not necessary to write in each operation `d0` to make sure that the operation is performed in double precision, just configure the compiler to treat all constants (and variables) as double precision. In the next example it is demonstrated that writing (2.) is not necessary neither. However, notice that in order to force the constant to be real (`e0`) is not needed.

Now take a look at the fourth example, it uses the integer 2 instead of converting it to a real number. While operations between two integer operands or between two real operands are developed as expected, a mixed-mode expression (where different data types are involved) must be treated with care. Arithmetic involving different types of operands or different kinds of the same type (i.e. `real (kind 4)` and `real (kind 8)`) will be carried out by converting the lowest-ranking operand to the highest-ranking operand so the result has this type and kind. The table 3.6 shows the ranking of each type.

This means that the integer 2 is automatically converted to a real value (which has higher-ranking associated) and the operation is performed. If double precision is needed is simple, just change the compiler option and execute the same program:

Data Type	Ranking
LOGICAL(1) and BYTE	Lowest
LOGICAL(2)	.
LOGICAL(4)	.
LOGICAL(8)	.
INTEGER(1)	.
INTEGER(2)	.
INTEGER(4)	.
INTEGER(8)	.
REAL(4)	.
REAL(8)	.
REAL(16)	.
COMPLEX(4)	.
COMPLEX(8)	.
COMPLEX(16)	Highest

Table 3.6: Ranking assigned to each data type, arithmetic will be performed with the highest ranking.

1.1/2 0.5500000000000000

The main conclusion is that there is no need of specifying always that constants are real values if the operation is performed with one operand being already real. But do not forget that at least one operand must define the type of operation to perform, if you do not write at least one real value, you are operating in the integers field and the divisions in the integer field totally ignore the decimal part of the result, so it is truncated (the rest of the operations in the integer field are performed as expected):

`write(*,'(a20, f17.15)') '1/3 ', 1/3`

which results in the :

1/3 0.0000000000000000

The situation can be tricky when more operands are involved:

`write(*,'(a20, f17.15)') '5/2 * 3. = ', 5/2 * 3.`
`write(*,'(a20, f17.15)') '3. * 5/2 = ', 3. * 5/2`

```
5/2 * 3. =      6.0000000000000000
3. * 5/2 =      7.5000000000000000
```

Both are the same operation, however the first example is not properly performed since the precedence of the operation is from left to right in products and divisions. Hence, $5/2$ is operated in first place, and the result is 2 in the integer field, which multiplied by 3. is 6. At least, in the division, it would be nice to force one value to be real with no need of changing the order of the operation.

Take a look at the following examples in real situations: PONER EJEMPLOS REALES

Finally, look at the fifth example, it can be a little tricky to understand at first. Notice that the compiler is configured for default real kind in simple precision so the value 1.1 is simple precision. According to the table above we could think that the value 1.1 is transformed to double precision in order to be operated with the value 2d0. However, the result is clearly carrying with the round-off of the value 1.1 in simple precision. The reason is that the value 1.1 is stored in double precision but no transformed to double precision.

If the same code is executed with default double precision then the value 1.1 is already double precision so there is not problem. Once again, according to our first conclusion, writing 2d0 in both cases is not necessary at all and just blurs the program.

```
1.1/2d0      0.5500000000000000
```

To be explained:

```
x**2  = x * x

y = 2 * x

y = 2d0 * x

y = 2. * x

x**2d0 = exp( 2 * ln x )

x = 1 / 2
x = 1 / real(2)
x = 1 / 2.
```

```

x = 1d0 * i / N ! NO GUSTA

x = i / real(N)

! same numbers
x = 1
x = 1.
x = 1D0
x = 1e0

y = x**2
y = x**2.

```

3.7 Condition number and stability

The condition number of a problem does not know anything about the operations and the round-off errors associated to the algorithm.

An algorithm is stable if every step is well conditioned.

Round-off + unstable = disaster. (e.g. alternate series)

Round-off + stable = bounded error.

