JAVA FOR THE STUDY OF EVOLUTION

VOLUME V

THE SCIENTIFIC METHOD WITH JAVA

version 3

A Java Companion to Basic Statistics.

José del Carmen Rodríguez Santamaría
Edited and commented by: Write here your name *
The EvolJava Community.
* Other affiliation

Original source: http://evoljava.com/

Dedication

To all those that renounce to the cosiness of fuzzy thought for the sake of precise knowledge and light and all the terrible suffering that these bring forth.

Contents

Dedication i

1 Starting up 1
- 1.1 General terminology . . . 1
- 1.2 Installing Java . . . 3
- 1.3 Purposes and projects . . . 5
- 1.4 Our first program . . . 6
- 1.5 Bugs and debugging . . . 8
- 1.6 How to import a project . . . 11
- 1.7 Review . . . 11

2 Integer numbers 13
- 2.1 Elementary operations . . . 13
- 2.2 Structured programming . . . 15
- 2.3 The if control structure . . . 21
- 2.4 Review . . . 22

3 Using Excel, Calc and Gnumeric 23
- 3.1 The String Type . . . 23
- 3.2 Loops . . . 25
- 3.3 Review . . . 29

4 Decimal numbers 31
- 4.1 Type double . . . 31
- 4.2 The for structure revisited . . . 32
- 4.3 Review . . . 33

5 Arrays 35
- 5.1 Vectors . . . 35
- 5.2 Tables with various columns . . . 38
- 5.3 Review . . . 42

6 Exploratory statistics **43**
6.1 A list of data . . . 43
6.2 Absolute frequency tables . . . 44
6.3 Grouping . . . 45
6.4 Bar charts . . . 50
6.5 Easy LaTeX . . . 51
6.6 PSTricks . . . 53
6.7 Automatic composition of PSTricks code . . . 56
6.8 Conclusion . . . 59

7 The Binomial and the normal distributions **61**
7.1 The binomial distribution . . . 61
7.2 Random numbers . . . 64
7.3 Simulation of a binomial distribution . . . 66
7.4 The standard normal distribution . . . 68
7.5 Conclusion . . . 71

8 The scientific method **73**
8.1 Null hypotheses . . . 73
8.2 Testing a mean . . . 75
8.2.1 Contrast against a single datum . . . 75
8.2.2 Contrast of the mean with various data . . . 84
8.2.3 Contrast with various data: the t distribution . . . 97
8.3 Hypotheses about variances . . . 98
8.3.1 One variance: the chi-square theorem . . . 99
8.3.2 Two variances: the F theorem . . . 100
8.3.3 Experiments with two proportions . . . 103
8.4 Comparison of two means . . . 103
8.4.1 Known variances . . . 104
8.4.2 Unequal inferred variances . . . 104
8.5 The KS test for normality . . . 119
8.6 Independence of attributes . . . 137
8.7 Simulation of anyone distribution . . . 137
8.8 Conclusion . . . 155

9 Evolvable software **157**
9.1 Five different styles . . . 157
9.2 Styles and reusability . . . 161
9.3 Intrigues about the genome . . . 166
9.4 Conclusion . . . 168

10 The chloroplast strategy **169**
10.1 The wrong solution . . . 169
10.2 A right solution: OOP(Object Oriented Programming) . . . 172
10.3 Conclusion . . . 184

11 Numerical analysis over the Z **185**
11.1 p-values . . . 185
11.2 The Riemann and trapezoidal rules . . . 187
11.3 Lab over the z-distribution . . . 189
11.4 Comparing with Gnumeric . . . 191
11.5 Conclusion . . . 193

12 A quest for perfection **195**
12.1 The Simpson's rule . . . 195
12.2 Taylor's theorem . . . 201
12.3 BigDecimals . . . 207
12.4 Big square roots . . . 214
12.5 BigDecimals and the z-distribution . . . 221
12.6 Deformations . . . 230
12.7 Conclusion . . . 237

13 Evolution on letter-zymes **239**
13.1 Evolution . . . 239
13.2 Review . . . 253

14 Taming evolution **255**
14.1 Taming evolution . . . 255
14.2 Improving our approximation . . . 274
14.3 Conclusion . . . 318

15 From densities to p- and critical values **319**
15.1 Some recursive definitions . . . 319
15.2 p-values of the diverse distributions . . . 325
15.3 Critical values . . . 329
15.4 Conclusion . . . 351

16 Project **353**
16.1 Toolboxes . . . 353
16.2 Succeed or perish . . . 354
16.3 Conclusion . . . 355

17 Evolution in perspective **357**
17.1 The skeleton of science . . . 357
17.2 Fundamental problems . . . 359
17.3 Fundamental philosophies . . . 360
17.4 The evolutionary theory . . . 363
17.5 Evolution and science . . . 364
17.6 The reaction of science . . . 373
17.7 Evolution in the open market of ideologies . . . 377
17.8 Challenges . . . 382
17.9 The pedagogic conundrum . . . 386

17.10 A simple test 388
17.11 A personal challenge 388
17.12 Conclusion 388

Answers to exercises **389**

Codex

Programming languages have at presently a very tight syntax, and Java is not an exception. So one must learn the formalism used for each case. We can find here the most usual expressions and the reference number where they appear along the text for the first time.

9 pag 6. Printing a message to the console:
`System.out.println("I am starting up! ");`

20 pag 13. Printing the value of variable x:
`System.out.println("Value of x=" + x );`

26 pag 16. Declaration of a procedure:
public static int sum(int x, int y)

34 pag 21. The if-else structure: REMEMBER THE DOUBLE ==

```
if (i==3) i = 4;
else i=5;
```

39 pag 23. Declaration of a String

```
String a = "Hello";
```

40 pag 24. The Tabulator: `"\t"`

41 pag 24. Concatenation of Strings:

```
String s4 = s1+s2+s3;
```

45 pag 25. The for structure:

```
for(int i = 1; i<7; i++)
 {
  System.out.println(i + tab + i*i);
 }
```

58 pag 31. Type `double`

```
double r = 3.234;
```

66 pag 35. Declaration of an array with integer entries

```
//Declaration
int data[];  // data is the name.
data = new int[ 10 ]; //memory for 10 entries
//Assignation
Data[2] = 3;
```

68 pag 36. Declaration and assignation of a linear array Data

```
 int  Data[] = {1,5,2,4,7,8,9,5,6,3,5};

int n = Data.length; //the length of the array Data.
```

84 pag 45. The negation of a Boolean clause:

```
if (!(i==2)) |(!(j==2))
```

72 pag 38. Declaration and assignation of a two dimensional array

```
int  Data[][] = {{3, 9},
       {4, 10},
       {5, 11},
       {6, 12},
       {7, 13},
       {8, 14}};
```

To address specific entries in a two dimensional array, we must keep in mind that Java begins counting form zero. So the instruction

```
int z = Data[3][1]
```

assigns to z the value 12. In general, the first index address the row and the second the column.

78 pag 44. The Math API

```
 double r = 5.2
 doube rSquare = Math.pow(r,2);
 double rCube = Math.pow(r, 3);
 double rEightRoot = Math.pow(r, 0.125);
double rSquareRoot = Math.sqrt(r);
```

To see other functions, type Math + a period.

85 pag 46. Logic: the and operator &, the or ||, the negation !.

```
if ( (Barriers[j] <= Vect[i]  ) & (Vect[i] < Barriers[j+1]) )
  FreqTable[j][1] =FreqTable[j][1] +1;
```

98 pag 56. Change of type `double` to `int`:

```
double aa = 3.4;
int i = (int) aa;
System.out.println(i);
```

106 pag 65. Random numbers

```
import java.util.Random; //At the preamble of the program
Random r  = new Random(); // somewhere at the beginning

//A random number from 0 to 5
r.nextInt(6)   //In within a method

//Other variants
Random k = new Random(456); //seeded mode for exact repetitions.
r.nextLong();
r.nextFloat();
r.nextDouble();
r.nextGaussian();//normal with mean 0 and dev 1.
```

115 pag 70. Random numbers that obey a normal random variable with predetermined mean equal to μ and deviation σ:

```
double d = r.nextGaussian()
int c = (int) (Math.floor(sigma*d + mu));
```

187 pag 172. Prototypes in OOP (Object Oriented Programming). Go to the text.

227 pag 208. One can declare and assign a value to a **BigDecimal** as follows:

```
int scale = 10 //number of significant ciphers.
BigDecimal arctan_5 = arctan(5, scale);
```

The arithmetic operators of both BigDecimals and BigIntegers function according to the point suffix formalism, which is proper of the OOP:

```
//Arithmetic operators:
//To add x: .add(x);
//To multiply by x: .multiply(x)
//To subtract x: .subtract(x)
//To divide by x: .divide(x, scale, roundingMode)
//Operators can be chained and are executed
//from left to right.
//Example
BigDecimal D = arctan_5.add(7);
```

227 pag 208. The while structure

```
while (this condition meets)
{
  do this and that;
}
```

234 pag 215. StringBuffers:

```
//Declaration and instantiation
StringBuffer sb = new StringBuffer ();//sb has no chars
//Random generator
java.util.Random r = new java.util.Random ();
//Ten random digits are appended to the end of
//StringBuffer sb
for (int i = 0; i < nDigits; i++)
   sb.append (r.nextInt (10));
//To know more, use the point suffix formalism
sb.
```

235 pag 216. BigIntegers:

```
//Initialization from a String
String s = "123456789123456789";
BigInteger n = new BigInteger(s);
//Initialization from a StringBuffer
StringBuffer sb = new StringBuffer ();
    java.util.Random r = new java.util.Random ();
    for (int i = 0; i < nDigits; i++)
      sb.append (r.nextInt (10));
BigInteger n = new  BigInteger (sb.toString ());

//To know more: use the point suffix formalism
n.
```

CHARS, STRINGS AND EVOLUTION

255. pag 239 and beyond. Random numbers and chars

```
import java.util.Random; //At the beginning of the program
Random r  = new Random(); // somewhere at the beginning

//A random number from 1 to 5
r.nextInt(6)   //In within a method

//This is a char or letter
char n = 'j';

//A random upper case char:
char ch =   ( char ) ( r.nextInt( 26 ) + 65 );

//A random lower case char:
ch =  ( char ) ( r.nextInt( 26 ) + 97 );

//Other variants
Random r = new Random (seedValue);

r.nextLong();

r.nextFloat();

r.nextDouble();

r.nextGaussian();
```

Operations with strings that are necessary to simulate evolution. The Replace operator:

```
s2 = s1.replace("ion", "ate");
c.replace(a,b);
```

Conversion of number to string. If the number is of type double, we use:

```
 s1  = String.valueOf(r2);
```

When the number is of type int, we use the next procedure:

```
int i = 3;
Integer n = i;
String s = n.toString();
System.out.println(s);
```

Copy operator

```
s2 = s1.substring(4);
s3 = s1.substring(4,8);
```

The length of a String:
`int` n = s1.length();
Char c is converted into string s

```
String s = c.toString();
```

A number from 0 to 9 is converted to a char.

```
//Integer number i is converted to char z,
//when z is printed it looks just like i.
char z=  ( char ) ( i +48  );
```

A char is withdrawn from a String s and a char is given a numerical code:

```
 char c  =   s.charAt(0);
int sign = Character.getNumericValue(c);
```

Preface

The series *Java for the Study of Evolution* is directed to scientists that want to manage a serious but not excessively expensive tool to study evolution by direct experimentation under perfectly controlled conditions. These requirements cannot be met in nature but only in simulations and mathematical models. In consequence, the series has three main purposes:

1. To endow the community of **researchers in biology and evolution** with *high level programming*, enabling an accurate study of models and simulations of the most diverse nature.

2. To clearly show how this tool is used to study the fundamental questions of evolution.

3. To suggest that the study of Java could be *very fruitful* for **undergraduates** in biological sciences even more than a course centered at calculus.

This Volume V version 3 THE SCIENTIFIC METHOD WITH JAVA works out a clear understanding of the modern scientific method. We use Java simulations to mechanistically comprehend the essence of science. No prerequisites in Java or statistics are assumed. Nevertheless, the companion volume *Basic statistics* might be of great help. The present volume can be used to learn and/or reinforce the fundamentals of Java apart from Vol I of this series. Version 3: more readability, less bugs, less errors, less misconceptions plus all programs are accepted by Netbeans 8.2 and its companion Java.

Every part of all our texts and programs in whatever volume can be edited, commented and criticized. Because every writing implies a responsibility, make explicit at a frontal place whether or not the original material was modified and by whom. Next, give a link to the original source and follow your own style and ethics. For instance, your own comments might be included in special paragraphs entitled *comment*.

Bogotá, Colombia, *José Rodríguez*
February 2017

Introduction

Science has efficiently hidden the essence of the scientific method behind mountains of mysterious mathematics. So, apart from some few persons, people use to imagine that science is the great wizard that has guided us to the truth and that will save the world. Anymore: with the help of Java simulations, we show that the scientific method is an error prone tool for pattern discrimination in the world of ideas about reality. Therefore, science is a craftsmanship whose relation with truth is a matter of discussion.

Given the outermost importance of our theme, it would be desirable for pedagogues and for newcomers to have a self contained material. To achieve that aim, we have decided to recycle a good part of the fist volume of this series *Java for the study of evolution, Vol 1 version 2, fundamentals* and to restrain ourselves from sophistication, say, graphics.

To begin with, we solve elementary tasks, the mean, the variance and all that. Next, we learn how to use random numbers and simulations to pursue a mechanistic understanding of the scientific method. We also face the problem of calculating the critical values of the usual statistics, z, t, χ^2 and F. While we obtain them from our simulations, we also want to find those values by direct receipts using mathematical expressions for density functions. We readily solve the problem up to a rough approximation else up to professional precision but throughout a lengthy calculation. Then we try to make our own path to the world of professionals, in which high accurateness must be achieved instantaneously. Our use of genetic algorithms, simulations of evolution, was not very fruitful. Instead, precision and velocity were found by using standard numerical algorithms.

All our material can be downloaded from

www.evoljava.com

where the course of elementary statistics that guide us together with a multivariate course of statistics with R, both in Spanish, can be found. Programs and LaTeXsources for all texts, including the present one, are also included there:

you can use and misuse every piece of our sources to design your own pedagogic material adopting and implementing your own and personal criteria.

We have now two initiations into Java, Vol I and this Vol V. Nevertheless, after working this volume, the next step for beginners is to work completely Vol I which has additional Java material, such as graphics. In spite of all our efforts, the Reader might experience a feeling that the path of programming is paved with thorns. That is certainly the crude reality. So, keep walking: the central thesis of our work is a modern formulation of the evolutionary theory that is based on the observations that the genome is software, verbal instructions for a special type of computer, the ribosome, and that evolution is a software developer. The Evolutionary Theory now reads: the genome is software and evolution is the developer that is responsible for its existence. So, if you, dear Reader, are not committed to become a great developer, how can you pretend to understand and judge our modern version of the evolutionary theory together with our claim that it is rampantly false?

Chapter 1

Starting up

`Beginning from total ignorance.`

1. ***Purpose.*** *We learn the general terminology, how to acquire Java and NetBeans, how to install them, and how to run a first program.*

1.1 General terminology

There are some usual terms that we specify below.

2. ♣ ***General definitions***

In first place, let us notice that we, human beings, use languages to communicate one with another. A **language** is composed of verbal instructions, a code of interpretation and a semantic content. A **verbal instruction** is a string of sounds or letters that belong in an **alphabet**. Verbal instructions must be deciphered according to a **code of interpretation**, which appears, say, in a dictionary. The deciphered verbal instructions convey information about the world or about actions that must be undertaken and it is in this way that they acquire **semantic content**.

A **computer** is a devise that has the potential ability of executing verbal instructions. A **program** is a set of verbal instructions, written in a specific **programming language**, that can be interpreted and executed by a computer. **Code** is a set of instructions that eventually contain correctly written programs. **Software** is a set of programs that execute a task and very specially that create a friendly interface with the human. **Hardware** is the wiring that converts programs into specific actions. A **robot** is a machine that has a computer that has been programmed to have certain independence.

A **developer** is a person that is committed to the design of software, i.e., of programs with a specific function, which are defined before beginning but that maybe modified along the process. It is absolutely sure that the first trial of a code is inviable. Errors are called **bugs**. But if one manages to produce a

viable code, most surely it will not do what one so eagerly expects. The process of adjusting the code to the predefined purpose is called **debugging**.

It is important to realize that the **genome** contains, apart from other things, a set of verbal instructions to program ribosomes to build specified proteins. All this can be read in (Rothamsted Research, [88], 2001)). The code that allows the deciphering of the verbal instructions contained in the genome is the **genetic code**, a dictionary that contains the correspondence among codons and amino acids. This dictionary is written in the t-RNAs: these molecules have in one extreme a codon and in the other the corresponding amino acid. Amino acids are assembled together by the ribosomes into proteins, which enable surviving in one of two ways: they may serve as structural items or as enzymes to catalyze chemical reactions. Thus, **surviving** is the feature that allows the genome to acquire a semantic content. A **cell** is a robotic entity: it is directed by the genome that is software and has a lot of independence.

So, the genome is for us the natural example of software. Let us emphasize this: **the natural example of software is the genome**. Thus, the evolution of the genome is a sub discipline of a more general matter: the study of the evolution of software in general. In fact, our study of **Java**, our platform of software design, will rise many questions of biological interest and especially in relation with the genome.

The name Java comes from a family of ants that lives in the island of Java. In a single tree but in different leaves one might find various nests of the same colony. The colony has various queens in proportion to the number of nests, which also determines the number of adult workers and immatures. But the numbers of males and of new queens go a path on their own.

To use Java directly is not as easy as using it in duo with NetBeans or NetBeans, which are precious interfaces of Java with the developer. All what follows refers to this ensemble that is generically referred to as Java. In strict sense, **Java is a great project** whose grandiosity might overwhelm at anytime. To begin, let us attempt a structural approach to it.

3. ♣ ***Definitions.*** *A* ***method*** *in Java is a portion of code that is designed to fill a determined function. A set of methods maybe gathered in a* ***class*** *to execute a great task. A class must be saved in a file with the same name, a relation that is case-sensitive. Classes are organized in books that are called* ***projects****.*

Java is the answer of our time to various and complex challenges posed to programming science. In this regard, we can understand the next points quite easily:

1. Automation: Usual repetitive tasks must be generalized, automated and implemented into a method.

2. Reusability: Any method must be ready to be incorporated into more larger methods.

3. Mutability: Any method must be modifiable to fit special needs.

4. Individuality: Any user shall have the opportunity to work according to his or her own style and needs.

5. Proficiency: the searching, calling or modification of a method must be millions of times easier than designing it ab initio.

6. Social proneness: one shall not get apart from the academic community or from the whole world. Java is the worldwide master in integration of programming workers.