Exact arithmetic (no round-off errors) stability is not concerned. Well/Ill conditioned refers to the problem

stability refers to the algorithm.

Example. condition number at $x = 0$ is one but it is unstable.

$$y = \sqrt{1+x} - 1$$

Examples:

1. $1+x$ no problem
 2. $\sqrt{1+x}$ no problem
 3. subtract $\sqrt{1+x} - 1$: BIG PROBLEM
-

look for another algorithm

$$y = \frac{x}{\sqrt{1+x}+1}$$

Backward stability. The significance of the error produced by the algorithm is due only to the conditioning of the problem.

Multiplications no problem

Subtractions near equal values

$$k = \frac{|x|}{|x-y|}$$

$$k = \frac{f(x+\Delta x) - f(x)}{f(x)} / \frac{\Delta x}{x} = \frac{f'(x)x}{f(x)}$$

$$\frac{\Delta y}{y} = \frac{f'(x)x}{f(x)} \frac{\Delta x}{x}$$

Stability

$$\frac{\tilde{f}(x) - f(x)}{f(x)}$$

Examples:

1. $\sqrt{x^2+1} - 1$
2. $\frac{1-\cos(x)}{x^2} = 0.5\left(\frac{\sin(x/2)}{x/2}\right)^2$
3. plot $y = (x-2)^9 x \in [1.95, 2.05]$ 1000 points

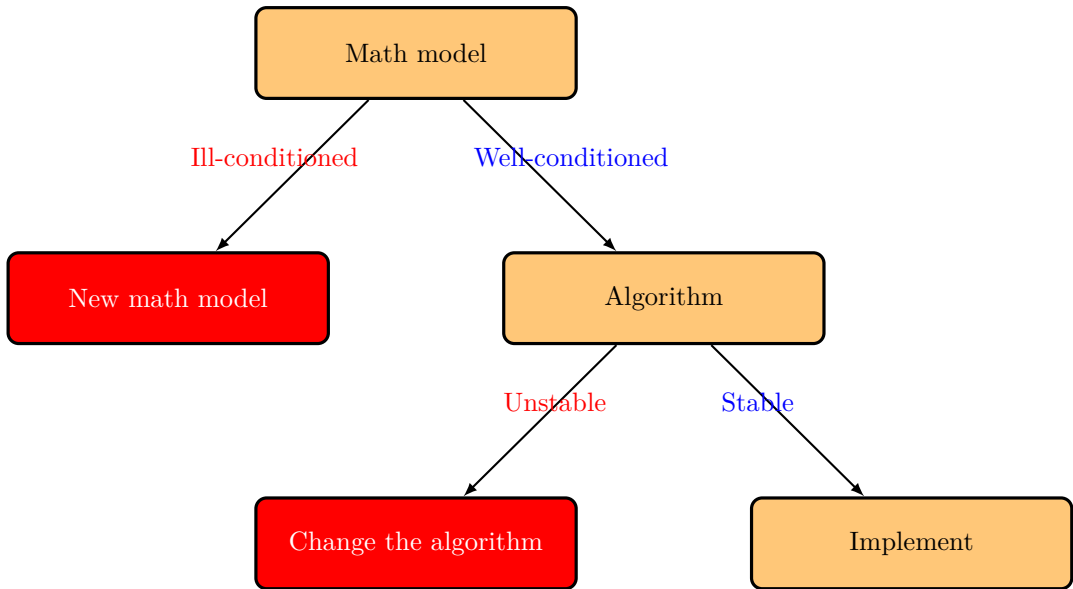


Figure 3.6: Software Development Life Cycle

3.8 Common numerical errors

```

subroutine errors_in_operations

  real (kind=4) :: S, dX = 0.3
  integer :: i, N = 1000000

  S = 0
  do i=1, N
    S = S + dX
  end do

  write(*,*) "-----"
  write(*,*) " do i=1, 1000000; S = S + 0.3 "
  write(*,*) " S = ", S ! result S = 3027.90
  write(*,*) " instead of S = 300000 "
  write(*,*)

  do i=1, N
    S = S - dX
  end do
  write(*,*) " do i=1, 1000000; S = S - 0.3 "
  write(*,*) " S = ", S ! result S = 26.1582
  write(*,*) " instead of S = 0 "
  write(*,*)

  write(*,*) " do i=1, 1000000; S = S + 1./1000000 "
  S = 0;
  do i=1, N