7. Easiness of use: Java + NetBeans or NetBeans make of high level computing a realizable and rewarding task.

We will learn to install **Java**, to learn how to use **NetBeans or NetBeans** to run an existing Java program, and to manage the fundamental ingredients of programming that are directed to study models and evolution.

The basic items to simulate **evolution** are strings and numbers. That is so because DNA is a string of characters whose alphabet is A G T C, while many variables of phenotypic interest are naturally encoded in the form of numbers.

1.2 Installing Java

Java was created by Sun Microsystems in 1995. Sun Microsystems was acquired by Oracle in 2010. Sun distributed Java for free and Oracle has continued with the same tradition. Thanks a lot.

Java is not accessed by developers directly but through a friendly interface. The most popular interfaces are *Eclipse* of IBM and *NetBeans* of Oracle and both are distributed freely. The easiest to install is *NetBeans* because one has the possibility to download a bundle with *NetBeans* and the latest version of Java.

Java needs a folder for work. Create and name one, for instance *workspace3*. You can do this before installing your interface or during installation.

To download *NetBeans + Java* prompt Google with `netbeans jdk bundle` and look for a link to Oracle. Follow the link and arrive to the distribution, which by February/2017 is *JDK 8u111 with NetBeans 8.2*. Accept license agreement, choose your system (Windows, Linux or Mac), load and install the bundle on your desktop.

You might decide to suffer a bit by choosing a separate installation of Java and NetBeans. Both can be downloaded from Oracle ([76], 2015). Java comes with a lot of tools, which together form a JDK (Java Development Kit). Follow the links, match your system, say Windows or Linux, download packages and extract them to an appropriate folder.

Instead of `NetBeans` one can try `Eclipse Luna` (apt to run JavaFX, the modern Java form of making graphics and animations) and/or `IntelliJ IDEA's`. All IDEs (Integrated Development Environment) can be searched for by their names.

To launch NetBeans click once over an icon with a link to the execution file.

The web is full in tutorials about Java, NetBeans and Eclipse. Prompt Google with the key words that include your target + tutorial. In general, tutorials serve when they are combined with experimentation and suffering. For work in the long range, the complete guide to NetBeans would be very useful (NetBeans, [74] 2014).

The first task is to play with the new toy. If you want to be a great master, you must learn to enjoy playing permanently. By the way, give a try to your kids: most surely, they are looking for something really cool. Incite them to look here and there and try out whatever comes to the mind.

A very clear tutorial to create and run the first application or program in NetBeans is (NetBeans, [75] 2015). The same ideas can guide you to create your first program in NetBeans because both platforms are designed on the same basis and at last they are very similar. A series of videos to teach both NetBeans and Java are presented by (dextercowley, [24] 2015). Besides, NetBeans presents in its welcome window specific instructions to create a first program. Go to *Tutorials* → *Create a Hello world application*. Follow the instructions and try to create your first program.

If you fail to create your first program, use our methodology, which comes below.

4. *Guarantee.* *If after some two hours of work you cannot follow the instructions of the present chapter to run the first program, it is mandatory that you write me because you are guaranteed to have a happy start.*

Furthermore, we present many programs (evoljava, [28] 2017) that are guaranteed to work perfectly on diverse versions of Java SE and NetBeans. Make your guarantee effective by expressing your discomfort to

jose@evoljava.com

It is rather difficult to understand how a program is designed but we have made an effort to explain why each program works, and the reader is committed to learn how to modify them to create variations that fulfill slightly modified functions. So, the first step is to learn how to copy a program into NetBeans that it could be run.

1.3 Purposes and projects

In Java, before copying or designing a program, one must make clear what the purpose one has in mind is. Our present purpose could be just to study the material of present volume. This intention defines a project. Then, we must communicate to Java our intention.

The organization of files in Java is in the form of a tree and is as follows: a project has two branches: Source Packages and Libraries. Source packages include all packages devised by the User. These packages are populated with files that may content say, a document written in plain text or maybe Java code in whose case their names have *.java* as suffix. Libraries are provided by NetBeans.

5. *Opening a project*

To open a project follow the sequence of options beginning from the main menu: $File \rightarrow newproject \rightarrow Java \rightarrow Next$ A window will pop up for you with an invitation to fill the relevant data about your project. Give it a name, say *eJVol5*. To end, click *Finish*.

By default, a source package is automatically generated together with a seed program. These have the same name as the project but may differ with regard to upper vs lower case.

Now that we have a purpose, a project, and a seed program, let us copy and run a previously written program.

6. *Copying a program*

Programs are not typed except when they are created. Instead, they are transcribed by a copy-paste procedure, for which you have four possibilities:

1. To copy each program from the present pdf document. In this case, you must resolve a little problem: the copy operation also copies navigating signals of the document, headers and page numbers so, they must be identified and deleted. The identification is done automatically after the paste operation is complete: extraneous lines of text appear with a red mark at the right margin. They must be deleted.

2. To get the Latex source, which can be opened with a Latex editor. You can copy any program from it without suffer anyhow. Additionally, you are invited to use our Latex source to devise your own material for yourself or for your pupils. In general, feel free to do anything good for forming a healthy and strong **evolJava** community.

3. To get a file, in plain text, with all programs, from which you can copy the desired program immediately. To get the file, go to http://www.evoljava.com

4. If you are a teacher, you might prefer to import the accompanying zipped package with all programs in this volume. This option is not recommended for beginners: suffering is possibly the best teacher.

7. ***Exercise:***

Copy the program `E9 Console` below and replace the seed program with it. To that purpose:

1. Select the material you want to copy and use Ctrl + C to copy it to clipboard.
2. Select the seed program that was generated by NetBeans.
3. Apply Ctrl + V over the seed program to replace it with program A7.

1.4 Our first program

Program must obey a certain pattern or structure that is complexity dependent.

8. ***The simplest structure***

Our first programs will have the next structure:

```
//Program EO structure
//Same as Program   A0
//This is the structure of a program.
//The name in Java of this program is structure
public class structure
{
  //Every program has a main
  public static void main(String[] args)
   {
    //Here are the instructions
   }
}
```

When Java runs a program, it looks in first instance for the main method and begins its execution.

We have below a true program whose purpose is to write a message in the console. The **console** is a special window where NetBeans reports results and errors. If whatever cause hides the console, one can restore its visibility in any one of the two following options:

a) In the main menu follow the path: $Windows \rightarrow ShowView \rightarrow Console$.

b) Click Alt+Shift + Q, then click C.

9. ***Printing to the console.*** *The next code contains a program, whose purpose is to print a message to the console.*

```
//Program E8 Console
//Same as Program   A7
//This is my first program.
//The name of this class is Console because
//it prints a message to the Console.
package ejvol5p;

public class Console {

    //The next is the main method
    public static void main(String[] args) {

        // This is a comment.
        // Comments help to understand programs.
        // The next line contains an instruction
        // that prints a message to the Console:
        System.out.println("I am starting up! ");

    }
}//End of Program E8 Console
```

We use in this program the method **System.out.println**. This method prints a message to the console and posits the console cursor at the beginning of the next line. One can use 'print' to write something but remaining in the same line.

10. ***Exercise*** *Observe in the anterior code that parentheses and keys always come by pairs. Java allows placing each parenthesis with freedom. Our directive is to enable a quick visualization of blocks. Point to the lines that contain a comment and to those that contain code properly, i.e., instructions that Java must decipher and execute. What distinguishes code from commentaries? Which is the function of the semicolon? Exercises are part of the text and results might be used in the sequel.* *Answer*

11. ***Running our program***

To copy and run this class, go to the Run menu. We must learn before everything else how to save and build the project and then run the file. To save: click over the Save All icon. To build the project: go to the menu Run + Build Project. To run the file: Run $->$ File.

We shall see the output of our first program that writes in the console the phrase: *I am starting up!*

Let us face to our first exercise in modifying a program. We are artificially evolving software. Java interfaces are very gentle with the developer and they mark errors in red. To **correct an error**, drag the cursor to the error and wait a moment that NetBeans pops up a message with the explanation of the nature of the error and possibly with a suggestion to correct it. Errors are also marked at the navigating vertical bars that surround the editor window. If a yellow light also appears, right click on it to read more suggestions for error correction. A suggestion can be executed at once by double clicking on it.

12. ***Exercise*** *Modify our first program in such a way as to write in the console your name and birthday. Every program goes on a separated file. So, you will need to know how to append a file to a project:*

1. *Click on* File $->$ New File *and a window will pop up.*

2. *Select your project. Verify that you are working with Java and that you will generate a Java Class. Click Next. In the emerging window fill in the name of the new class:* Biography. *The editor window will filled in the seed of a new program. In its upper bar, one can see the name that we gave to the class, say* Biography.java.

3. *You can now modify your new class at pleasure.*

Cut way: Under the Projects tab, look for your project. Expand its tree to find Source packages, *and the package* ejvol5p. *Right click on the package* ejvol5p. *A contextual menu will pop up and choose* New $\rightarrow$ Java Class. *Fill in the name and then click on* Finish. *Answer*

13. ***License.*** *Dear Reader: as part of the exercises, you must modify the programs. So, you have the* ***license*** *to use all programs written in this book in whatever form you want even to get rich and very specially to modify them as you are asked in the exercises. This freedom might help the formation of a strong EvolJava community.*

1.5 Bugs and debugging

The most immediate reality of every developer, no matter how good and skilled he or she might be, is the great quantity of errors that he or she commits when composing a program or modifying a piece of software.

14. ***A program is known by its fruits***

There are two types of errors: those that hinder compilation and those that hinder function. **Compilation** is the process of translating a program that is written in Java into machine language, which is composed of elementary instructions that can be executed by the computing hardware. In the case of Java, the computing hardware is really a simulated one that is called the **JVM (Java Virtual Machine)** and its machine code is name **bytecode**. This machine is the same for every platform, Windows, Linux or Mac. There is moreover another translator that is platform dependent and that receives the bytecode and outputs the machine language according to the platform.

A compilation error or **bug** happens when the program is not accepted by the compiler. NetBeans is very gentle with the developer and it marks errors in red. To **correct an error**, drag the cursor to the error and wait a moment while Java pops up a message with the explanation of the nature of the error and possibly with a suggestion to correct it. Errors are also marked at the navigating vertical bars that surround the editor window. If in the left bar a red X appears, drag the cursor to it and an explicative message will appear. If a yellow light also appears, right click on it to read more suggestions for error correction. A suggestion can be executed at once by double clicking on it.

Sometimes a program is not accepted but it has no errors. This happens when one follows the procedure to run a program: *right clicking* $\rightarrow$ *Run as* but nothing is given to choose. In this case one has a grammatical error, say, the main method is not declared as *static*.

The most difficult errors happen when the program is accepted for compilation, runs and produces results but these make no sense in regard with the intended function of the program. To correct these type of errors, one must keep in mind that software is in general highly informative and so to know its output there is only one way, to run the program in the computer or in the mind, to see what it does. In general, mental calculations are not reliable and so one must resort to real execution: **software is known by its fruits**. This means that one must force the program to write out intermediate results to follow detailed calculations. To do this, one inserts lines of code with instructions to write over the console interesting variables and their names. Next, one compares output with correctly expected results and can decide whether or not the error in design has been committed before of after the line under scrutiny.

NetBeans has special facilities for **debugging**, which is the process of cleaning bugs. To this aim, it is good to activate before everything else line numbering: one must right click over the left vertical bar that surround the editor window. A menu will pop up and over it, one checks the box corresponding to *Show line numbers*. Next one must target a given line of the project as compilation breaking point: right click over the left vertical bar of the editor window and choose *Toggle Breakpoint*. Next, follow the menus *Window* $\rightarrow$ *Open perspective* $\rightarrow$ *Java Browsing*. Next one chooses *Run* $\rightarrow$ *Debug as* $\rightarrow$ *Java Application*. The program will run until the breakpoint and a table will be displayed with the variables and their names when the execution runs at this precise place. Additionally, helping menus get activated: they are located in the Menu *Run* $\rightarrow$ *step in, step over*. Play a bit with them to see what happens.

Reality seems to recurrently show that apart from never stop suffering there is nothing else to debugging. Nevertheless, some people consider that debugging is a science on their own, so, they develop black lists with statistics and special strategies for catching bugs and things like that. Actually, there are programing languages and styles of developing software that are error prone. We will discuss below some strategies directed to attack such weaknesses, which can not be completely abolished because bug free software is technically an impossibility. The reason is that software to be bug free must be tested for every possible case and this can take longer than allowed time.

My first black list of bugs:

1. Terminal semicolon ";" is lacked.
2. A variable has been erroneously typed.
3. Keys or parentheses do not go by pairs.
4. The type of a variable has been violated, say, variable `i` has been declared as int (integer 0, ± 1, ± 2, ...) but it is instantiated as `int i = 3.56;`

15. *Self awareness*

This text has been provided with many, many exercises that the reader could grow enough personal experience to agree else disagree that *there is no software development without huge quantities of bugs, whose corrections generate more bugs*. Besides, software development encloses shifting on the fly of strategies and/or goals, events that are rich sources of bugs.

16. *High levels of variability*

The correction of bugs induce creative challenges that might be solved in one of many possible ways. By this reason, the correction of bugs is translated into huge variability of terminated works. The Reader can verify this by comparing his or her answers to exercises with those of the Author: they always will be different.

17. *Keeping an eye on evolution*

Our aim is to understand evolution. A fundamental presupposition is that DNA or evolution have no magic at all, i.e., the evolution that happens in living beings obeys the very same laws that rule evolution in whatever possible world. To begin with, *the genome is software* since it comprise verbal instructions for the synthesis of peptide chains, and *evolution is a software developer*. Therefore, learning to program in Java is a very suitable laboratory experience to grasp simple and powerful laws of software development and of evolution.

1.6 How to import a project

NetBeans offers the opportunity to effortlessly import complete projects with all its files. All one needs is to turn NetBeans with its already created workspace and follow the next procedure.

18. *The eJVol5P project*

Let us import into the workspace the project *eJVol5P* with all the programs of this volume. All its files are ready to be run. To that aim:

1. Download the *eJVol5P.zip* file associated to Vol 5 and keep it in a suitable folder.
2. Over NetBeans follow the menus: *File* → *Import* → *General* → *Existing projects into Workspace* → *Select Archive File*. In the new dialog, browse for the folder where you put the file. Click over the file and next on *Finish*. This operations create a new project, *eJVol5P*, with all the source files of Vol V.
3. To see your recently created project, go to the menu *Window* → *show view* → *Package explorer*. On the corresponding window, look for your project and expand its tree to find source files. After that, all Java files will appear below *ejvol5p*.
4. To run a specific program, select it and next pulse F3 to show it on the *editor window*. Left click on the editor window and with right clicking follow the *run as* menu. Next, choose *Java application*.
5. To see the output of the program, go to *Window* → *Show View* → *Console*.

These instructions are guaranteed to function. Otherwise, write me to negotiate a truly functional set of instructions.

1.7 Review

We have learned how to acquire and install Java and its companion NetBeans and we have created our first project to which we appended our first program. After going over the exercises of this volume, the Reader will gain sufficient experience to take as own our crude and inescapable reality: there is no software without previous fierce debugging or cleaning of errors. This experience is strongly formative with a direct relation with biology because the genome is our natural example of software: it contains verbal instructions, a program, that must be decoded by t-RNAs and executed by the ribosomes, which play the role of central processing units of a natural computer. Given that the genome is software, what or who did develop it? Dear Reader, after working this volume, you are invited to rapidly work Volume I and next to work out Vol II and III, whose aim is to transform this terrible question in a part of science. A perspective over those works might be gained in the last chapter of this volume.

Chapter 2

Integer numbers

$.... -3, -2, -1, 0, 1, 2, 3....$

19. ***Purpose.*** *Length is measured in centimeters and memory is measured in* ***bits****. In spite of the tremendous technological developments, memory continues to be a scare resource. So, to observe prudence in its management is an elementary duty. That is why a variable must be declared as belonging to a type, whose first information is to say how many bits must be allotted to record an instance of that variable. Our first type is* ***int****, our first version of integers. Dealing with them, we get acquainted with the control instructions* ***for*** *and* ***if*** *and with the structuring of programs.*

2.1 Elementary operations

20. ***Counting begins with zero.*** *While we are used to begin counting with one, Java begins with zero. The numbers used for that purpose in Java are the* ***non negative integer numbers:*** *0, 1, 2, 3, ... Nevertheless, integers can be positive or negative and the sign is encoded just as in English. Because of memory management, integers must be bounded somehow. Later, we will investigate the upper limit. In the next program, we learn to add two numbers.*

A variable must come in a program with the declaration or specification of its type. This must be done just once. In the next program, we learn how to declare an **integer variable** and how to **add** two numbers. To write on the console, one uses the instructions

```
System.out.println(" The sum of x =" + x + " and ");
System.out.println( " y = " + y + " is " + Result );
```

The syntax reads: *print to the console(this and this)* but instead of *and* one uses a +. The text among quotations will be printed verbatim to the console but variables are replaced by their values. The program follows:

```
//Program E20 IntAddition
//Same as Program A13
//This program introduces integers
//and their addition.
package ejvol5p;

public class IntAddition {

    public static void main(String[] args) {
        //Declaration and initialization
        // of variables as integers.
        int x = 2;
        int y = 3;
        //Computation
        int result = x + y; //result is declared as integer
        // Report
        System.out.println(" The sum of x = " + x + " and ");
        System.out.println(" y = " + y + " is " + result);
    }
}//End of Program E20 IntAddition
```

21. ***Exercise*** *Copy the code to the clipboard, next open in any project of NetBeans a class, name it as intAddition. Next replace the automatic generated text by that kept in the clipboard. Run the code. The output appears in the console. Verify the result.*

22. ***Exercise*** *Modify the previous program that the output appears in one line instead of two.* Answer

23. ***STOPPING A PROGRAM.*** *To stop the executing machine when a program is running, click on the terminator icon, in live red, at the upper bar of the console. If the red is very pale, the activated program has been stopped. The next is a secret that you must keep in mind: sometimes an undesirable execution has been started and it may run for hours slowing down the velocity of the CPU (central processing unit). Thus, if the machine gets too slow, there are possibly some programs that run in background because you forgave of them. To make sure that all launches have been canceled follow and examine the menu* `Run -> Stop Built/Run`.

24. ***Exercise*** *Modify the previous program to make a multiplication of two numbers. To* ***multiply*** *apply the* ***star*** * operator as in "a*b". Indicate also a* ***division*** *(use /) and a* ***subtraction*** *(use -). Warning: 6 /4 = 1.5, but when one declares the result to be integer, the division is ONE. The usual division (6 /4 = 1.5) will be recovered in the next chapter.* Answer

2.2 Structured programming

A structured program is a program that is composed of various subunits, modules, subroutines, functions or methods, each of which fills in a determined function, which is known to perfectly work. Structuring a program makes things easier but also has a price in relation with syntax troubles. **Structured programming** was the great invention of the 1970s: you can build great structures by recursion to tested building blocks. This style is the implementation of a simple but very powerful directive to try to break down complexity: divide, name and conquer.

25. ***Definition.*** *A* ***method*** *is a block of Java code that can function on itself. It has a name, requires an input, an output and a body. The input is the data that is processed by the method with their type, say,* `int`*. If one declares the input, we speak of a* ***parametrized method*** *otherwise we have a not parametrized, free method with a void input. The set of parameters are also called the* ***argument*** *of the method. The body is the code that conforms the method. The output is the product of the code of the body, whose type must be specified.*

In the next declaration

```
//Parametrized method
public static int sum(int x, int y)
{
 return  x+y;
}
```

we have a method whose name is `sum`. The input consists in two numbers `x` and `y`. Both numbers are of type `int` (integer). The method produces an output of type int. This is declared as `int sum(...)`. This means that we add two numbers that are integers and whose result is also integer. The reserved word `static` refers to a mode of memory assignation and calling protocols: almost all methods of this volume are static. The reserved word `public` says that this method can be used by whatever other program or class in the Java project to which the code belongs in. If one wants to avoid possible interferences, one may declare a method as `private` and in that way it will be used only by the program that contains it. The body of the method is embraced in within keys {}. The output produced by the body of the program is marked by the reserved word `return`.

It may happen that a method produces so a complex output that it is cheaper to go without specifying it. In that case we declare the output as `void` and the body of the method works on globally defined variables. Let us explain this. In Java, a variable can live just when it is needed or it can be programmed to live during the whole execution of the program. The art of programming makes intelligent use of those two options to avoid bugs due to interference of domestic problems with global ones. A **globally defined variable** is one that

lives forever and is declared as `static` at the very beginning of the class and before methods. Globally defined variables can be invoked from whatever method, in the way as `sum` is called in the next example. In this method, variables x and y are **local** or short lived: they live just when the method is called. The example with void output is

```
//Method with void output.
public static void sum(int x, int y)
 {
  //variable sum is globally defined.
  //x and y are locally, short lived variables.
  sum =   x+y;
 }
```

Example with void input:

```
//Not parametrized method.
//The input consist of globablly defined variables
//The output is of type int
public static int    sum()
 {
  //variables   x and y are globally defined.
  return    x+y;
 }
```

Example with void input and output:

```
//No input, no output, only operations
//over globally defined variables
public static    void sum()
 {
  //variables sum, x and y are globally defined.
  sum =   x+y;
 }
```

The next code shows how everything is glued to form a program.

26. ***Code that shows a structured program with one auxiliary parametrized method.***

```
//Program E26 Structured
//Same as Program A18
//Structuring a program

package ejvol5p;

public class Structured {
```

```
//Variable w is globally defined
//but private to the present program:
//it cannot be accessed from other programs
//in this project.
private static int w;

// The next is a parametrized   method.
// Its function is to sum two numbers
// of type int, one is s, the other is t.
// The output of the method is declared as int.
// It is computed after
// the reserved word "return".
// The declaration of a method or a class
// does not terminate with semicolon.
private static int sum(int s, int t) {
    //s and t are local variables
    //they exist only while the method is
    //called and executed.
    System.out.println("Method sum(...) in operation:");
    System.out.println("s = " + s);
    System.out.println("t = " + t);
    //sum is a locally defined variable
    //only for domestic use
    int sum = s + t;
    System.out.println("sum = " + sum);
    //Invocation of a globally defined variable
    System.out.println("Globally defined w = " + w);
    return s + t;
}

//This is the main method.
//This is the head of the class.
public static void main(String[] args) {
    //Declaration and initialization
    // of variables:
    //x and y are locally defined
    int x = 2;
    int y = 3;
    //Initialization of a globally defined variable
    w = 7;
    //Invocation of a method
    int z = sum(x, y);
    // Report to the console
    System.out.println("\nMethod main(...)   in operation;");
    System.out.println("The sum of x = " + x + " and ");
```

```
        System.out.println("y = " + y + " is " + z);
    }
}//End of Program E26 Structured
```

27. ***Exercise*** *Add the previous class to your project and run it. Verbalize its functioning. Indicate the relation between the order of dominance among methods and their position along the text. Remember that a class must be saved in a separate file and the names of the file and of the class must coincide, an operation that is case sensitive.* *Answer*

28. ***Exercise*** *Change the relative position along the text of the methods of the previous program and try to run it to see what happens.*Answer

29. ***A style that helps.*** *Java does not perceive any change in the relative positions along the text of the different methods. So, one can posit them in whatever position one likes. Nevertheless, programming is a difficult task and it helps to keep track of the hierarchy of the methods by choosing a specific style, say, the more dominant is a method the lower is its position in the text. Feel free to develop your own style.*

30. ***A program can be structured as much as desired.*** *Let us see how the previous program is divided to include two auxiliary methods:*

```
//Program E30 TwoMethods
//Same as Program A22
//Structuring a program
package ejvol5p;

public class TwoMethods {
    //This is a method.
    // Its function is to sum two numbers
    // of type int, one is x, the other is y.
    //The result is declared as int.
    //The output of the method appears after
    // the reserved word "return".

    public static int sum(int x, int y) {
        return x + y;
    }

    //This method reports to the console
    // some information about x,y,z.
    // It does not report a result to the main
    // method, so its output is void.
```

```
    public static void report(int x, int y, int z) {
        System.out.print(" The sum of x = " + x + " and ");
        System.out.println(" y = " + y + " is " + z);
    }

    //This is the main method.
    //This is the head of the class.
    public static void main(String[] args) {
        //Declaration and initialization
        // of variables
        int x = 2;
        int y = 3;
        //Invocation of a method
        int z = sum(x, y);
        // Report to the console
        report(x, y, z);
    }
}//End of Program E30 TwoMethods
```

31. ***The degree of structuring depends on the style of the programmer. Let us consider two styles. First: a method and its commentaries shall not surpass the capacity of the editor window. Second: a method is worth dividing when the divisions are natural according to their function; otherwise division generates more work at the time of development and more noise when the need of revision arises.***

32. ***Structuring of the genome.***

The information in the genome is also structured. A nice proposal to understand the style of structuring is this: if a piece of code can be reused to complement another part of the genome, it is worth being separated. The separating symbols are known as **introns** and the useful code as **exons**. This is so because the DNA is transcribed to RNA and the introns are deleted from the RNA into the nucleus and so they remain internal to the nucleus, while the exons can leave the nucleus to be translated by ribosomes. Apart from this structuring, the cell also has spatial structuring in diverse **organelles**.

There was a time in programming science when programming was not structured. A program consisted just of code that was deciphered and executed line by line one after another. By the same token, there was a time in the history of the Earth when the genome was not structured and there were no organelles. In that time, life was dominated by prokaryotes. Structured information appeared with eukaryotes, cells that contained diverse organelles, a nucleus separated by a membrane and structured genomes.

One may suppose that the appearing of eukaryotes occurred with the intrusion of certain prokaryote inside another. The intruding bacterium could have evolved into a nucleus with a membrane, while the recipient could have lose

its nucleus but gained the ability to host another parasitic cells that became organelles.

One can plague science with suppositions of this kind. They look attractive but lack any quantitative prediction and, more to the point, are naive: eukaryotes are important more because of their structuring programming than by their spatial compartmentalization.

Which is the mystery enclosed by structured programming?

Be it in reference to natural or to artificial software, structured programming amounts to a revolution that boosts the combinatorial power of possible changes: without structuring, one can combine letters but with structuring programming one can combine modules, methods or exons.

One combines letters to create exons and genes, one combines genes to create genomes. Thus, eukaryotes differ more by the form and tempo of regulation of their genes than by genes themselves. But prokaryotes may preferentially differ by their genes, whose regulation is quite simple (but very effective).

Now, we are forced to ask: Was the appearing of structuring of the genome triggered by bacteria parasitism? To answer this question, it is instructive to look at the evolution of software engineering.

Structuring programming appeared in software engineering because it was created to fill in three needs. First, to try to kill **complexity** using a strategy of divide + name + conquer. Second: to get profit of modular design. This type of design is the one used in modern times by students: they copy from Internet one paragraph from here and other from there, concatenate them together and go to the teacher with a wonderful work. In software engineering, modular design is called **software reusability**, a terminology that is gaining popularity in evolutionary biology. Third reason: to shorten compilation times. With the advent of high level programming languages, the compilation or translation to machine language was a highly consuming time activity because debugging forced the developer to compile a program many times. But with modular design, small pieces of code could be perfectly assembled and compiled once and forever. In this way, the terrible load of debugging + compiling was lightened at least a bit.

The first reason is not applicable to nature, because it lacks any notion of complexity. The second reason possibly may be applied to nature if one shows that structuring the genome is an evolutionary stable strategy, i.e. that it is competitive in evolutionary terms. If that were the case, the permanence until our days of an already extant structuring of the genome could be understood. But this says nothing about its appearing. This problem is very interesting because the evolutionary competitiveness of structured genomes is by no means clear:

Certainly, a modular approach to problem solving can do more with less and the regulation capacity also grows. But the cost of control also grows and, because of length considerations, it is by far much more expensive to synthesize the DNA of a eukaryote than that of a prokaryote.

Can we, the EvolJava community, enrich these wonderful discussions with simulations that could lead to brilliant experiments? Dear Reader: our respon-

sibility in this regard is very high because we indeed have the possibility to illuminate somehow these terrible problems.

2.3 The if control structure

33. ***Exercise*** *Develop else reuse a previous program to divide two numbers and to study division by zero. Verify that Java refuses to execute a division by zero. This must be so, because a division by zero leads to contradictions. Prove this.* *Answer*

34. ***Using conditional execution.*** *To prevent the spoiling of a program by a division by zero, one can use an* ***if-(then) -else*** *control condition.*

Let us suppose that we have the next pseudo-code:

if `i` equals 3 then assign to `i` the value 4 otherwise assign 5 to it.

The corresponding Java encoding is the following:

```
if (i==3) i = 4;
else i=5;
```

Dear Reader, please, notice and remember once and forever that the `if structure` tests a logical condition and if it refers to equality between two numbers, **the sign of equality comes double**! By contrast, an assignation comes with just one =.

Let us see how this control structure is used to make a program robust against a division by zero:

```
//Program E34 Robust
//Same as program A26
//This program divides two integers.
//This program is Robust against
//division by zero.

package ejvol5p;

public class Robust {

    public static void main(String[] args) {
        //Declaration and initialization
        // of variables
        int x = 7; //Assignment: the symbol = appears once.
        int y = 3;
        //Computation
```

```
        if (y == 0) //Logical testing: double ==
        {
            System.out.print("The division by zero is forbidden");
        } else {
            // Report
            System.out.print(" The integer division of x =" + x);
            System.out.println(" by  y = " + y + " is " + x / y);
        }
    }
}//End of Program E34 Robust
```

35. ***Exercise*** *Copy the code to your project and run it. Test it for various values of the variable y including zero. What do you observe?* *Answer*

36. ***Robustness as an impossibility.*** *To design software with guaranteed robustness against all the possible inputs given by the client is impossible in practical terms. That is why companies produce beta versions that are released for free in the hope that users could report the most usual problems. The same happens in biology: death + reproduction +* ***evolution*** *could be understood as a solution to the practical impossibility of devising perfect robustness against the milieu and self-destructing operations. Can this intrigue be made into a theory about the extinction of species?*

37. ***Challenge*** *Develop a structured program that makes the following:*

1. *Writes your name, birth date, last employment and the date.*
2. *Calculates your age in years, in weeks and in days.*
3. *If you are younger than 22, write your favorite sport. If you are older than 22, write your favorite team.*

2.4 Review

We have become acquainted with integers and their fundamental operations: addition, multiplication, division and subtraction. We learned also about structured programming and discovered that our natural example of software, the genome, also comes structured. How did this come into being? By design? By evolution? Can we help ourselves with Java simulations to throw light on these questions? Of course we can and we take it as a challenge.

Chapter 3

Using Excel, Calc and Gnumeric

`Java + worksheets`

38. ***Purpose.*** *We learn here how to combine the calculating power of Java with the graphical facility of* ***Excel****. Instead of Excel one can use* ***Calc of LibreOffice*** *or* ***Gnumeric****, which are distributed for free. Instead of Windows, one also can work with Solaris, or UNIX or Linux. Excel, Calc and Gnumeric allow the possibility of working with huge volumes of data, a reason that forces us to learn the* `for-control` *structure of Java.*

3.1 The String Type

Numeric activities are served with int and related types. What do we have for texts? We have strings, learned here, and chars, found elsewhere.

39. ***Declaration of a string***

We know how to make a report to the console and for that purpose we use a printing instruction. It is actually a call to a method that already is implemented in Java:

```
System.out.println();
```

A method has an argument and an output. The argument or input of the print instruction must belong to the type **String**, which is the Java name for text, which must be declared among quotations:

```
String s = "This is a string";
System.out.println(s);
```

40. ***The tabulator***

The output of the program just below contains the first 7 numbers and their corresponding squares. The idea is that we want to report the output graphically with the help of Excel, so we must make sure that the output is correctly written in the format of Excel, which also fits that of LibreOffice and of Gnumeric. The problem is that our output has two columns and we must guarantee that when copied to the clipboard and then pasted into a sheet, one shall see two columns instead of one. Inserting a **tab** signal among the entries of different columns will do it. The instruction responsible for that operation contains the encoding of a tab as follows:

```
"\t"
```

That symbol could appear as in the next instruction which prints the number 1 followed by a tab and next by the number 5:

```
System.out.println("1 \t 5");
```

41. ***Concatenation***

To copy one string after another, the symbol + shall be used, that represents the **operator of concatenation**. So, the last three instructions of the next code are equivalent:

```
String tab = "\t";
System.out.println("7" + tab + "49 ");
System.out.println("7 \t  49 ");
System.out.printf("7" ,  "\t" , "49 \n ");
```

All these instructions write down string 7, next a tab space is left and then the string 49 is written. Java distinguishes between the string "7" and the number 7. Nevertheless, both are written in the console as number 7 and so both can be pasted into a sheet of Excel, Calc or Gnumeric as numbers.

42. ***A concrete example that uses tabulators.***

```
//Program E42 Tabulator
//How to use the Tabulator
//to print columns of numbers
//in a format appropriate for
//Excel,  Calc of LibreOffice or Gnumeric.

package ejvol5p;

public class Tabulator {
```

```
    public static void main(String[] args) {
        // Here a Tabulator sign is defined.
        String tab = "\t";

        // Report to the console
        System.out.println("1" + tab + "1");
        System.out.println("2" + tab + "4");
        System.out.println("3" + tab + "9");
        System.out.println("4" + tab + "16");
        System.out.println("5" + tab + "25");
        System.out.println("6" + tab + "36");
        System.out.println("7" + tab + "49");
    }
}//End of Program E42 Tabulator
```

43. ***Exercise*** *Run the program, copy the output to the clipboard, paste it to a sheet of Excel and invoke over these data the menu Insert → graphics → XY(Dispersion). You shall see the graphic of $y = x^2$, which is a parabola. If you work with Calc of LibreOffice , please follow the next instructions: Run the program, copy the output to the clipboard, paste it to a sheet of Calc with the right-click instruction PasteSpecial → unformatted → unicode + tab → OK. To make a graphic: select the 2 columns of data, invoke Insert → char → XY(Scatter) → lines only → create. You shall see the graphic of $y = x^2$.*

44. ***Challenge.*** *Modify the previous program to make a graphic in Excel of $y = x^3$.*

3.2 Loops

Since we are expected to work with a large quantity of numbers, it is a good occasion to see the use of the **for** instruction in Java to execute loops.

45. ***The `for-control` structure*** *has the following pseudo-syntax:*

for(beginning with this number; while this condition holds; increment the counting index by one after each loop)
do this and that.

As an instance, the next block

```
//Program E46 Loop
//Same as Program A34
//Introduction of the for structure.
//The "for" instruction causes a Loop
//to deal with repetitive tasks.
```

```
package ejvol5p;

public class Loop {

    public static void main(String[] args) {
        String tab = "\t";
        for (int i = 1; i < 7; i++) {
            System.out.println("i = " + i + tab
                    + "i*i = " + i * i);
        }
    }
}//End of Program E46 Loop
```

begins the counting index with 1, prints 1 and its square, increments the counting index in one, gets 2, compares it with 7, decides that it must continue looping, prints a 2 with its square and so on until the square of six.

In this piece of code we find for the first time the symbol $i++$. This is part of the new syntax, (after C++), where we have the next instructions:

$i+=1$: increment i by one.

$++i$: increment i by one, then use the new value of i in the expression, where i is located.

$i++$: use the current value of i in the expression where i is posited and then increment the value of i by one.

As one can guess, these symbols and many other facilities of Java allow great proficiency in software development. That is not our fundamental aim. Our purpose is to learn to study evolution and every kind of related problems. Thus, we follow the next directives: a) Learn the maximum of biology and evolution with a maximum reuse of elementary things in programming. b) Learn something only when it pays a lot. It is unfortunate, nevertheless, to witness that the minimum to be learned actually is a huge! Why must this be so? It is a question that also concerns evolution, since **evolution** is before everything else the creation of software!

We will use the notation

$i = i + a$

that means: add a to the current value of i.

This old fashioned notation is both clearly and effective. But in a for loop, we will use $i++$ because it means there exactly the same as $++i$ and $i = i+1$. So, one must not care about possible confusions.

46. ***The code that contains a for loop follows:***

```
//Program E46 loop
```

```
//Same as Program A34
//Introduction of the for structure.
//The "for" instruction causes a loop
//to deal with repetitive tasks.
public class loop
{
 public static void main(String[] args)
  {
   String tab = "\t";
   for(int i = 1; i<7; i++)
   {
     System.out.println("i = " + i + tab
                + "i*i = " + i*i);
   }
  }
}
```

The letter i is a variable that controls the loop that repeats the operation of printing into the console the squares of the first 6 numbers.