```

```
      S = S + 1./N
    end do
    write(*,*) " S = ", S ! result S = 26.1582
    write(*,*) " instead of S = 1 "
    write(*,*)
end subroutine
```

Listing 3.6: Round_off.f90

3.9 Catastrophic cancellation

```

subroutine catastrophic_cancellation

  real (kind=4) :: a, b, c
  complex (kind=4) :: x1, x2

  complex (kind=16) :: x1e, x2e
  real (kind=16) :: aq, bq, cq

  real (kind=16) :: x0, mu

  write(*,*) " Normal numerical computation "
  write(*,*) " a x**2 + b x + c = 0 "
  x0 = 0.3
  a = 1; b = -2*x0; c = x0**2; x1e = x0; x2e = x0

  call roots_2th(a, b, c, x1, x2 )
  call print_results(a, b, c, x1, x2, x1e, x2e)

  write(*,*) " Catastrophic cancellation example 1 "
  write(*,*) " a x**2 + b x + c = 0 "
  mu = 10000
  aq = 1; bq = -2*mu; cq = 1
  a = aq; b = bq; c = cq;
  call roots_2th(a, b, c, x1, x2 )
  call roots_2thq(aq, bq, cq, x1e, x2e )
  call print_results(a, b, c, x1, x2, x1e, x2e)

  write(*,*) " Solving catastrophic cancellation example 1 "
  write(*,*) " a x**2 + b x + c = 0 "
  call robust_roots_2th(a, b, c, x1, x2 )
  call print_results(a, b, c, x1, x2, x1e, x2e)

  write(*,*) " Catastrophic cancellation example 2 "
  write(*,*) " y = sqrt( x**2 + 1 ) - sqrt(x**2 + 2) with x=1000 "
  aq = 1000;
  a = aq
  x1 = sqrt( a**2 + 1 ) - sqrt( a**2 + 2 )
  x1e = -1/( sqrt( a**2 + 1 ) + sqrt( a**2 + 2 ) )
  call print_results2("a =", a, "x1 = ", x1, "x1e = ", x1e)

end subroutine

```

Listing 3.7: Round_off.f90

3.10 Truncation errors and round-off errors

```
subroutine summation_example

  integer :: N
  real (kind=4), allocatable :: a(:)
  real (kind=4) :: SN
  integer :: i

  N = 10000000
  allocate( a(N) )

  do i=1, N
    a(i) = 1 /real(i)**2
  end do

  SN = sum( a )
  call print_results("normal ", SN)

  SN = sum( a(N:1:-1) )
  call print_results("inverse summation ", SN)

  SN = jahrSum( a )
  call print_results("jahr ", SN)

  SN = KahanSum( a )
  call print_results("Kahan ", SN)

end subroutine
```

Listing 3.8: Summation_errors.f90

3.11 IEEE exception examples

```
subroutine IEEE_exception_examples

end subroutine
```

Listing 3.9: IEEE_operations.f90

3.12 TO MERGE

1. Calculando el epsilon correcto suma hasta un cierto error en modo debug 2. En ese caso Debug sumar 10000000 de terminos no supone diferencia porque son numeros que no se pueden sumar ya 3. Sumar hacia atras si reduce enormemente el error de floating en cada suma y consigue mucha mas exactitud

4. Pero la suma hacia delante si se hace con el modo Release optimiza el bucle y calcula con mas precision 10000000 de terminos de la suma que el caso en el que para en el epsilon que le corresponde. Esto es magia del compilador que optimiza y seguramente haga sumas parciales hacia atras y por eso da mas exactitud.