47. ***Exercise*** *Run the program and verbalize the function of each line.*

48. ***Exercise*** *Modify the previous program to produce a suitable output to make a graphic in Excel (or Calc or Gnumeric).* *Answer*

49. ***Exercise*** *Because one always commit many bugs or errors when composing a program, NetBeans offers an inbuilt facility to help fixing them. To know how it functions, run the previous program in* ***debugging form****. To that aim:*

1. *In the editor window, in the most left column, right-click in front of the instruction that contains the for loop and choose Toggle Breakpoint. A blue circle will appear in the left column.*
2. *In the main menu, choose: Run → Debug As → Java Application*
3. *Make sure that the console is visible: in the main menu, choose window → view → Console.*
4. *Press F5 some 7 times and observe that the output appears step by step.*
5. *Observe that F5 is one among various possibilities of 'debugging' this program. Find others and play with them.*

50. ***Exercise*** *In the previous program instead of the instruction*

```
for(int i = 1; i<7; i++)
```

use the next one

```
for(int i = 1; i<=7; i++)
```

Check out the effect of the change. Answer

51. **Exercise** *Modify the previous program to having it calculate the squares but beginning from zero and going until 10 included. Show the corresponding graphic.* Answer

52. **Exercise** *Modify the previous program to make it calculate the cubes but beginning from zero and going until 10 included. Show the corresponding graphic.* Answer

53. **Challenge** *The division of i by 2 is indicated as i/2. Try and fail to modify the previous program to make it calculate the half of the first 15 integer numbers. The reason of your failure is that if the output of a division of an integer by two is declared as integer, it is an operation that rounds the result. To conserve decimals fractions, we need to use another type of numbers that appear in the next chapter.*

54. ***Integers are finite***

Integers go in ancient programming languages from approximately -32000 to + 32000, one by one.

To make an estimation of the allowed range of the integers in Java, let us use the next program. We will know that the limit is surpassed because when adding a positive number to another positive number, we get a negative one. Our program reports the first N multiples of 1000. We begin with $N = 10$.

```
//Program E54 HowMany
//Program A40
//The upper bound of integers in Java

package ejvol5p;


public class HowMany {

    public static void main(String[] args) {
        String tab = "\t";
        int N = 1000000;
        for (int i = 1; i <= N; i++) {
            System.out.println(i + tab + 1000 * i);
        }
```

```
    }
}//End of Program E54 HowMany
```

55. ***Exercise*** *Run the program and play with it. Gradually increase the value of N to see when the result bends to negative values. Verify that 10000000000 is an int. (The actual capacity depends on the number of bits accepted by the microprocessor).*

56. ***Challenge.*** *If one doubts that an int will not suffice, one can use the type* ***long****, which is more generous and functions just as the type int, the only difference is that they admit larger numbers. Investigate how to declare them, how to operate with them and how large can they be. Hint: proceed by instinct else punch the web with the prompt* `java long`*.*

3.3 Review

We have managed to connect the output of Java, when it is of the int type, to Excel, Calc or Gnumeric. Now, we enjoy the possibility to make graphics of our outputs. In prevision of high volumes of data, we have learned the use of the `for-control` structure of Java and of some of its variations. As an application, we found that for Java over 64 bits the number 10000000000 is admitted as integer.

Chapter 4

Decimal numbers

```
-17.234
```

57. ***Purpose*** *We learn to manage decimal numbers in Java.*

A decimal number is a string of digits among which a comma or a period seems to float. That is why they are encoded as belonging in the **float type**. But Java also has the type *double*, which is enough for all our needs.

4.1 Type double

The developers of Java were careful enough to make this language both accessible to the beginner and perfect for experts. A price that was paid was redundancy: some types have two or more versions, one for an introduction to Java and another for heady duty. So, we have the type *int* which is used to encode numbers of ordinary life while the expert version is *Integer* (research). In regard with decimal numbers, the type *double* is suitable for most tasks - in any case, there are more sophisticated versions that we will encounter below.

58. ***Declaration and operations of type double***

Java allows the declaration of a decimal number as **double**, which means that we deal with a number with many decimal significant figures. To operate numbers of type *double*, we follow exactly the same syntax as for those of type *int*. How that works is seen in the following program:

```
//Program E58 Decimal
//Same as Program A43
//The type double encodes Decimal numbers.

package ejvol5p;

public class Decimal {
```

```
    public static void main(String[] args) {
        String tab = "\t";
        for (int i = 1; i <= 15; i++) {
            double r = i;
            System.out.println(i + tab + r / 2);
        }
    }
}//End of Program E58 Decimal
```

59. ***Exercise*** *Run the previous program. Try to graphic its output in Excel or Calc. Explain.* *Answer*

60. ***Exercise*** *Modify the previous program to test the capacity of the console where the output is presented. Try with 100 as an upper limit of the counting index i. Use next 200,.. .., 1000, 2000,.. .. Do not let the console to misguide you: you must be able to see as the first as the last line without mutilations.* *Answer*

4.2 The for structure revisited

The `for` structure has in Java many possible variants.

61. ***A for with integer increment:***

```
//Program E61 Seven
//Same as Program A46
//Variation of the for statement:
//incrementing value can take on int values.

package ejvol5p;

public class Seven {

    public static void main(String[] args) {
        String tab = "\t";
        for (int i = 0; i <= 100; i = i + 7) {
            System.out.println(i + tab + 1000 * i);
        }
    }
}//End of Program E61 Seven
```

62. ***Exercise*** *Run the code and find the changes in the output. Next, find the causal changes in the code.* *Answer*

63. ***Fractional increments***

In the next program we see a variation of the for structure in which the index variables admit decimal increments:

```
//Program E63 Fractional
//Program A48
//Variation of the for statement
//with Fractional increments.

package ejvol5p;

public class Fractional {

    public static void main(String[] args) {
        String tab = "\t";
        // The index is declared as "double",
        // so it accepts a decimal  expression
        // in its increment.
        for (double i = 1; i <= 2.01; i = i + 0.01) {
            System.out.println(i + tab + 2 * i);
        }
    }
}//End of Program E63 Fractional
```

64. ***Exercise*** *Run the program, study its output and find the function of each instruction. Verify your theories by composing a program to calculate the squares of the real numbers beginning with 3 and ending with 4 with step 0.05. Test your result with a graphic in Calc: you must find a parable that smoothly interpolates between the square of 3 and that of 4.* *Answer*

4.3 Review

We have gotten acquainted with the treatment of Java to decimal numbers when they are declared of type *double*. In combination with a `for-control` structure, our new type allows for nice possibilities. It is also found that this type loses precision beyond 14 significant figures.

Chapter 5

Arrays

Tables are called arrays.

65. ***Purpose.*** *In programming languages, tables are called* ***arrays****. Tables can have one, two, three or more columns and one, two or more rows. A table with one column is unidimensional. A table with various columns are bidimensional. Multidimensional arrays are also enabled by Java. We become acquainted in this chapter with one and two-dimensional arrays. A numeric array with one column may represent numerical data and so we learn to calculate their mean and variance.*

5.1 Vectors

If the table has just one column and contains numbers and nothing else, we say that we have a **vector**.

66. ***Squares.*** *In the following program we use a vector to posit the squares of the first 10 numbers (unsigned integer numbers in Java begin with zero).*

The **declaration of an array** can be made in one step but one might prefer two as in the next program:

```
//Program E66 ArrayOne
//Same as  Program A81
//We introduce one-dimensional arrays.

package ejvol5p;

public class ArrayOne {

    public static void main(String args[]) {
        // Declaration of an array with integer entries
```

```
        int data[];  // data is the name.

        // Assignment of the capacity of the array:
        // ten entries, beginning from zero.
        // data[i] can be defined for i from zero to 9
        // data[10] is not defined
        data = new int[10];

        // Assignment of each entry
        for (int i = 0; i <= 9; i = i + 1) {
            data[i] = i * i;
        }

        // The array is written to the console.
        for (int i = 0; i <= 9; i = i + 1) {
            System.out.println(i + " " + data[i]);
        }
    }
}//End of Program E66 ArrayOne
```

67. ***Exercise*** *Run the code and verbalize its functioning. Modify it to calculate the cubes of the first 12 numbers.* *Answer*

68. ***The mean and variance.*** *Let us see how one can calculate the* ***mean and variance*** *of some data. In the next program we can see another form as an* ***array*** *can be* ***declared and initialized****:*

```
//Declaration and assignation of the array Data
int  Data[] = {1,5,2,4,7,8,9,5,6,3,5};
```

In this instruction, we declare that Data[0]=1, Data[1]= 5, ..., Data[10]= 5. Moreover, we use the instruction

```
int n = Data.length;
```

to keep in the integer variable n the length of the array Data. In that way, one can modify the data at pleasure without worrying about further bookkeeping.

We implement the mean and variance by the formulas:

$Mean = \frac{\Sigma_0^{n-1} Data[i]}{n}$

$Var = \frac{\Sigma_0^{n-1} (Mean - Data[i])^2}{n-1}$

The whole program follows:

```
//Program E68 MeanData
```

```
//Same as program A83
//Mean and variance of some data.

package ejvol5p;

public class MeanData {

    public static void main(String args[]) {
        //Declaration and assignment of the data array
        int Data[] = {1, 5, 2, 4, 7, 8, 9, 5, 6, 3, 5};
        // Report of data
        System.out.println("Data are");
        // To know the length of the array : Data.length
        for (int i = 0; i < Data.length; i++) {
            System.out.print(Data[i]);
            System.out.println(); //pass to the next line
        }

        //   Let us calculate the mean
        double sum = 0;
        int n = Data.length;
        for (int i = 0; i < n; i++) {
            sum = sum + Data[i];
        }
        double mean = sum / n;
        System.out.println("Sum \t " + sum);
        System.out.println("Mean \t " + mean);

        //    Let us calculate the variance
        double sum2 = 0;
        for (int i = 0; i < n; i++) {
            sum2 = sum2 + (Data[i] - mean) * (Data[i] - mean);
        }
        double var = sum2 / (n - 1);
        System.out.println("Variance \t " + var);
    }
}//End of program E68 MeanData
```

69. ***Exercise*** *Run the code and verbalize its functioning. Explain how one can make sure that all data are included in the sum, mean and variance. Modify the code to calculate the mean and variance of own data of type double.* *Answer*

70. ***Navigating helps.*** *One can modify a program without understanding it at all. This is precisely a reason that enables evolution to make the same task. Nevertheless, the human experience is very simple: composing a program or modifying it is much easier if one understands it. Now, to understand large*

programs, it is convenient to know how to navigate across them. The reason is that to gain an overall insight into a large program, one must tie ends that are far away in the text of the program. These jumping operations could become tedious. So, NetBeans provides at least 4 proficient helps to ***navigate*** *through a program:*

1) Choose a pivot point, an important one, in your program and in front of it right click on the most left margin of the editor and choose the option 'Add Bookmark'. Please, choose not a commentary. Once clicked, a dialog will pop up in which the theme considered in your pivot point will appear. Click OK else change the pivot point else edit on your own the name of the bookmark. Once your bookmark gets accepted, a blue marker is displayed in the most right margin of the editor window. If you drag the cursor to it and waits for its conversion to a hand, the name of your bookmark is displayed. The bookmark can be invoked by clicking on it. When several book markers are elected, one can choose among them by name and by relative position. To delete a bookmark, drag the cursor to it over the left most bar, right click, and choose remove bookmark.

2) Right-click on the editor window and choose $Folding \rightarrow CollapseAll$. *As a result, NetBeans will hide the body of each method while titles remain visible. So, one can look the overall plan of design. One can expand and shrink at will any method: If a method is compressed, one can expand it by clicking on the corresponding plus sign at the margin. Also, one can hide the body or implementation of a method by clicking on the corresponding minus sign.*

3) NetBeans allows you to visit automatically the source code of a method, whose invocation appears at any place: click in the name of the method, right click, choose `Navigate` *and then* `Go to source`. *You will be teletransported to the source of the requested method.*

4) In the main menu, choose $Window \rightarrow Navigator$. *Go to the source of a method by double clicking on it.*

71. ***Challenge.*** *Enjoy and introvert the aforementioned navigating options of NetBeans. Get aware of what high quality software means. Proof else refute that the quality of software has an overwhelming diversity. How can we convert your answer into a question to study evolution, which is a blind and mindless developer of software?*

5.2 Tables with various columns

Java can manage **rectangular or 2-dimensional arrays**, which are tables with various columns, each one of which has the same number of rows. Java has a very natural declaration and management formalism.

72. ***Declaration and management***

The next instruction presents one of the various forms as Java allows to declare a two dimensional array or table. It has two columns and six rows, so one can interpret it as a frequency table, in which the first column keeps the class markers and the second the corresponding absolute frequencies:

```
int  Data[][] = {
       {3, 9},
       {4, 10},
       {5, 11},
       {6, 12},
       {7, 13},
       {8, 14}    };
```

To address specific entries in a two dimensional array, we must keep in mind that Java begins counting form zero. So the instruction

```
int z = Data[3][1]
```

assigns to z the value 12. In general, the first index address the row and the second the column.

73. *Frequency tables.* *Let us devise a program to calculate the mean, variance and standard deviation of any given frequency table.*

The entries are kept on the zeroth-column and their frequencies in the first. The mean and variance are calculated by the following formulas:

Number of rows of the table $= l = Data.length$
Total number of data $= n = \Sigma_0^l Data[i][1]$
$Mean = \frac{\Sigma_0^l Data[i][0] Data[i][1]}{n}$
$Var = \frac{\Sigma_0^l Data[i][1](Data[i][0]-mean)^2}{n-1}$

The program follows:

```
//Program E73 MeanFreqTable
//Finds the mean and variance of a table
//of absolute frequencies.

package ejvol5p;

public class MeanFreqTable {

//Frequency table: first coordinate = class marker
//second coordinate = frequency
    private static final double FREQTABLE[][] = {
        {4, 3},
        {7, 6},
        {10, 10},
```

```
        {13, 15},
        {16, 4},
        {19, 2}};

//Prints the frequency table
    private static void printFreqTable(double FreqTable[][]) {
        System.out.println("x and its absolute frequency");
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][]) {
        double n = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][]) {
        double mean;
        double sum = 0;
        double n = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            //Class marker times frequency
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
            n = n + FreqTable[i][1];
        }
        mean = sum / n;
        return mean;
    }

//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][]) {
        double mean = meanFreqTable(FreqTable);
        int m = FreqTable.length;
```

```
        // Let us calculate the variance
        double sum2 = 0;
        for (int i = 0; i < m; i++) {
            sum2 = sum2 + FreqTable[i][1] * (FreqTable[i][0] - mean)
                    * (FreqTable[i][0] - mean);
        }
        double n = 0;
        for (int i = 0; i < m; i++) {
            n = n + FreqTable[i][1];
        }
        double var = sum2 / (n - 1);
        return var;
    }

//Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][]) {
        double var = varFreqTable(FreqTable);
        double dev = Math.pow(var, 0.5);
        return dev;
    }

//Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][]) {
        printFreqTable(FreqTable);
        double n = nFreqTable(FreqTable);
        System.out.println("Number of measured objects = " + n);
        double mean = meanFreqTable(FreqTable);
        System.out.println("Mean = " + mean);
        double var = varFreqTable(FreqTable);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable);
        System.out.println("Deviation =  " + deviation);
    }

    public static void main(String[] args) {
        meanVarTable(FREQTABLE);
    }

}//End of Program E73 MeanFreqTable
```

74. ***Exercise and challenge.*** *Run out this code and study its design. Modify it to get the mean and variance of own data.*

75. ***Challenge*** *Figure out the way as Java deals with* ***3-dimensional arrays*** *and compose a program to demonstrate the use of one such array.*

5.3 Review

We have learned the fundamentals of one and two-dimensional arrays. Numeric arrays represent quantitative data, whose mean and variance were calculated. We also learned about the facilities of NetBeans to navigate across large programs. We will present a structured version of the same programs in the next chapter, such as they are used in ordinary life.

Chapter 6

Exploratory statistics

`Compressing information`

76. ***Purpose*** *We explore the very first part of statistics: from raw quantitative data to grouping, tables, mean, variance and charts. Structured programming, with parametrized methods, are used all around.*

6.1 A list of data

77. ***Definitions.*** *Data are said to be* ***quantitative or numeric*** *when they are numbers for which arithmetic operations make sense. If we have a* ***list*** *of numerical data X and a given datum is x_i, meaning that it appears in place i, the* ***mean or average*** *is*

$$\bar{X} = \frac{\Sigma x_i}{n} = \frac{\Sigma x}{n}$$

Sample variance s^2*: measures the nonconformity of the data if they are represented by the mean. (We divide by n in descriptive statistics and by $n-1$ in inferential tasks.)*

$$s^2 = \frac{\Sigma (x_i - \bar{X})^2}{n-1} = \frac{\Sigma (x - \bar{X})^2}{n-1}$$

The ***sample standard deviation*** s *is the square root of the variance:*

$$s = \sqrt{\frac{\Sigma (x_i - \bar{X})^2}{n-1}} = \sqrt{\frac{\Sigma (x - \bar{X})^2}{n-1}}$$

Coefficient of variation C*: measures the degree of homogeneity of data with respect to the mean*

$$C = \frac{s}{\bar{X}}$$

Other people prefer the coefficient of variation to be given in percentage.

78. ***Mathematics with Java.***

Elementary operations, +, -, *, /, are always ready to be used. Additionally, Java has **Math**, a powerful library or API (Application Program Interface) for more complex operations and functions. To see the multiple functions and procedures implemented in `Math`, all one needs to do is to type Math with a period `Math.` as in the next line:

```
double r = 5.2
//Use not copy+paste, instead type this directly:
doube rSquareRoot = Math.
```

if one waits for a while, NetBeans will display a list of all possible functions that one might pick up. One of them is

```
double sqrt(double a)
```

which is a method that computes the square root function of a number. Its input `double a` is a number of type double and produces an output of type double (high precision arithmetic). To see that, one clicks upon the function to have it highlighted, an operation that will be ensued by the popping up of a dialog window with the corresponding explanation.

Math has not a function for square. Instead it has the power function that one can use to calculate the desired power

```
double r = 5.2
doube rSquare = Math.pow(r,2);
double rCube = Math.pow(r, 3);
double rEightRoot = Math.pow(r, 0.125);
```

79. ***Exercise.*** *Compose a program that computes the mean and variance of a list of data.* *Answer*

80. ***Exercise.*** *Reuse previous programs to calculate the mean and variance of the integer numbers from 0 to 999. Hints: Declare an array of data for 1000 entries as follows:*

```
private static double DATA[] = new double[1000];
```

and use a `for-loop` *to fill in DATA.* *Answer*

6.2 Absolute frequency tables

If data are repeated, as in the scores of a group of students, one might group data in a **frequency table**, such that datum x is repeated F times. In that case, the mean and variance take the following form.

The mean is

$$\bar{X} = \frac{\Sigma(xF)}{n} = \frac{\Sigma(x_i F_i)}{n} = \frac{(x_1 F_1)+....+(x_n F_m)}{n}$$

The variance is:

$s^2 = \frac{\Sigma F_i (x_i - \bar{X})^2}{n-1}$

and the standard deviation is

$$s = \sqrt{\frac{S_{xx}}{n-1}}$$

81. ***Exercise.*** *Devise the code to calculate the mean and variance of a table of absolute frequencies.* *Answer*

82. ***Alternate procedure to find the variance:***
Step 1: the pondered sum of squares is calculated

$$\Sigma x^2 F = x_1^2 F_1 + x_2^2 F_2 + ... + x_n^2 F_m$$

Step 2: the square xx term is calculated

$$S_{xx} = \Sigma x^2 F - \frac{(\Sigma xF)^2}{n}$$

Step 3: the variance is:

$$s^2 = \frac{S_{xx}}{n-1}$$

83. ***Exercise.*** *Design the code that finds the variance using the alternate method.* *Answer*

6.3 Grouping

In exploratory analysis, **numerical data might be grouped** by classes corresponding to appropriate intervals or ranges. Grouping produces a table of absolute frequencies, which is usually drawn to see the tendency of the data, central or otherwise. The mean and variance might be calculated for that table.

84. ***Logic in Java***

We already know how to deal with an if structure, as in the next instruction:

```
if (i == 3) i = 4;
else i = 5;
```

The verbalization of this piece of code is: if `i` equals 3 then assign to i the value 4 otherwise assign 5 to it. Another way of verbalization: if is is `true` that `i` equals 3 then assign to i the value 4 otherwise assign 5 to it. This last form makes explicit the fact that we deal with logical values, `true` else `false`. The fact is that Java works with a two valued or **boolean logic**: a logical proposition can be either `true` else `false` but not both or anything else.

One can include in the `if-structure` a compound proposition that joins simple predicates. So, one can join two propositions with `&`, the `and` operator, or with `|`, the `or` operator. The `and` operator takes two boolean propositions and produces a boolean proposition that is `true` whenever both input propositions are `true` otherwise the output proposition is `false`. The `or` operator produces a `true` proposition when at least one of the incoming propositions is `true` and `false` when both proposition are `false`.

These operators are defined redundantly: `&` evaluates both sides of the operation. The same happens with `|`. By contrast, `&&` evaluates the left side of the operation, if it's true, it continues and evaluates the right side. The same happens with `||`.

In some cases, one needs to deny a given proposition, i.e., to produce a new proposition with inverted truth value. This is done by `!` the `not` operator.

As an example, let us see how the operator `and` works in the following instruction of the next program:

```
if ((Barriers[j] <= Vect[i]) & (Vect[i] < Barriers[j+1]))
  FreqTable[j][1] =FreqTable[j][1] +1;
```

This instruction has the syntax:

```
if ( (a) and (b)) then do this.
```

Exactly, the instruction verifies whether or not the number kept in `Vect[i]` is in the interval that goes from `Barriers[j]` to `Barriers[j+1]`. So, the number must be greater or equal to the inferior limit and less than the superior one.

If the logical condition is not fulfilled, one has the possibility to add an `else` clause to specify what to do, as in the example above.

85. *The next code groups data by interval classes. It outputs a table of absolute frequencies with its mean and variance. The intervals are closed from below and open from above to avoid double counting.*

```
//Program E85 Grouping
//Groups data in interval classes.
```

```
//Input:
// A list of data,
//a list of the borders of the intervals,
//and a  list of class markers,
//Outputs a frequency table with its
//mean and variance.

package ejvol5p;

public class Grouping {

//Declaration of data:
//observed values as a list
    private static final double DATAVECT[]
            = {
                48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
                50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
                49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
                49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
                51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
                49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
                50.8, 51.5, 51.0
            };

//Barriers of the intervals
    private static final double BARRIERS[] = {48.5, 49.5, 50.5, 51.5, 52.5};
    private static final double CLASSMARKERS[] = {49, 50, 51, 52};
    private static final int NCLASSES = CLASSMARKERS.length;
    private static final double FREQTABLE[][] = new double[NCLASSES][2];

//Vect[] is the input, a frequency table is the output
    private static void group(double Vect[]) {

        int nData = Vect.length;
        for (int j = 0; j < NCLASSES; j++) {
            FREQTABLE[j][0] = CLASSMARKERS[j];
            FREQTABLE[j][1] = 0;
        }
        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < NCLASSES; j++) {
                if ((BARRIERS[j] <= Vect[i]) & (Vect[i] < BARRIERS[j + 1])) {
                    FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
                }
            }

        }
```

```
    }

//Prints the frequency table
    private static void printFreqTable(double FreqTable[][]) {
        System.out.println("x and its absolute frequency");
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][]) {
        double n = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

//Calculates the sum XF   of
//a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][]) {
        double sum = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the sum X2F   of
//a table of absolute frequencies
    private static double
            sumX2FFreqTable(double FreqTable[][]) {
        double sum = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][0]
                    * FreqTable[i][1];
        }
        return sum;
```

```
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][]) {
        double n = nFreqTable(FreqTable);
        double mean = sumXFFreqTable(FreqTable) / n;
        return mean;
    }

//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][]) {

        double n = nFreqTable(FreqTable);
        double Sxx = sumX2FFreqTable(FreqTable)
                - Math.pow(sumXFFreqTable(FreqTable), 2) / n;
        double var = Sxx / (n - 1);
        return var;
    }

// Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][]) {
        double var = varFreqTable(FreqTable);
        double dev = Math.pow(var, 0.5);
        return dev;
    }

//Calculates the coefficient of variation of
//a table of absolute frequencies
    private static double coefficient(double FreqTable[][]) {
        double c = devFreqTable(FreqTable) / meanFreqTable(FreqTable);
        return c;
    }

// Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][]) {
        printFreqTable(FreqTable);
        double n = nFreqTable(FreqTable);
        System.out.println("Number of measured objects = " + n);
        double sumXF = sumXFFreqTable(FreqTable);
        System.out.println("sum xF  = " + sumXF);
        double mean = meanFreqTable(FreqTable);
        System.out.println("Mean = " + mean);
        double sumX2F = sumX2FFreqTable(FreqTable);
```

```
        System.out.println("sum x2F  = " + sumX2F);
        double var = varFreqTable(FreqTable);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable);
        System.out.println("Deviation =  " + deviation);
        double coefficient = coefficient(FreqTable);
        System.out.println("Coefficient of variation =  " + coefficient);
    }

    public static void main(String[] args) {
        group(DATAVECT);
        meanVarTable(FREQTABLE);
    }

}//end of Program E85 Grouping
```

86. ***Exercise.*** *Run the program and play with the code trying slight modifications.*

87. ***Exercise.*** *Grouping in the previous code is defined by the borders or barriers of the chosen intervals, for which an exhaustive list is given. This input format is good when one has intervals of irregular length. But when they have the same length, a more useful input might be the following: the initial border and the length of the intervals. Design the corresponding program, which must calculate class markers and the number of intervals.* *Answer*

88. *Challenge. Our previous procedures to group data in intervals is good for exploratory analysis, when one has no clear idea of what to do. But sometimes one might be strongly biased by an idea, say, that data obeys a central tendency distribution. In a case like that, a better approach to grouping might be to calculate the mean and deviation and use these statistics to pursue a better subdivision and ranges. Develop the corresponding code. Warning: if you consider that an open minded, exploratory analysis is the best, then you can speak of* `data driven analysis` *and refer to the other approach as* `theory biased` *or as* `theory laden`.

6.4 Bar charts

A table of absolute frequencies can be fairly represented by a **bar chart**.

89. *Research: have Java to draw a bar chart for a given table of absolute frequencies. Hint: consult Vol I of this series.*

Let us engage in the task of developing a program that compose the LaTeX code for the bar chart of a given frequency table. For the sake of those that are not acquainted with LaTeX, let us make a short introduction.

6.5 Easy LaTeX

How can one make sure that one understands something? By making things clear to others. In fact, understanding is a difficult, very difficult task and the questions of others efficiently alleviates that hard enterprise. Thus, becoming a writer is a natural byproduct for anyone that aims at understanding evolution. In this regard, there are many low level operations that one must do to communicate to people what one has written and that are in modern times solved by automatic computer guided text processing. The specific needs of the scientific community has generated specific solutions. One of the most outstanding is LaTeX.

LaTeX is a word processor for scientific documents that produces fine terminations. It is **evolvable** in at least two aspects. First, it can be used by beginners, who can add more and more packages to enable more and more possibilities in the measure that the need arises, say, for working with numbered equations. Second: it easily accommodates any change in the text while keeping demanded format including appropriate numeration and indexation.

LaTeX comes in various implementations, for Windows we have MikTex while Live-Tex is for Linux. Miktex must be downloaded, by its name, but Live-Tex usually comes with the Linux distributions.

LaTeX is usually seen through an interface, an application that alleviates the problem of the User of communicating with LaTeX. Some popular interfaces for Windows: TexnicCenter, WinEdt and Texmaker. For Linux: Texmaker, Kile and Lyx. All these programs can be downloaded, by their names, from Internet. With the exception of WinEdt, all are free. Nevertheless, WinEdt functions as shareware during two months. One can install it again if only the previous installation has been uninstalled. WinEdt is more easy to install than TexniCenter but both need the prior installation of MikTex.

LaTeX receives as input a document written in plain text and produces as output a dvi or a pdf document. The pdf's of this series were produced in that way and we have provided the LaTeX source in order that everyone could tailor at pleasure his pedagogic material.

90. ***Structural and regulative texts***

One must distinguish in LaTeX two types of texts: structural and regulative. To fix ideas, let us consider the next expression:

```
She said \textit{Hello}.
```

This expression is processed by LaTeX into the next one:

She said *Hello*.

We see a law: strings that begin with backslash, `\`, are **regulative instructions** for LaTeX, they are commands that indicate actions to be undertaken by LaTeX. In the considered case we have the command `\textit{}` to apply the

italic formatting for the text in within braces. Otherwise, strings are **structural,** part of the document to be processed.

91. ***Challenge.*** *Decide whether or not DNA strings also must be classified as regulative else structural instructions.*

92. ***A simple example.***

We also have general commands that are put at the beginning of the source document, the one that is written in plain text. So, the simplest article looks like this:

```
\documentclass[a4paper,10pt]{article}
\usepackage[latin1]{inputenc}
\title{\LaTeX for everybody}
\author{JR}

\begin{document}

%This is the symbol for a comment: the title must be declared and made.
\maketitle

\begin{abstract}
The simplest article is shown.
\end{abstract}

\section{Main parts of a \LaTeX document}

The code for  the simplest article contains:
\begin{enumerate}
\item The declaration of the document as article.
\item The type of  alphabet, latin1.
\item  The title.
\item  The abstract.
\item  A  single section.
\end{enumerate}

\end{document}
```

93. ***Exercise*** *In your favorite LaTeX editor, open a new document, paste this code, save it, compile it (call LaTeX over it) and look for the dvi (device independent) output.*

We see that the code for a LaTeX document must be interpreted according to a code, a dictionary, and then executed. So, it is software. Every LaTeX editor is a CAD (computer assisted design) facility that allows the writer to concentrate upon the creation of the content of the document together with the form to create the maximum clarity and impact on the reader. The writer decides what

must be done and LaTeX does it for him or her. This is an enormous alleviation. The reader is also benefited because it can concentrate on studying the content of the document without trying to decipher letters, symbols or equations.

94. ***Warning*** *In spite of all CAD facilities, it is normal to produce documents that are incomprehensible, entangled and void. Thus a writer must work very hard and during very long time upon him or herself to achieve clarity, purposiveness and grace. In other words, thousand failures precede a first moderate success. That is why children must be trained with wisdom but seriousness.*

6.6 PSTricks

LaTeX admits the possibility of graphic insertion in the text. Graphics have special formats, say, *jpg* or *gif*. Actually, this formatting compresses the drawings according to certain rules and that is all to them. There is nevertheless another approach to graphics that is very appropriate for science: we do not pay attention to a graphic itself but to the patterns that it contains. Now, there are some few fundamental patterns: lines, circles, polygons, parables, that are all geometric figures described by simple mathematical formulas. Therefore, a pattern approach to drawings leads to a tremendously efficient form of compression, purposiveness and clarity. If we find a transparent form of encoding patterns, we have a drawing language. In the LaTeX world, there are various famous graphic languages: LaTeX-2ϵ, PICTEX, PSTricks, Asymptote, Eukleides. We choose PSTricks which is very good for 2d graphics and admits extension to 3d figures. PSTricks means PostScript tricks and enrich LaTeX to high quality graphics.

A piece of code in PSTricks looks like this:

```
\begin{center}
\psset{unit=1 mm}
\begin{pspicture}(0,0)(45,35)
\pspolygon[linestyle=dashed](10,30)(20,20)(40,20)(30,30)
\rput(35,35){A}
\rput(45,20){B}
\rput(15,15){C}
\rput(5,35){D}
\end{pspicture}
\end{center}
```

and this code produces the next graphic

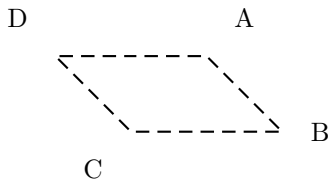


Figure 6.1. A parallelogram made with PSTricks

We distinguish a header that announces that the figure must be centered, the specification of the scale, the opening of a session with PSTricks, the extremal vertexes of the box where the figure must be allocated, the code for a polygon in dashed lines, the name of each vertex and the termination of the centering environment and of the PSTricks session.

95. ***Exercise*** *In the PSTricks code above, classify each portion of code as structural else as regulative.*

While PSTricks is wonderful, to compose the code for drawings by hand and mind is a very tedious task. This process is extremely slow because of the exceedingly high probability of committing bugs, whose corrections generate more bugs: Is there any form as we can use computers to help us? Yes:

96. ***Example***

The next is a LaTeX document that contains a piece of code for a drawing in PSTricks:

```
\documentclass[12pt]{article}
\usepackage[latin1]{inputenc}
\usepackage[english]{babel}
\usepackage{textcomp}
\usepackage{mathptmx}
\usepackage{helvet}
\usepackage{courier}
\usepackage{pstricks}
\usepackage{makeidx}




\begin{document}

\author{Elra Badil }
```

```
\title{ \LaTeX with PSTricks}

\date{}



\maketitle

\section{ Installing PSTricks }

For windows:
Miktex 2.7 or higher  will install automatically
every necessary add-on, so you must not worry
about how to install PSTricks.

For Linux: it comes with the distributions and
is always ready for use.

Graphic:


\begin{center}
\psset{unit=1 mm}
\begin{pspicture}(0,0)(45,35)
\pspolygon[linestyle=dashed](10,30)(20,20)(40,20)(30,30)
\rput(35,35){A}
\rput(45,20){B}
\rput(15,15){C}
\rput(5,35){D}
\end{pspicture}
\end{center}

\end{document}
```

97. ***Exercise*** *In your favorite LaTeX editor, open a new document, paste this code, save it, compile it (call LaTeX over it) and look for the dvi (device independent) output. In rare cases, the dvi document contains a complete and perfect output. Most usually, one transforms the dvi document in a ps version (look for the corresponding menu), and this in turn is transformed into a pdf document (look again for the menu) and after calling the pdf viewer, one can see everything in order.*

6.7 Automatic composition of PSTricks code

If we have a frequency table and want the PSTricks code for the corresponding bar chart, we can use Java to produce the code automatically. We see an implementation of this idea in the next program, which is forced to execute a **change of type** of a variable from `double` to `int`.

98. ***Change of type***

One needs in some occasions to assign to a variable of type `double` a given integer value. The easiest way to do that is as follows:

```
double aa = 3.4;
int i = (int) aa;
System.out.println(i);
double bb = 3.8;
int j = (int) bb;
System.out.println(j);
```

The reported values of `i` and `j` are both 3, so the rounding is downwards. More ways of associating an *int* to a *double* can be found in the *Math* library.

99. ***The code.*** *The next program receives as input a frequency table and produces the (LaTeX) PSTricks code for the corresponding bar chart.*

```
//Program E99 PstricksCode
//The next program receives as input a frequency table
//and produces the (Latex) PSTricks code for
//the corresponding bar graph.

package ejvol5p;


public class PstricksCode {

    //Width of each BAR
    private static final double DELTAX = 4;

    //Declaration of input: a frequency table,
    //the value of x and its frequency
    private static final double FREQTABLE[][]
            = {
                {3, 3},
                {4, 4},
                {5, 6},
                {6, 2},
                {7, 1},
```

```
                {8, 5}
            };

//Produce the pstricks  code for the bar graph
//of a frequency table
    public static void makeCode(double FreqTable[][]) {

        int n = FreqTable.length;
        //The minimum x-coordinate is found
        double Minxx = 1000;
        for (int i = 0; i < n; i++) {
            if (FreqTable[i][0] < Minxx) {
                Minxx = FreqTable[i][0];
            }
        }
        int Minxxx = (int) (DELTAX * Minxx - DELTAX);
        //The maximum x-coordinate is found
        double Maxx = 0;
        for (int i = 0; i < n; i++) {
            if (FreqTable[i][0] > Maxx) {
                Maxx = FreqTable[i][0];
            }
        }
        int Maxxx = (int) (DELTAX * Maxx + DELTAX);
        //The maximum y-coordinate is found
        int Maxyy = 0;
        for (int i = 0; i < n; i++) {
            if (FreqTable[i][1] > Maxyy) {
                Maxyy = (int) FreqTable[i][1] + 2;
            }
        }
        //Starting clause
        //x- and y-units.
        //The needed space is defined
        System.out.println("\\begin{center}"
                + "\n\\psset{xunit=0.15,yunit=0.3}"
                + "\n\\begin{pspicture}(" + Minxxx + ",0)("
                + Maxxx + "," + Maxyy + ")");
        //A rectangle is drawn for each entry in the
        //frequency table

        int lim1, lim2;
        for (int i = 0; i < n; i++) {
            lim1 = (int) (DELTAX * FreqTable[i][0] - DELTAX / 2);
            lim2 = (int) (lim1 + DELTAX);
            System.out.print(
```

```
                    "\n\\psline(" + lim1 + ",0)"
                    + "(" + lim1 + "," + FreqTable[i][1] + ")"
                    + "(" + lim2 + "," + FreqTable[i][1] + ")"
                    + "(" + lim2 + "," + "0)"
            );
        }

        //Horizontal axis
        System.out.println("\n\\psline("
                + Minxxx + ",0)(" + Maxxx + ",0)");

        //Class markers (first coordinates) are indicated
        for (int i = 0; i < n; i++) {
            int xposition = (int) (DELTAX * FreqTable[i][0]);
            int yposition = - 1;
            System.out.print(
                    "\n\\rput*(" + xposition + "," + yposition + "){"
                    + (int) (FreqTable[i][0]) + "}"
            );
        }

        //End clause
        System.out.println("\n\\end{pspicture}");
        System.out.println("\\end{center}");
    }

    public static void main(String[] args) {
        makeCode(FREQTABLE);
    }
}//End of Program E99 PstricksCode
```

100. ***Exercise.*** *Run the program and play with the code trying slight modifications.*

101. ***Exercise and challenge.*** *Modify the previous program so that the PSTricks code includes instructions to print the height of each column over its roof. Challenge: presented programs are not general: if one changes the frequency table a bit, a lot of coordinated changes must be done in the code to achieve the promised result. Attempt the corresponding generalization.* *Answer*

102. ***Expertise: graphics editors***

What we have seen in the previous code can be perfected as much as desired. Readers interested in this line of research can give a look to some free application that are written in Java or in C and that help a lot in the very difficult task of composing the PSTricks code for a drawing: one is **JPictEdt**, other is **xfig** and a third one is **Inkscape**. One can use Google to download them by their

names. The code is also available and in some cases one is allowed to made modifications with no commercial profit. In spite of some bugs, the generality of code represents a good challenge for those that want to incursion in the design of professional quality software: we are principally interested in software that runs and nothing else but if one wants professional software, one must face additional degrees of complexity and sophistication that might look as exaggerate but that now are normal in the trade.