5. Correr en modo release pero pedir que lo pinte dentro del bucle hace lo mismo que debug porque no es capaz de aplicar la optimizacion si le pides que pinte cosas entre medias.

In the context of floating-point representation, the intrinsic function ϵ represents the smallest number that added to 1 results in a number higher than 1. This value depends on the number of binary digits reserved to store the floating-point value (as it is covered in the section 3).

It seems clear that adding a number higher than $\epsilon/2$ to 1 will be captured by the floating-point precision jumping to the value $1 + \epsilon$, a lower value will not be added since the result is closer to 1 (and further from $1 + \epsilon$). For a number different than 1, let's say for example $S = 12.54$ a slight modification of the epsilon is needed. However, the idea is the same, the floating-point value does not allow adding more terms to S when those values are less than $\epsilon/2$.

Part III

Advanced programming techniques:

Chapter 1

Overview

One of the main characteristics to reuse code is generic programming. Generic programming is based on abstract variable types that are then instantiated when they are used for specific variable type.

Since Python is non typed language, generic programming in this language is straightforward. However, in Fortran the use of abstract `class(*)` allows to use different data types at run time.

Listing 1.1: `main_advanced.f90`

Chapter 2

Scope

2.1 Introduction

One of the most important matters that we need to understand when we begin to write our own programming codes is the scope. In other words, variables which are public and those which are private. This is called the scope of the visibility of some variable in some part of our code.

In any programming language, the scope of variables, objects, functions or procedures is the set of statements in which the variable can be used or modified. The region of a program in which this variable or identifier is visible is called the scope.

Hence, public and public variables or procedures are specified explicitly. If not, local and global variables are visible. Local variables are those which specified inside the function or subroutine that we are dealing with and global variables are those that can be accessed by common variables of my own module or by inclusions of other modules.

In the following code, variables x, y, z are visible inside the subroutine **Test**. Variable z is a local variable, y is a global variable of module **modB** and x is a global variable of module **modA**. All global variables of **modB** are seen in **Test** by means of the sentence **use modB**. Besides, since **modB** uses **modA**, all global variables of **modA** are seen in **modB**.

2.2 Fortran

Listing 2.1: `modB.f90`

Listing 2.2: `modA.f90`

2.3 Python

Chapter 3

First class functions and lexical scoping

Named parameters and default parameters

3.1 Introduction

3.2 Fortran

Listing 3.1: `First_class_functions.f90`

Listing 3.2: `First_class_functions.f90`

Listing 3.3: `First_class_functions.f90`

Listing 3.4: `First_class_functions.f90`

3.3 Python

```
from numpy import array

def function_examples():

    # Integral of f(x) from a to b
    R = Integral(h, 0., 1.)
    print("\n Integral of h(x) from 0 to 1 =", R )

    # Integral of x**n f(x) from a to b based on Integral
    R = Moment(h, 3, 0., 1.)
    print(" \n Integral of x**3 * h(x) from 0 to 1 =", R )

    # Integral of x**n f(x) from a to b with lambda function
    R = Moment2(h, 3, 0., 1.)
    print(" \n Integral of x**3 * h(x) with lambda f =", R )

def Integral(f, a, b):

    N = 100
    dx = (b-a)/N;
    x = array( [ a + dx*i for i in range(N+1) ] )
    return dx * sum( f(x) )

def Moment(f, n, a, b):

    def g(x):
        return x**n * f(x)

    return Integral( g, a, b)

def Moment2(f, n, a, b):

    return Integral( lambda x: x**n *f(x), a, b)

def h(x):

    return x**2 + x + 1
```