6.8 Conclusion

We have used some statistical tasks, to find the mean and variance of a list of data, to exercise our Java skills. We begin to notice that very powerful programs can be constructed by recursion to some few Java reserved words and control structures. This is the power of the word, of the language, in action. We saw how the universal division of texts between regulative and structural ones has been implemented in LaTeX. Regulative texts encode for commands while structural ones encode for arguments for the commands. It is the very same division of verb and substantive in human languages. Besides, we made a tiny excursion into the world of automatic software design, a science that awakes some intrigues: Can we use automatic software design to create a new species? Was the software of the genome designed automatically?

Chapter 7

The Binomial and the normal distributions

```
Yes else not and further
```

Two very simple models of probabilistic sources are the Binomial and the normal distribution.

7.1 The binomial distribution

We use a **Binomial distribution** to model a random source of data that produces at each trial one of two possible outcomes, head else tail. If the probability of head is p and that of tail is q, for n trials, the probability of r heads and $n-r$ tails is given by

$$p(r) = C_r^n p^r q^{n-r} = C_r^n p^{n-r} q^r$$

where

$$C_r^n = \frac{n!}{r!(n-r)!}$$

103. ***The code*** *The next code receives as input the degree of the binomial (the number of coins), the probability of head and amplification factor. It produces the corresponding probability distribution:*

```
//Program E103 Binomial
//Outputs the distribution of
//the  binomial distribution
//for  n coins
```

```
//probability of head p,
//and F repeats.

package ejvol5p;


public class Binomial {

    //The degree of the binomial, the number orf coins.
    private static final int N = 5;
    private static final double COEFFICIENTSBINOMIAL[]
            = new double[N];
    private static final double TERMSBINOMIAL[]
            = new double[N];
    private static final double TERMSBINOMIALF[]
            = new double[N];
    private static int n;//number of coins
    private static double p; //probability of head
    private static int F;//number of repeats

    //Prints data in vector
    private static void printVector(double Vect[], int n) {
        for (int i = 0; i <= n; i++) {
            System.out.println(Vect[i]);
        }
    }

    //Sums data in vector with real numbers
    private static double sumVector(double Vect[], int n) {
        double sum = 0;
        for (int i = 0; i <= n; i++) {
            sum = sum + Vect[i];
        }
        return sum;
    }

    //Returns numb!
    private static long factorial(long numb) {
        if (numb <= 1) {
            return 1;
        } else {
            return numb * factorial(numb - 1);
        }
    }

    //The  binomial distribution
```

```
    //for  n repeats
    //probability of head p,
    //amplification factor F.
    private static void binomialDistribution(int n,
            double p, double F) {
        long biCoeff;
        for (int r = 0; r < n + 1; r++) {
            biCoeff = factorial(n)
                    / (factorial(r) * factorial(n - r));
            COEFFICIENTSBINOMIAL[r] = biCoeff;
        }
        System.out.println("\nCoefficients ");
        printVector(COEFFICIENTSBINOMIAL, n);
        System.out.println("Sum = "
                + sumVector(COEFFICIENTSBINOMIAL, n));

        for (int r = 0; r < n + 1; r++) {

            TERMSBINOMIAL[r] = COEFFICIENTSBINOMIAL[r]
                    * Math.pow(p, r) * Math.pow(1 - p, n - r);
        }
        System.out.println("\nProbabilities for 0,1,2...");
        System.out.println("Paste this vector to Excel"
                + " and make a graphic");
        printVector(TERMSBINOMIAL, n);
        sumVector(TERMSBINOMIAL, n);
        System.out.println("Sum = "
                + sumVector(TERMSBINOMIAL, n));
        for (int k = 0; k < n + 1; k++) {
            TERMSBINOMIALF[k] = F * TERMSBINOMIAL[k];
        }
        System.out.println("\nAbsolute frequencies for F = " + F);
        printVector(TERMSBINOMIALF, n);
        System.out.println("Sum = "
                + sumVector(TERMSBINOMIALF, n));
    }

    //Detailed computation of the mean
    private static void detailedMean() {
        System.out.println("\nDetailed mean");
        double sum = 0;
        double term;
        for (int r = 0; r < n + 1; r++) {
            term = r * TERMSBINOMIAL[r];
            System.out.println(term);
            sum = sum + term;
```

```
        }
        System.out.println("mean = " + sum);
    }

    //Detailed computation of the variance
    private static void detailedVariance() {
        System.out.println("\nDetailed variance");
        double sum = 0;
        double term;
        for (int r = 0; r < n + 1; r++) {
            term = Math.pow(r - n * p, 2) * TERMSBINOMIAL[r];
            System.out.println(term);
            sum = sum + term;
        }
        System.out.println("Variance = " + sum);
    }

    public static void main(String[] args) {
        n = 3;//number of coins
        double one = 1;
        p = one / 8; //probability of head
        F = 34;//number of repeats
        System.out.println("Binomial distribution: \n"
                + "n = " + n + ", p = " + p + ", repeats = " + F);
        binomialDistribution(n, p, F);
        detailedMean();
        detailedVariance();
    }

}//End of Program E103 Binomial
```

104. ***Exercise.*** *Run the program and play with the code trying slight modifications.*

We have seen how one can calculate the binomial distribution but we do not know how to simulate it. We will learn how to do that in the section below, which is a direct application of the inbuilt ready to use random generator that comes with Java.

7.2 Random numbers

A succession of numbers is random when one cannot predict what will be the next number. To simulate random numbers, Java uses a generator of pseudo-random numbers, i.e., deterministic numbers that are almost random, i.e., that pass many tests in regard with randomness but that are produced by concrete pieces of code with deterministic outputs.

105. ***Definition.*** *A* ***random generator*** *is a code that serves as a controllable source of random numbers, i.e., a sequence of numbers such that the probability of correctly guessing the next numbers is almost nil. Random numbers are used everywhere in simulation: to simulate behaviors and to contrast null hypotheses. In the next program we see how they are put into work.*

The fundamental instructions to work with random numbers follow. In the first sentence we declare that we have the intention of working with random numbers, so we command to bring forth the appropriate tool, kept by Java in a library called util. The second instruction declares that we turn of the generator. The third instruction declares that we will generate a nonnegative random `int` number less than 6. Zero is a possible outcome.

```
//In the preable of the program
import java.util.Random;
//Somewhere at the beginning
Random r  = new Random();
//In within a method
int  x= r.nextInt(6)
```

106. ***The code to demonstrate the use of random numbers follow.***

```
//Program E106 RandomNumbers
//same as Program A97.
//Generation of random numbers and chars
//The generation of random numbers
//is supported by the class Random,
//so, that class must be activated.

package ejvol5p;

import java.util.Random;

public class RandomNumbers //Random numbers  and chars.
{

    public static void main(String args[]) {
        //A sequence of random numbers is generated as follows:
        //We turn on the sequence:
        Random r = new Random();
        //We invoke the next random number,
        //each time it is needed.
        //The parameter 6 means that the random numbers
        //will be nonnegative and less than 6.
        //Generated numbers are integers.
        System.out.println("Random number " + r.nextInt(6));
```

```
        System.out.println("Random number " + r.nextInt(6));
        System.out.println("Random number " + r.nextInt(6));
        System.out.println("Random number " + r.nextInt(6));
        System.out.println("Random number " + r.nextInt(6));
        System.out.println("Random number " + r.nextInt(6));
        System.out.println("Random number " + r.nextInt(6));
        System.out.println("Random number " + r.nextInt(6));
        System.out.println("Random number " + r.nextInt(6));
        System.out.println("Random number " + r.nextInt(6));

        //Simulation of a die
        // For a large sequence, a loop is necessary
        System.out.println("Simulation of a die");
        for (int i = 1; i <= 20; i++) {
            // To simulate a die, we add 1
            // because Java begins counting with 0.
            // Observe that the symbol +
            // is used in two very different senses:
            System.out.print(1 + r.nextInt(6) + "  ");
        }
        System.out.println();
    }
}//End of Program E106 RandomNumbers
```

107. ***Exercise*** *Run and modify the code to see how it functions.*

108. ***Challenge*** *Play with the following instructions to verify their function:*

Random r = new Random(**seedValue**); (always initializes the generator in the same form)

r.**nextLong();** (generates a random integer of type long, which is more powerful than that of type int.)

r.**nextFloat();** and r.**nextDouble();** (generates a decimal number between 0 and 1 with a uniform distribution.)

r.**nextGaussian();** (generates a random decimal number with a z distribution.)

7.3 Simulation of a binomial distribution

To simulate a random source that obeys a binomial distribution is expedite in Java. We use a random variable with uniform distribution in the (0,1) interval and posit a threshold p. Next, we throw a number, if it is less than p, we define that a head happened else that a tail was gotten.

109. ***The code.*** *This program simulates a random variable with a binomial distribution $X \sim Bi(n, p)$, with n coins and with probability of head equal to p.*

```
//Program E109 BinomialSim
//This program simulates a binomial distribution
//with n coins and probability of head p.
//Slight modification of Program A199

package ejvol5p;

import java.util.Random;

public class BinomialSim {
    //Random generator is turned on

    static Random r = new Random();

    //Statistical test to see
    //the fairness of the simulation
    public static void test(int Males, double p) {

        double sigmaP = Math.pow((p * (1 - p) / 1000), 0.5);
        double M = Males;
        double zexp = (M / 1000 - p) / sigmaP;
        System.out.println("The z of the experiment is " + zexp);
        double s = 1.96;
        System.out.println("The critical z is   " + s);
        String A1 = "Yes.";
        String A2 = "No.";
        String l;
        if (Math.abs(zexp) < s) {
            l = A1;
        } else {
            l = A2;
        }
        System.out.println("Is the generator fair?  " + l);
    }

    public static void main(String[] args) {
        // Probability of head
        double p = 0.2;
        int numberHeads = 0;
        int n = 1000;
        System.out.println("Binomial distribution ");
        System.out.println("Number of coins " + n);
        System.out.println("Probability of tail " + p);
        int exp = (int) (n * p);
```

```
        for (int j = 0; j < n; j++) {
            //We generate a random number
            //in within 0 and 1
            //with a uniform distribution
            double c = r.nextDouble();
            // We verify c against the threshold p
            if (c < p) {
                numberHeads = numberHeads + 1;
                System.out.println("Head ");
            } else {
                System.out.println("Tail ");
            }
        }
        System.out.println("Actual number of heads: " + numberHeads);
        System.out.println("Expected number of heads " + exp);
        test(numberHeads, p);
    }
}//End of Program E109 BinomialSim
```

110. ***Exercise.*** *Run the program and play with the code.*

7.4 The standard normal distribution

The idealization of a binomial distribution by a continuous curve renders the normal distribution. This distribution is the usual model for a **central tendency distribution**, one that groups itself around the mean or median.

111. ***Definitions.*** *The* ***Euler number****, e is defined by*

$e = \sum_0^\infty \frac{1}{k!}$

where $0! = 1$ *and* $r! = (r-1)!r$*. The value of* e *is approximately* 2.71*.*

A non negative function f from reals on reals is a ***probability density function*** *of a random variable X when the area under the function and in within points a and b is equal to the probability $p(a < X < b)$.*

The ***Bell of Gauss*** *is the function defined by*

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where e represents the Euler number.

When the Gauss of Bell represents the probability density function of a random variable X, we have the ***normal distribution****. In this case, μ is the mean and σ is the standard deviation of the distribution, which is noted*

$$X \sim N(\mu, \sigma)$$

This expression reads: X is a random variable with normal distribution of mean (or expected value) μ and deviation σ.

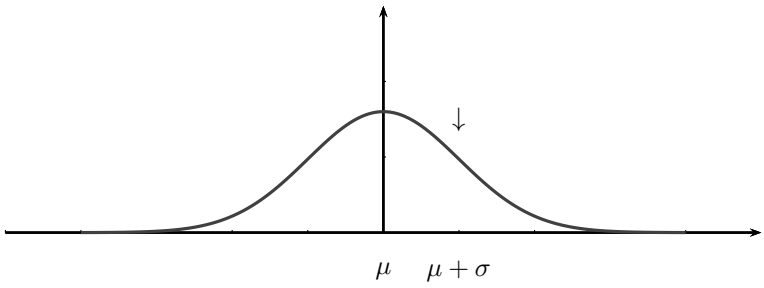


Figure 7.0. The Bell of Gauss. The center of symmetry is the mean. The inflexion point is where the curvature changes as indicated with a vertical arrow. The distance between the mean and the inflexion point is the standard deviation.

112. ***Theorem.*** *The binomial distribution $Bi(n, p)$ with n coins and the probability of head p, is approximated by the normal distribution with mean $\mu = np$ and standard deviation $\sigma = \sqrt{np(1-p)}$. The approximation is the better the larger is n and the closer is p to 1/2 .*

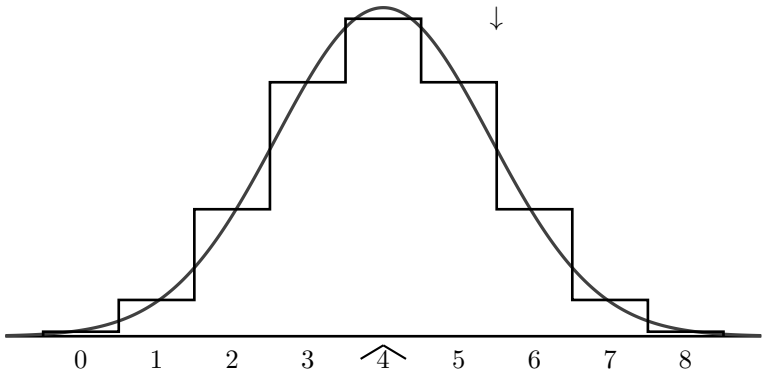


Figure 7.1. The $Bi(8, 1/2)$ is perfectly approximated by a Gauss Bell.

113. ***Challenge.*** *Design a Java Applet to show the fitting of the binomial distribution by the Bell of Gauss. Applets and graphics were introduced in Vol 1 of this series, Java for the study of evolution.*

114. ***Challenge.*** *Develop a Java program to produce the PSTricks code for a Bell of Gauss. Hint: study the PSTricks code for the previous figure. That code can be found in the LaTeX source accompanying this volume.*

115. ***The code for a simulation of a normal distribution follows.*** *We use the method*

```
double d = r.nextGaussian()
```

which generates a random variable with normal distribution of mean 0 and standard deviation 1. Next, we produce a transformation to generate a random variable with predetermined mean equal to μ and deviation σ:

```
int c = (int) (Math.floor(sigma*d + mu));
```

We use a Math method to round downwards an output of type double. The code follows:

```
//Program E115 NormalSim
//Simulation of
//the normal distribution.
//with  mean mu and deviation sigma
//Based on program A208.
//The program outputs a frequency table,
//of sampled values.

package ejvol5p;

import java.util.Random;

public class NormalSim {
    //  Turn on of the random generator

    static Random r = new Random();

    static int Freq[];//Predeclaration

    public static void main(String[] args) {
        //mean
        double mu = 5;
        //Deviation
        double sigma = 3;
        //size of the random sequence
        int lim = 1000;
        //Frequency table.
        Freq = new int[5000];
```

```
        // Initialization
        for (int j = 0; j < 60; j++) {
            Freq[j] = 0;
        }
        //We generate a random sequence
        for (int j = 0; j < lim; j++) {
            //Random numbers with Gaussian distribution
            //mean zero and deviation 1.
            double d = r.nextGaussian();
            // System.out.println(d ) ;
            // Rescale numbers: mean m and deviation s
            int c = (int) (Math.floor(sigma * d + mu));
            //Shift that allows registering negative values
            Freq[c + 30] = Freq[c + 30] + 1;
        }
        int k;
        // Report
        System.out.println("k" + "\t p(k)");
        for (int j = 1; j <= 100; j++) {
            k = j - 30;//Contrary balancing shift
            System.out.println(k + "\t" + Freq[j]);
        }
    }
}//End of Program E115 NormalSim
```

116. ***Exercise.*** *Run the code and make a graphic of the histogram of absolute frequencies. Decide by instinct whether or not the bar chart can be fitted by a Bell of Gauss. Play with the code: modify the mean and/or the deviation. Redefine the size of the sequence for increasing limits and describe the behavior of the distribution.*

7.5 Conclusion

We use a Binomial distribution to model a random source of data that produces at each trial one of two possible outcomes, head else tail. We have used a receipt to calculate the probability of k heads if one throws n coins. We have used the pseudo-random generator to simulate that situation. The normal distribution can be understood as the enveloping continuous curve to the binomial distribution else as an idealization of a distribution of central tendency. The simulation of a normal distribution is also straightforward with Java.

Chapter 8

The scientific method

Ideas vs. facts

117. ***Introduction and motivation.*** *The scientific method is basically an attitude towards knowledge: science is distinguished by its compulsory tendency to contrast what one sees in the world with what one thinks or believes in. In this chapter, we see how this operates in normal life to abstract next a proper methodology.*

8.1 Null hypotheses

The simplest model of science is that it is a set of propositions with experimental value, i.e., that one can test in experiments or in observations.

118. ***Example.*** *Let us reformulate in scientific terms a usual behavior that one follows by instinct. We follow our course of Statistics Vol. 1.*

Laura is a student that leaves the campus at 2 pm and arrives at home nearly 3 and a half. Rarely before 3 or later than 4. Today, it is Thursday. Laura has not arrived and it is a quarter past four. Her mother expects her and she is already going into preoccupation, but she recovers the calm after a call from Laura: she is getting late because she is working with her friends in the solution of a homework. Time flies and the mother is just to fall into rage and desperation because it is already six o'clock and Laura has not returned home. Laura calls her mother but her explanations do not cause any appeasing effect over her. So, the mother begins to call the friends of Laura and uncle John. A bit later she plans to call to the police and to the father at his work. The mother suspects that there is something more besides a simple homework: it is clear that something happened to Laura else she is hiding some facts.