Listing 3.5: First_class_functions.py

Chapter 4

Overloading: operators and functions

4.1 Fortran

Listing 4.1: `roots.f90`

4.2 Python

```
from cmath import sqrt #complex math library

def roots_2th_example():

    print("Roots of a x**2 + b* x + c = 0 ")

    #a, b, c = input(" Enter a, b, c").split()
    a = 1; b = 1; c = 1

    x1 = ( - b + sqrt( b**2 - 4* a*c ) )/(2*a)
    x2 = ( - b - sqrt( b**2 - 4* a*c ) )/(2*a)

    print( " x1=", x1, "x2 =", x2) # output
```

Listing 4.2: roots.py

Chapter 5

Vector operations

5.1 Fortran

Listing 5.1: `vectors_and_matrices.f90`

Listing 5.2: `Fourier.f90`

Listing 5.3: `Fourier.f90`

5.2 Python

```
def Fourier_example():

    N = 24      # Fourier truncated series
    M = 2000    # points to plot

    x = array( [ 2*pi*i/M for i in range(M) ] )
    y = array( list( map( lambda x: Fourier_series( N, x ) , x ) ) )

    plt.plot(x, y)
    plt.show()
```

Listing 5.4: Fourier.py

```
def Fourier_series(N, x):

    M = int(N/2)

    # complex :: c(-N/2:N/2), e(-N/2:N/2)
    # real :: a(0:N/2), b(0:N/2)
    a = array( [ 0. for k in range(M+1) ] )
    b = array( [ 0. for k in range(M+1) ] )
    # WARNING DOESN'T WORK if array is initialized with 0 instead of 0.
    c = array( [ complex(0., 0.) for k in range(-M, M+1) ] )
    e = array( [ exp( 1j * k * x ) for k in range(-M, M+1) ] )

    for k in range(1, M+1):
        b[k] = 2*(-1)**(k+1) / (pi*k)

    # c(0:N/2) = ( ak - I * bk )/2
    # c(-N/2:-1) = conj( c(N/2:1:-1) )
    # WARNING in python c[0:3]= c[0], c[1], c[2]
    # Python is a pain in the ass with indexes starting at 0.
    for k in range(0, M+1):
        c[M+k] = ( a[k] - b[k]* 1j )/2

    for k in range(0, M+1):
        c[k] = conj( c[N-k] )

    f = sum( c * e ).real

    return f
```

Listing 5.5: Fourier.f90

Chapter 6

Polymorphism

6.1 Example with objects

6.1.1 Fortran

Listing 6.1: `polymorphism.f90`

6.1.2 Python

```

class polygon():
    def __init__(self, sides): #constructor
        self.N_sides = len(sides)
        self.length = sides

    def perimeter(self):
        return sum(self.length);

class square(polygon):    #inheritance from figure
    S = 4
    def __init__(self, L):
        super(polygon,self).__init__()
        self.L = L
        self.N_sides = 4
        self.length = [ L, L, L, L ]

    def perimeter(self):    #method overriding
        return 4*self.L

class circle(polygon):
    def __init__(self, R): #constructor
        self.R = R

    def perimeter(self):

        PI = 3.1416
        return 2 * PI * self.R

def polymorphism_example():

    P = polygon( [1., 2., 3. ] )    # polygon
    S = square( 4. )                # square length side = 4
    C = circle( 10. )               # circle of radius = 10

    Polygons = [ P, S, C, P ]

    L = 0
    for P in Polygons:
        L = L + P.perimeter()

    print(" Total perimeter = ", 2 * 6 + 4 * 4 + 2 * 3.1416 * 10      )