Let us reformulate this story in statistical terms:

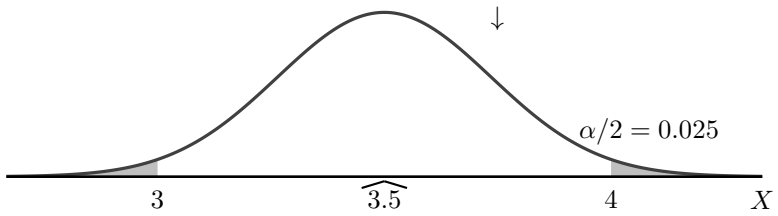


Figura 8.0. X is the arriving time of Laura at her home. If $3 < x < 4$, the mother takes a decision: to remain quietly. When $x < 3$ or $X > 4$ she takes another decision: to turn alarms on.

Let X be the arrival time of Laura to her house. X has a central tendency that is modelled by a normal distribution. The mean is 3.5 and the deviation is chosen in such a way that the 95% of times, X is in within 3 and 4. The other 5% she arrives before 3 else past 4. So, the 2.5% she arrives earlier than 3 and other 2.5% she arrives later than 4. Now, all that is equivalent to saying that in within 3 and half past 3 there are 2 standard deviation and that there are other 2 standard deviations in within half past 3 and 4. Therefore, the deviation is a quarter. In short: $X \sim N(3.5,\, 1/4)$. Let us notice that if Laura arrives before 3, the mother begins to suspect that she does not want to make the homework. So it is bad if she arrives too early and it is also bad if she arrives too late. We say that we work with two tails, because the two extremes cause to turn the alarms on.

The context says that there are two barriers that divide the usual facts from the unusual ones. The barriers are the third and fourth hours. Laura arrives at home in within those barriers the 95% of times. And the 5% of times, Laura breaks the barriers. We say that we work with a significance of 5%. If an event is normal with this significance, we take a decision: let us remain quietly. But if an event is classified as abnormal, we take another decision: to turn the alarms on.

We have used an automatic procedure to choose one of two decisions. First: to belief that Laura is solving her homework. Second: to think that something bad is occurring to Laura or that she hides some fact. Our automatic procedure is error prone: when Laura arrives in within 3 and 4, our procedure dictates that Laura is a good girl. But that may be false: what about George, a bad company according to their fathers, and that loves to accompany her from the campus to near home? Or, if she arrives very late our procedure dictates that shes is a bad girl. But, what if she was dedicated to solve that very difficult task of physics with spinning bodies and conservation of angular momentum? We have the very same situation as in a verdict: one can blame the innocent or one can freed the culprit.

This is all to the scientific method: everything in science is no more than

the same game, although under possible more complex situations. Let us summarize:

119. ***The scientific method in a nut shell.***

Science *is basically a contrast between what one sees and what one beliefs. For simple models, we can measure the discrepancy between facts and ideas. If this discrepancy is in within certain bounds that define what is normally expected, one usually prefers to remain quietly assuming that what one believes is the truth. But if the discrepancy is too large, one is invited to invent and test new explanations for observed facts. More operationally:*

1. *Formulate your belief and classify outcomes as usual else as unusual.*
2. *Experimental events that classify as usual are explained by your belief, which is officially called a* ***null hypothesis****. Unusual ones deserve the invention and study of other explanations.*
3. *Null hypothesis must be clear and easy to calculate i.e., it must be simple to calculate outputs to classify then as usual or normal else as unusual or abnormal. Therefore, null hypotheses always appear as an equation, say, that the mean is 5 or that the bigger variance is 7 times the smaller one.*
4. *The scientific method is error prone: one can reject the null hypothesis by considering an extreme event that is perfectly explained by the same mechanism that gave rise to the null hypothesis. Or, one can accept the null hypothesis when an event looks normal but in reality it was caused by another mechanism not considered the null hypothesis. That is why we say that the scientific method has no relation with the truth but only with the confidence in our beliefs.*

8.2 Testing a mean

Our purpose now is to see how the scientific method is implemented for the simplest case, which refers to a null hypothesis about the mean of a normal distribution.

8.2.1 Contrast against a single datum

The simplest case of the study of a null hypothesis refers to the mean of a normal distribution.

120. ***Example.*** *Let us suppose that in certain subgroup of people the length of the hair of men is $X \sim N(3,\ 0.8)$. We have found a young man with a hair with length $X = 21$. Do we have a reason to believe that this man is an exponent of a subculture different than that of the common people?*

Solution: The first step is to define the proportion of events that must classify as normal in contrast with those that classify as abnormal. That proportion is the **significance**. By default, it is taken as 0.05. The second step is to formulate H_o, the null hypothesis. In regard with the length of hair, let us pay attention to the mean:

$H_o : \mu = 3$

This null hypothesis can be read in the following form: the value of an outcome is the mean $\mu = 3$ except by a random noise with a normal distribution with mean zero and standard deviation $\sigma = 0.8$. So

$X = \mu + w$

where w obeys a normal distribution $w \sim N(0, 0.8)$.

The third step is to define the number of tails of our test. Since our attention was attracted by a young fellow, whose hair was too long, we are paying attention to the upper tail, that of the long hair. So, our test is one-tailed and all the significance goes there.

The fourth step is to calculate the null hypothesis and to divide outputs between two classes: normal and abnormal. To that aim, we simulate a source $X \sim N(3, 0.8)$, and for each output x we measure the discrepancy between x and μ. The measure of the discrepancy is given by

$d = \frac{x-\mu}{\sigma} = \frac{x-3}{0.8}$

The d values are grouped into a frequency table and thanks to it, the normal values are divided from the abnormal ones in the upper tail, knowing that the abnormal ones must sum up to a proportion of 5% of all the population.

121. ***The next code*** *contrast a fact,* $x_o = 21$*, against the null hypothesis* $H_o : \mu = 3$ *(knowing that* $\sigma = 0.8$*).*

```
//Program E121 TestMean
//This program contrasts   a    fact, XO = 21,
//against the null hypothesis
//that x must follow a normal distribution with
//MEAN = 3
//(knowing that   the DEVIATION is  0.8).

package ejvol5p;

import java.util.Random;
```

```
public class TestMean {
    //fact
    private static final double XO = 21;
    //idea
    private static final double MEAN = 3;
    //restriction
    private static final double DEVIATION = 0.8;
    //number of radom trials
    private static final int NTRIALS = 7435;
    //Significance level
    private static final double ALPHA = 0.05;
    //Turn on of the random generator
    static Random r = new Random();

    //Numbers generated at random
    private static final double EVENTS[] = new double[50000];
    //Discrepancy between fact and idea
    private static final double DISCREPANCY[] = new double[50000];
    private static double min;
    private static double max;
    //The left border of intervals, less or equal to the
    //minimum value of data.
    private static double infLimit;
    //Interval length
    private static double intervalLength;
    //Max number of allowed classes
    private static final int MAXNCLASSES = 100;
    //Borders of the intervals
    private static final double BARRIERS[] = new double[MAXNCLASSES + 1];
    private static final double CLASSMARKERS[] = new double[MAXNCLASSES];
    private static int nClasses = 20;
    //Distribution of the discrepancies
    private static final double FREQTABLE[][] = new double[MAXNCLASSES][2];

    //Generates    random numbers
    //with the requested normal distribution
    //This is a virtual world that obeys the
    //null hypothesis.
    public static void virtualWorld(double mean, double deviation) {
        //We generate a random sequence
        for (int i = 0; i < NTRIALS; i++) {
            //Random numbers with Gaussian distribution
            //mean zero and deviation 1.
            double d = r.nextGaussian();
            // System.out.println(d ) ;
```

```
            // Rescale numbers: new mean = mean  and
            // new deviation = deviation
            EVENTS[i] = deviation * d + mean;
        }
    }

    //Measures the discrepancy between an event
    //and an idea (the mean)
    public static double discrepancy(double event,
            double mean,
            double deviation) {
        double d;
        //Measuring of discrepancy
        d = (event - mean) / deviation;
        return d;
    }

    //Measures the discrepancy between each fact of
    //the virtual world and idea (the mean)
    public static void discrepancy(double Events[],
            double mean) {
        double d;
        for (int i = 0; i < NTRIALS; i++) {
            //Measuring of discrepancy
            d = discrepancy(Events[i], mean, DEVIATION);
            DISCREPANCY[i] = d;
        }
    }

    //The range of discrepancies is evaluated
    //and is used to calculate the length of each
    //interval to group Discrepancies.
    public static double intervalLength() {
        min = 100000;
        max = 0;
        for (int i = 0; i < NTRIALS; i++) {
            if (DISCREPANCY[i] < min) {
                min = DISCREPANCY[i];
            }
            if (DISCREPANCY[i] > max) {
                max = DISCREPANCY[i];
            }
        }
        System.out.println("min = " + min);
        System.out.println("max = " + max);
        double range = max - min;
```

```
        double length = range / nClasses;
        return length;
    }

    //Prints the frequency table
    private static void printFreqTable(double FreqTable[][],
            int nClasses) {
        System.out.println("x and its absolute frequency");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

    //Calculates the sum    of absolute frequencies
    //of a frequency table
    private static void
            test(double FreqTable[][]) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][1];
        }
        System.out.println("Number of events = " + sum);
    }

//The border of each interval is calculated
    private static void getBorders(double Vect[]) {
        BARRIERS[0] = infLimit;
        CLASSMARKERS[0] = infLimit + intervalLength / 2;
        //Calculate maximal value of data
        double maxValue = Vect[0];
        for (int j = 1; j < Vect.length; j++) {
            if (Vect[j] > maxValue) {
                maxValue = Vect[j];
            }
        }
        System.out.println("Max value = " + max);
        //Calculates number of classes
        nClasses = (int) ((maxValue - infLimit) / intervalLength) + 1;
        System.out.println("Number of classes = " + nClasses);
        //Calculates interval borders and class markers
        for (int j = 1; j <= nClasses; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + intervalLength;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + intervalLength;
        }
        /*System.out.println( "Borders are" );
```

```
        printVector(Barriers);
        System.out.println( "Class markers are" );
        printVector(classMarkers);*/
    }

//Data are grouped into interval classes.
    private static void group(double Vect[]) {
        intervalLength = intervalLength();
        infLimit = min - intervalLength / 2;
        getBorders(Vect);
        int nData = NTRIALS;
        //Class markers a re defined
        for (int j = 0; j < nClasses; j++) {
            FREQTABLE[j][0] = CLASSMARKERS[j];
            FREQTABLE[j][1] = 0;
        }
        //Frequencies are calculated.
        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < nClasses; j++) {
                if ((Vect[i] >= BARRIERS[j])
                        & (Vect[i] < BARRIERS[j + 1])) {
                    FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
                }
            }

        }
    }

    //The critical value in the upper tail is calculated
    //with significance level = ALPHA = 0.05
    private static double criticalValue(double Vect[]) {
        double c = 0;
        //Number of events that classify as extreme
        int e = (int) (NTRIALS * ALPHA);
        System.out.println("Number of extreme events = " + e);
        //Clon of Vect[]
        double Clon[] = new double[NTRIALS];
        /*
        for (int i = 0; i < NTRIALS; i++) {
        Clon[i] = Vect[i];
        }
         */
        System.arraycopy(Vect, 0, Clon, 0, NTRIALS);
        //Discrepancies are sorted
        //in decreasing order until completion of
        //e events.
```

```
        int Champ;
        for (int i = 0; i < e; i++) {
            Champ = 0;
            for (int j = 1; j < NTRIALS; j++) {
                if (Clon[j] >= Clon[Champ]) {
                    Champ = j;
                }
            }

            System.out.println(i + "th ind. is  No "
                    + Champ + " Discrep =  " + Clon[Champ]);
            c = Clon[Champ];
            Clon[Champ] = 0;
        }
        return c;
    }

    public static void main(String[] args) {
        virtualWorld(MEAN, DEVIATION);
        System.out.println("Discrepancies");
        discrepancy(EVENTS, MEAN);
        group(DISCREPANCY);
        System.out.println("Frequency table of discrepancies");
        printFreqTable(FREQTABLE, nClasses);
        test(FREQTABLE);
        double criticalValue = criticalValue(DISCREPANCY);
        System.out.println("The expected critical value "
                + "is 1.56");
        System.out.println("Our critical value is = "
                + criticalValue);
        System.out.println("Experimental event = " + XO);
        //Discrepancy due to the    experiment
        double dExp = discrepancy(XO, MEAN, DEVIATION);
        System.out.println("Discrepancy due "
                + " to the experiment = " + dExp);
        if (dExp > criticalValue) {
            System.out.println("Extreme event according to Ho:"
                    + " invent another theory");
        } else {
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }

    }
}//End of Program E121 TestMean
```

122. ***Exercise.*** *Run the program and play with the code to understand how it functions. Mathematicians go exactly over the same procedure but with one plus: they consider an infinite sample at once and so they get exact results we asymptotically tend to.*

123. ***Exercise.*** *High quality software is composed of small units, each of which has been thoroughly tested, piece by piece, and along the hierarchical structure, so subunits, units and the whole ensemble is guaranteed to function. This is in theory. In practice, a thoroughly testing might demand millions of years and so one tries clever testing instead. For instance, the previous program contains a test about the number of considered events. That test is good to find rampant bugs in the grouping method. Apply this philosophy to improve quality of the previous code: modify it so that one can draw in Excel or Calc of LibreOffice the frequency table to see whether or not the distribution fits a Bell of Gauss. Calculate the mean and standard deviation of the set of events and of the frequency table of discrepancies and verify that they must be similar to the expected ones on mathematical grounds.* *Answer*

Our examples allows us to explicitly state the general method that guides modern science:

124. ***General procedure of the scientific method***

The scientific method is perceived distinctly in statistics than in science. This is a cause of confusion, so let us ventilate the differences.

The central idea of the scientific method in science is simple and is the same for all problems no matter how complex they could be. A formulation might be as this (modified from vol 1):

1. One observes the world and extracts a belief.
2. One declares exactly what is the belief that one wants to test: which is the system, which are the laws of its unfolding, and which are the predictions that we must expect under a given set of initial and border conditions (an initial condition is one that applies only at the onset of the experiment, a border condition is one that operates while the experiment is running).
3. One makes an experiment, i.e., one prepares the system in those initial and border conditions listed in the belief, and then one observes the output, the behavior.
4. Because one is unable to control everything, one must repeat the experiment many times to characterize noise.
5. One compares the distribution of experimental results found in the real world with predictions: if the predicted value is an outlier with respect to the distribution of experimental results, one declares that the experiment does not support the belief and that we must look for a more suitable explanation. Otherwise, one declares that the belief explains the experiment

or, more rigorously, one should say: our experiment was unable to reject the belief.

In basic statistics, the simplest caricature of scientific method has the next flavor:

1. We characterize the output of a system as a random variable with a given distribution. We formulate a proposition about its mean or variance or shape. This is the null hypothesis: it is about the global behavior of the system.

2. Over the background of the formulated distribution, our aim is to contrast a specific event or set of events with the null hypothesis.

3. To that aim, we built an artificial, virtual world, with the same characteristics as those proposed by the null hypotheses.

4. One runs many times the virtual world under the same conditions as those declared by the null hypothesis. Mathematicians runs the system infinitely many times but in simulations, we run the system 1000, 10000 or one million times.

5. For each run in the virtual world one measures, always using the same protocol, the discrepancy between each event and that value predicted by the null hypothesis. One registers the distribution of those discrepancies and decides what events are normal and which are outliers. To that aim, one previously defines which is the fraction of events that one must consider as outliers, which is the level of significance. This allows to calculate barriers that divide the normal events or region of acceptance of the null hypothesis from the region of outliers or region of rejection of the null hypothesis.

6. One uses the very same protocol used in the last point to measure the discrepancy between the distinguished event, that possibly came from an experiment of from an observation, and that value predicted by the null hypothesis.

7. If the discrepancy originated by the distinguished event is an outlier with respect to the distribution of the virtual world, one declares that the experiment does not support the null hypotheses. Otherwise, one declares that the null hypothesis explains the distinguished event or, more rigorously, one should say: our data or distinguished event was unable to reject the null hypotheses.

We see that both implementations are very similar. Nevertheless, one makes explicitly clear in the statistical formulation that the null hypothesis is not what one beliefs but some descriptor of the basal theoretical behavior of the system that one studies.

This basic idea can be adapted to many circumstances and so science becomes a matter of creativeness, vision, endurance together with freedom of mind and force of character.

8.2.2 Contrast of the mean with various data

We have seen in the past section how to test a null hypothesis in regard with the mean, say, $\mu = 5$, against one single datum, say $x = 3$, knowing the standard deviation of the population, $\sigma = 0.7$, i.e, the deviation of an infinite data set, which was sampled at random. The next complication is to test a null hypothesis about the mean against a finite number of data, say 3, 4, 5, 3, 5, 5, 4, 3. Mathematicians have solved the problem and their answer is known as the Central Limit Theorem. In its weakest version, that theorem reads:

125. ***The Central Limit Theorem.***

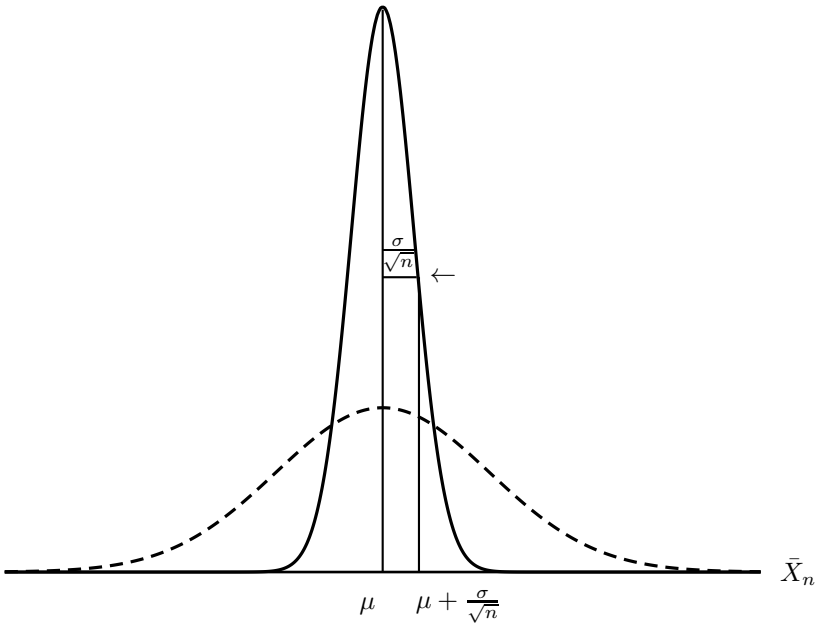


Figure 8.1. Random variable X has a normal distribution with mean $\mu_X = \mu$ and standard deviation $\sigma_X = \sigma$: its density function appears punctured. Random variable $\bar{X}_n$ also has a normal distribution with mean $\mu_{\bar{X}_n} = \mu_X = \mu$, and deviation $\sigma_{\bar{X}_n} = \frac{\sigma_X}{\sqrt{n}}$. The vertical arrow points to the inflexion point. The mean of X is equal to the mean of $\bar{X}_n$ but the degree of dispersion is smaller for the sample mean because a mean averages differences and so sample means tend to get closer to the overall mean.