```

Listing 6.2: polymorphism.py

6.2 Example with ODEs integration

6.2.1 Python

```
def Euler(U1, t1, t2, F):
    return U1 + (t2-t1) * F(U1, t1)

def oscillator(U, t):
    return array( [ U[1], -U[0] ] )

def complex_oscillator(Z, t):
    return -1j * Z

def test_simulation(U0, Nv, F):

    N = 100
    U = array( zeros( [N+1, Nv], dtype=type(U0) ) )
    time = linspace(0, 10, N+1)
    U[0, :] = U0

    for i in range(N):
        U[i+1,:] = Euler( U[i,:], time[i], time[i+1], F )

    if Nv==1:
        plt.plot( U[:].real, U[:].imag )
    else:
        plt.plot( U[:,0], U[:,1] )

    plt.axis("equal")
    plt.show()

if __name__ == '__main__':
    # Example of a system of two real equations
    U0 = array( [1, 0] )
    test_simulation( U0, 2, oscillator )

    # Example of a complex equation sith the same function
    Z0 = complex( 1, 0 )
    test_simulation( Z0, 1, complex_oscillator )

    # state vector to integrate is polymorphic
```

Listing 6.3: polymorphic_ODES.f90

Chapter 7

Map, filter and reduce

7.1 Fortran

Listing 7.1: `map_filter_reduce.f90`

Listing 7.2: `map_filter_reduce.f90`

7.2 Python

```
def test_map_filter_reduce():

    sp = array( ["123", "111", "444", "555"] )
    m = array( [1, 2, 3, 4] )
    # MAP
    m = array( list( map(str_to_number, sp) ) )
    #m[:] = str_to_number( sp[:] ) # it does not work
    #N = len( sp )
    #for i in range(N) :
    #    m[i] = str_to_number( sp[i] )
    m = g( m ) # it works with functions

    # FILTER
    f = array( list( filter(my_filter, m) ) )
    f = f [ f > 200 ] # similar filter

    # REDUCE
    r = max(m)
    print(" array of strings = ", sp)
    print(" MAP: array of numbers = ", m)
    print(" FILTER: filtered array = ", f)
    print(" REDUCE: max value = ", r)
```

Listing 7.3: map_filter_reduce.py

```
def str_to_number(str):

    return int(str)
```


Listing 7.4: map_filter_reduce.py

Chapter 8

Pointers and targets

8.1 Fortran

Listing 8.1: `n_body_problem.f90`



Listing 8.2: `n_body_problem.f90`

8.2 Python

```
def F_NBody_problem(U, Nb, Nc):  
  
    # Write equations: Solution( body, coordinate, position-velocity )  
    Us = reshape( U, (Nb, Nc, 2) )  
    F = array( zeros(len(U)) )  
    dUs = reshape( F, (Nb, Nc, 2) )  
  
    r = reshape( Us[:, :, 0], (Nb, Nc) )    # position and velocity  
    v = reshape( Us[:, :, 1], (Nb, Nc) )  
  
    drdt = reshape( dUs[:, :, 0], (Nb, Nc) ) # derivatives  
    dvdt = reshape( dUs[:, :, 1], (Nb, Nc) )  
  
    dvdt[:, :] = 0 # WARNING dvdt = 0, does not work  
  
    for i in range(Nb):  
        drdt[i, :] = v[i, :]  
        for j in range(Nb):  
            if j != i:  
                d = r[j, :] - r[i, :]  
                dvdt[i, :] = dvdt[i, :] + d[:] / norm(d)**3  
  
    return F
```

Listing 8.3: n_body_problem.py

```
def Initial_positions_and_velocities( Nc, Nb ):

    U0 = array( zeros(2*Nc*Nb) )
    U1 = reshape( U0, (Nb, Nc, 2) )
    r0 = reshape( U1[:, :, 0], (Nb, Nc) )    # position and velocity
    v0 = reshape( U1[:, :, 1], (Nb, Nc) )

    # body 1
    r0[0,:] = [ 1, 0, 0]
    v0[0,:] = [ 0, 0.4, 0]

    # body 2
    v0[1,:] = [ 0, -0.4, 0]
```

Listing 8.4: n_body_problem.py

```
def Integrate_NBP():

    def F_NBody(U, t):

        return F_NBody_problem( U, Nb, Nc )

    N = 100000    # time steps
    Nb = 4        # bodies
    Nc = 3        # coordinates
    Nt = (N+1) * 2 * Nc * Nb

    t0 = 0; tf = 4 * 3.14
    Time = linspace(t0, tf, N+1) # Time(0:N)

    U0 = Initial_positions_and_velocities( Nc, Nb )

    #U = odeint(F_NBody, U0, Time)
    U = Cauchy_problem( F_NBody, Time, U0, RK4)

    Us = reshape( U, (N+1, Nb, Nc, 2) )
    r = reshape( Us[:, :, :, 0], (N+1, Nb, Nc) )

    for i in range(Nb):
        plt.plot( r[:, i, 0], r[:, i, 1] )
    plt.axis('equal')
    plt.grid()
    plt.show()
```

Listing 8.5: n_body_problem.py

Chapter 9

Cauchy problem

9.1 Cauchy problem solver

9.2 Temporal schemes

9.3 Stability region

9.4 N body problem

Chapter 10

Fractals

10.1 Fortran

10.1.1 VonKoch

```
recursive subroutine VonKoch_curve(xi, xf, n)
    real, intent(in) :: xi(:), xf(:)
    integer, intent(in) :: n

    real :: R(2), alpha, x1(2), x2(2), x3(2)

    if (n == 0) then

        call curve( [ xi(1), xf(1) ], [ xi(2), xf(2) ], 2)

    else

        x1 = xi + (xf - xi) / 3.0
        x3 = xf - (xf - xi) / 3.0

        R = x3 - x1
        alpha = atan2( R(2), R(1) ) + PI / 3.0
        x2 = x1 + norm2(R) * [ cos(alpha), sin(alpha) ]

        call VonKoch_curve(xi, x1, n - 1)
        call VonKoch_curve(x1, x2, n - 1)
        call VonKoch_curve(x2, x3, n - 1)
        call VonKoch_curve(x3, xf, n - 1)

    end if
end subroutine
```

Listing 10.1: VonKoch.f90

10.1.2 Mandelbrot

```
function Mandelbrot_set(x0, xf, y0, yf, N) result(R)
    real, intent(in) :: x0, xf, y0, yf
    integer, intent(in) :: N
    real :: R(N,N)

    integer :: i, j, k
    real :: x, y
    complex :: z, c

    do i=1, N
        do j= 1, N
            x = x0 + (xf-x0)*i/real(N)
            y = y0 + (yf-y0)*j/real(N)
            c = cmplx(x, y)
            z = 0
            do k=1, 1000
                z = z**2 + c
                if (abs(z)>2) exit
            end do
            R(i,j) = abs(z)
        end do
    end do
end function
```

Listing 10.2: Mandelbrot.f90

10.2 Python

10.2.1 VonKoch

10.2.2 Mandelbrot

Chapter 1 1

Games

11.1 Fortran

11.1.1 Sudoku

```
recursive subroutine branches_solver(A, i, j, As)
    integer :: A(9,9), i, j
    integer, intent(out) :: As(9,9)

    integer :: k, k0, i1, j1

    /** solution is the branch which gets the last sudoku cell
    if (i > 9) then
        As = A

    /** explore k valid number in position i,j
    else
        do k = 1, 9
            if ( is_valid(A, i, j, k) ) then

                k0 = A(i, j)
                A(i, j) = k
                call new_cell(k0, k, i, j, i1, j1)
                call branches_solver(A, i1, j1, As)

                /** set A(i,j) to try another branch
                if(k0==0) A(i,j) =0
                call print_branch(i,j)
            end if
        end do

    end if
end subroutine
```

Listing 11.1: Sudoku.f90

11.2 Python

11.2.1 Sudoku

Part IV

Software development

Chapter 1

Software development

1.1 Software development life cycle

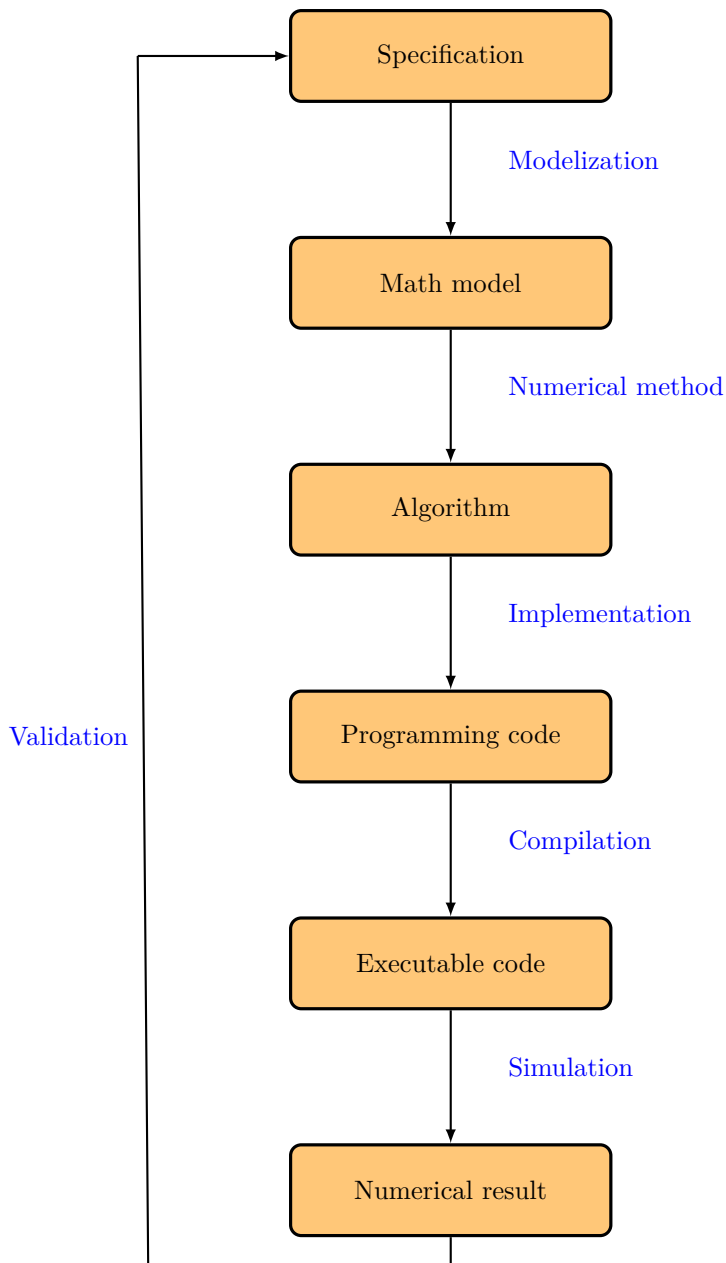


Figure 1.1: Software Development Life Cycle

1.2 Methodology

1.2.1 Names of function, classes and variables

- Snake case
- Camel case.
- Upper case : classes and ...
- Indentation.
- max length.

1.2.2 Code developing methodology

- Top-down. functional paradigm. multi-layer and multi-group decomposition.
- Xtreme programming. Driver: write the code. Reviewer: ask questions. Unit testing. Every function a validation program.
- Continuation techniques.
- Test Driven Development.
 1. Design a write a test.
 2. Write the code.
 3. Run the test and verify.
 4. Refactoring the code. clean up.
 5. Repeat and start new test.

Mantra: RED (fail), GREEN(pass), REFACTOR

- Axiomatic design. max independence. minimum information content. customer needs – functional requirements.

Duplication is prohibited.

Object class represents its purpose.

1.3 Imperative or von Newmann style

second semester PEI1 (Mario)

1.4 Functional paradigm

second semester PEI1 (jahr)

References

- [1] F. SANGLARD. Floating point visually explained. https://fabienanglard.net/floating_point_visually_explained/. [Online; accessed 20-March-2021].
- [2] I. Sun Microsystems. What every computer scientist should know about floating-point arithmetic, 1994. Part No: 800-7895-10.