If $X \sim N(\mu_o, \sigma)$ then $\bar{X}_n \sim N(\mu_o, \sigma/\sqrt{n})$ where $\bar{X}_n$ is the random variable that results from the following procedure: take a random sample with n objects and calculate their mean. Return objects to their population and repeat the same procedure at least 1000 times (one million times would be preferable). With the gotten means, built a frequency table and next a bar char. The resultant envelop must be a Gauss Bell with mean μ and deviation $\sigma/\sqrt{n}$.

This implies that

$\frac{\bar{X}_n - \mu_o}{\sigma/\sqrt{n}} \sim N(\mu = 0, \sigma = 1)$.

In regard with the scientific method, in which we test a null hypothesis against n experimental or observational values, this theorem reads:

1. *Let $H_o : \mu = \mu_o$, which means that an event shall be equal to μ_o apart from the random noise with mean zero and deviation σ. We want to know whether or not the mean is μ_o, so we test the null hypothesis with two tails against a random sample with n events and experimental sample mean $\widehat{\mu}_n$.*

2. *If in a simulation, for each random sample with n events and average $\bar{X}_n$, we calculate the discrepancy d between what we see, $\bar{X}_n$, and what we expect or think, μ_o, according to*

   $d = \frac{\bar{X}_n - \mu_o}{\sigma/\sqrt{n}}$

   then d will have a normal distribution with mean zero and deviation one: $d \sim N(\mu = 0, \sigma = 1)$.

3. *For a confidence level α and two tails, we find z_c, the critical z. For instance, for the default value $\alpha = 0.05$ and two tails, $z_c = 1.96$.*

4. *Let $\widehat{\mu}_n$ be the mean of the experimental sampled values, i.e., an experimental instance of $\bar{X}_n$. To accept else reject the null hypothesis all we need to do is to compare the discrepancy due to $\widehat{\mu}_n$*

   $d(\widehat{\mu}_n) = \frac{\widehat{\mu}_n - \mu_o}{\sigma/\sqrt{n}}$

   with the critical z: if $|d(\widehat{\mu}_n)| \leq z_c$ accept the null hypothesis: the mechanism that caused the mean to get the value μ_o is congruent with experimental observations because the discrepancy between what one sees and what one thinks is small and is explained by the noise. But if $|d(\widehat{\mu}_n)| > z_c$ reject the null hypothesis: the discrepancy between what one sees and what one thinks is large enough to suspect that it is not caused by noise but by a systematic effect to elucidate. In consequence, you are invited to formulate and test another explanation to your results or observations.

More strong versions of this theorem make profit of the fact that $\bar{X}_n$ is a distribution of central tendency that for large n (say, $n > 30$) approaches a normal distribution no matter what distribution X could have, normal or not.

Such astonishing behavior is partially explained by the fact that the mean averages differences, so we expect random sample means to have a distribution of central tendency. But the asymptotic approaching to the normal distribution is explained at another price because a distribution of central tendency must have a very special shape to classify as normal. The ultimate reason might be as follows: the normal distribution is the best continuous fitting to the binomial one, so the central limit theorem says that, in regard with the mean, any distribution can be approximated by a binomial, a fact that is equivalent to say that any system can be digitalized, mimicked by a system run on zeros and ones.

126. ***Example.*** *If the duration of telephone calls of people of certain village follows a normal distribution and has a standard deviation $\sigma = 0.7$ minutes, and 8 calls taken at random have durations in minutes of 3, 4, 5, 3, 5, 5, 4, 3, can we say that the mean duration is 2? (This situation arouse when a telephone adviser was fired because he was unable to have the mean duration of his answers equal to 2 minutes.)*

If we apply the central limit theorem developed by mathematicians, we discover that we have 8 data and that therefore $\bar{X}_8$ has standard deviation $0.7/\sqrt{8} = 0.247$. On the other hand if we take the confidence level as 0.05 then the corresponding $z = 1.96$ because we take two tails since question refers to an equality. So, to test the belief that $\mu = 2$ against observed values 3, 4, 5, 3, 5, 5, 4, 3, we calculate the mean of this sample: $\widehat{\mu} = (3+4+5+3+5+5+4+3)/8 = 4$. Next, we measure the discrepancy between observed value, 4 and predicted value 2:

$$d = \frac{\bar{X}_n - \mu_o}{\sigma/\sqrt{n}} = \frac{\bar{4}-2}{07/\sqrt{8}} = \frac{2}{0.247} = 8.097$$

We see that $8.097 > 1.96$, therefore we conclude that the discrepancy is too large to be explained by random noise alone: we have the right to look for another explanation other than that leading to the expectation $\mu = 2$. In the case of the telephone assistant, the company pretends to battle the sociology of the village, whose people tend to speak to much. It must offer training to employees in regard with managing short and succinct helping, otherwise employees will be unable to fight against the norm of call durations.

Let us solve the same problem by a simulation, whose purpose is to show us mechanistically the origin of the central limit theorem.

127. ***Code to test a null hypothesis about the mean against some experimental values knowing the mean of the population. Let us carefully watch out how we is the scientific method***

```
//Program E127 CLT
//This program contrasts a fact
//against a null hypothesis.
//Fact: a random sample of  X
//3, 4, 5, 3, 5, 5, 4, 3
//has mean 4.
//Null hypothesis:
//X must follow a normal distribution with
//mean = 2
//(knowing that   the deviation of X is  0.7).
//The program contains some tests
//for correctness.

package ejvol5p;

import java.util.Random;

public class CLT {
    //fact = mean of an experimental sample

    private static final double MUHAT = 4;
    //Size of the sample
    private static final int SAMPLESIZE = 8;
    //idea = null hypothesis, expected mean
    private static final double MEAN = 2;
    //restriction
    private static final double XDEVIATION = 0.7;
    //Central limit theorem
    private static final double XBARDEVIATION
            = XDEVIATION / Math.sqrt(SAMPLESIZE);
    //number of random trials
    private static final int NTRIALS = 7435;
    //Significance level
    private static final double ALPHA = 0.05;
    //Turn on of the random generator
    static Random r = new Random();
    //Means of sampleSize
    //numbers generated at random
    private static final double XBARS[] = new double[50000];

    //Discrepancy between fact and idea
    private static final double DISCREPANCY[] = new double[50000];
    private static double min;
    private static double max;
    //The left border of intervals, less or equal to the
```

```
//minimum value of data.
private static double infLimit;
//Interval length
private static double intervalLength;
//Max number of allowed classes
private static final int MAXNCLASSES = 100;
//Borders of the intervals
private static final double BARRIERS[]
        = new double[MAXNCLASSES + 1];
private static final double CLASSMARKERS[]
        = new double[MAXNCLASSES];
private static int nClasses = 20;
//Distribution of the discrepancies
private static final double FREQTABLE[][]
        = new double[MAXNCLASSES][2];

//Generates   sampleSize  random numbers
//with the requested normal distribution
//and calculates the mean
public static double xBar(double mean,
        double deviation,
        int sampleSize) {
    //We generate a random sequence
    //of size equal to that of the sample
    double event;
    double sum = 0;
    double xbar;
    for (int i = 0; i < sampleSize; i++) {
        //Random numbers with Gaussian distribution
        //mean zero and deviation 1.
        double d = r.nextGaussian();
        //System.out.println(d ) ;
        // Rescale numbers: mean = mean  and
        // deviation = deviation
        event = deviation * d + mean;
        //System.out.println(event ) ;
        sum = sum + event;
    }
    xbar = sum / sampleSize;
    //System.out.println(xbar);
    return xbar;
}

//We repeat lim times the following procedure:
//sampleSize random  numbers are generated,
//whose mean is kept in vectSample[].
```

```
//This is a virtual world that obeys the
//null hypothesis.
public static void virtualWorld(double mean,
        double deviation,
        int sampleSize) {
    //We generate a random sequence
    //of size equal to that of the sample
    for (int i = 0; i < NTRIALS; i++) {
        //Random numbers with Gaussian distribution
        double d = xBar(mean, deviation, sampleSize);
        // System.out.println(d ) ;
        XBARS[i] = d;
    }
}

//Measures the discrepancy between an event
//and an idea (the mean)
public static double discrepancy(double event,
        double mean,
        double deviation) {
    double d;
    //Measuring of discrepancy
    d = (event - mean) / deviation;
    return d;
}

//Measures the discrepancy between each fact of
//the virtual world and idea (the mean)
public static void discrepancy(double Events[],
        double mean,
        double xBarDeviation) {
    double d;
    for (int i = 0; i < NTRIALS; i++) {
        //Measuring of discrepancy
        //Measuring of discrepancy
        d = (Events[i] - mean) / xBarDeviation;
        DISCREPANCY[i] = d;
    }
}

//The range of discrepancies is evaluated
//and is used to calculate the length of each
//interval to group Discrepancies.
public static double intervalLength() {
    min = 100000;
    max = 0;
```

```
        for (int i = 0; i < NTRIALS; i++) {
            if (DISCREPANCY[i] < min) {
                min = DISCREPANCY[i];
            }
            if (DISCREPANCY[i] > max) {
                max = DISCREPANCY[i];
            }
        }
        System.out.println("min = " + min);
        System.out.println("max = " + max);
        double range = max - min;
        double length = range / nClasses;
        return length;
    }

    //Prints the frequency table
    private static void printFreqTable(double FreqTable[][],
            int nClasses) {
        System.out.println("x and its absolute frequency");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

    //Calculates the sum    of absolute frequencies
    //of a frequency table
    private static void
            test(double FreqTable[][]) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][1];
        }
        System.out.println("\nNumber of events = " + sum);
    }

//The border of each interval is calculated
    private static void getBorders(double Vect[]) {
        BARRIERS[0] = infLimit;
        CLASSMARKERS[0] = infLimit + intervalLength / 2;
        //Calculate maximal value of data
        double maxValue = Vect[0];
        for (int j = 1; j < Vect.length; j++) {
            if (Vect[j] > maxValue) {
                maxValue = Vect[j];
            }
```

```
        }
        System.out.println("Max value = " + maxValue);
        //Calculates number of classes
        nClasses = (int) ((maxValue - infLimit) / intervalLength) + 1;
        System.out.println("Number of classes = " + nClasses);
        //Calculates interval borders and class markers
        for (int j = 1; j <= nClasses; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + intervalLength;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + intervalLength;
        }
        /*System.out.println( "Borders are" );
 printVector(BARRIERS);
 System.out.println( "Class markers are" );
 printVector(CLASSMARKERS);*/
    }

//Data are grouped into interval classes.
    private static void group(double Vect[]) {
        intervalLength = intervalLength();
        infLimit = min - intervalLength / 2;
        getBorders(Vect);
        int nData = NTRIALS;
        //Class markers a re defined
        for (int j = 0; j < nClasses; j++) {
            FREQTABLE[j][0] = CLASSMARKERS[j];
            FREQTABLE[j][1] = 0;
        }
        //Frequencies are calculated.
        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < nClasses; j++) {
                if ((Vect[i] >= BARRIERS[j])
                        & (Vect[i] < BARRIERS[j + 1])) {
                    FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
                }
            }

        }
    }

    //The critical value with two  tails is calculated
    //with significance level = ALPHA = 0.05
    private static double criticalValue(double Vect[]) {
        double c = 0;
        //Number of events that classify as extreme
        //in either tail
        int e = (int) (NTRIALS * ALPHA / 2);
```

```
        System.out.println("Number of extreme events in "
                + "each tail = " + e);
        //Clon of Vect[]
        double Clon[] = new double[NTRIALS];
        /*
        for (int i = 0; i < NTRIALS; i++) {
        Clon[i] = Vect[i];
        }
         */
        System.arraycopy(Vect, 0, Clon, 0, NTRIALS);
        //Discrepancies are sorted
        //in decreasing order until completion of
        //e events.
        System.out.println("\nDiscrepancies are sorted in "
                + "\ndecreasing order until completion of demanded"
                + "\nnumber of extreme  events.");
        int Champ;
        for (int i = 0; i < e; i++) {
            Champ = 0;
            for (int j = 1; j < NTRIALS; j++) {
                if (Clon[j] >= Clon[Champ]) {
                    Champ = j;
                }
            }

            System.out.println(i + "th ind. is  No "
                    + Champ + " Discrep =  " + Clon[Champ]);
            c = Clon[Champ];
            Clon[Champ] = 0;
        }
        return c;
    }

    /*Test with mean and deviation of events*/
    //Sums data in vector with real numbers
    private static double sumVector(double Vect[],
            int lim) {
        double sum = 0;
        for (int i = 0; i < lim; i++) {
            sum = sum + Vect[i];
        }
        return sum;
    }

//The mean of a list of data
    private static double meanVector(double Vect[],
```

```
            int lim) {
        double sum = sumVector(Vect, lim);
        int n = lim;
        double mean = sum / n;
        return mean;
    }

//The variance of a list of data
    private static double varVector(double Vect[],
            int lim) {
        //Mean
        double mean = meanVector(Vect, lim);
        // Variance is calculated by definition
        int n = lim;
        double sum2 = 0;
        for (int i = 0; i < n; i++) {
            sum2 = sum2 + (Vect[i] - mean) * (Vect[i] - mean);
        }
        double var = sum2 / (n - 1);
        return var;
    }

//The mean and variance of a list of data
    private static void meanVarVector(double DataVect[],
            int lim) {
        System.out.println("Sampled data");
        System.out.println("Data length = " + lim);
        double SampleMean = meanVector(DataVect, lim);
        System.out.println("Mean = " + SampleMean);
        double SampleVar = varVector(DataVect, lim);
        System.out.println("Variance \t " + SampleVar);
        double SampleDeviation = Math.pow(SampleVar, 0.5);
        System.out.println("Deviation = " + SampleDeviation);
        double coeffVar = SampleDeviation / SampleMean;
        System.out.println("Coefficient of variation = "
                + coeffVar);
        System.out.println("Expected mean = " + MEAN);
        System.out.println("Expected deviation = " + XBARDEVIATION);
    }

    /*Test with mean and deviation of frequency table*/
//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][],
            int nClasses) {
        double n = 0;
```

```
        for (int i = 0; i < nClasses; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

//Calculates the sum XF   of
//a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the sum X2F   of
//a table of absolute frequencies
    private static double
            sumX2FFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][0]
                    * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][],
            int nClasses) {
        double n = nFreqTable(FreqTable, nClasses);
        double mean = sumXFFreqTable(FreqTable, nClasses) / n;
        return mean;
    }

//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][],
            int nClasses) {

        double n = nFreqTable(FreqTable, nClasses);
```

```
        double Sxx = sumX2FFreqTable(FreqTable, nClasses)
                - Math.pow(sumXFFreqTable(FreqTable, nClasses), 2) / n;
        double var = Sxx / (n - 1);
        return var;
    }

// Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][],
            int nClasses) {
        double var = varFreqTable(FreqTable, nClasses);
        double dev = Math.pow(var, 0.5);
        return dev;
    }

// Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][],
            int nClasses) {
        double n = nFreqTable(FreqTable, nClasses);
        System.out.println("Number of measured objects = " + n);
        double sumXF = sumXFFreqTable(FreqTable, nClasses);
        System.out.println("sum xF  = " + sumXF);
        double mean = meanFreqTable(FreqTable, nClasses);
        System.out.println("Mean = " + mean);
        double sumX2F = sumX2FFreqTable(FreqTable, nClasses);
        System.out.println("sum x2F  = " + sumX2F);
        double var = varFreqTable(FreqTable, nClasses);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable, nClasses);
        System.out.println("Deviation =  " + deviation);
    }

//Test: grouping must produce a Gauss'Bell
    private static void tableForExcel() {
        System.out.println("\nPaste next numbers to Excel, "
                + "LibreOffice or Gnumeric."
                + "\nMake a bar char."
                + "\nA Gauss'bell shall appear else there is a bug.");
        System.out.println("Absolute frequency of discrepancies:");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FREQTABLE[i][1]);
        }
    }

    private static void testZone() {
        System.out.println("\n\nTest zone");
```

```
        //Mean and variance of events
        System.out.println("\nMean and Variance of events:");
        meanVarVector(XBARS, NTRIALS);
        //Insert here a z-test for a mean and
        //a chi-square test for the variance.
        System.out.println("\nThe table of absolute "
                + "frequencies of discrepancies"
                + " \nmust represent"
                + " a normal distribution \nwith mean zero"
                + " and standard deviation one.");
        //Mean and variance of frequency table
        System.out.println("\nMean and Variance of Frequency"
                + " table of discrepancies: \nExpected values: \n"
                + "mean = 0 \nDeviation =   1");
        meanVarTable(FREQTABLE, nClasses);
        //Insert here a z-test for a mean and
        //a chi-square test for the variance.
        //Prints the absolute frequencies of discrepancies
        //they can be pasted to Excel, LibreOffice or Gnumeric
        tableForExcel();
        //Insert here a test for normality
    }

    public static void main(String[] args) {
        virtualWorld(MEAN, XDEVIATION, SAMPLESIZE);
        System.out.println("\nDiscrepancies");
        discrepancy(XBARS, MEAN, XBARDEVIATION);
        group(DISCREPANCY);
        System.out.println("Frequency table of discrepancies");
        printFreqTable(FREQTABLE, nClasses);
        //Minor test for correct grouping
        test(FREQTABLE);

        double criticalValue = criticalValue(DISCREPANCY);
        //Test for critical value
        System.out.println("The expected critical value "
                + "is 1.96");
        System.out.println("Found upper critical value  = "
                + criticalValue);
        System.out.println("Experimental event = " + MUHAT);
        //Discrepancy due to the    experiment
        double dExp = discrepancy(MUHAT, MEAN, XBARDEVIATION);
        System.out.println("Discrepancy due  to the "
                + "experimental event = " + dExp);
        if (dExp > criticalValue) {
            System.out.println("Extreme event according to Ho:"
```

```
                    + " invent another theory");
        } else {
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }
        testZone();
    }
}//End of Program E127 CLT
```

128. ***Exercise.*** *Run the program and play with the code.*

8.2.3 Contrast with various data: the t distribution

In the central limit theorem, the measure of discrepancy between the expected mean μ_o and the experimentally found value $\widehat{\mu}_n$ was calculated using σ, the deviation of X, which can be understood as a sample mean of a random sample of infinite size. The question now is what happens when instead of σ we take s, the deviation of the sample, with only n events? The mathematical answer to this question comes in the form of a t-test that uses a t distribution instead of a z. Since the t-test works with a deviation obtained from a finite number of data, it is expected to be more imprecise than the central limit theorem. In consequence, the t-distribution looks as a z distribution but with a larger deviation. Actually the deviation of the t density function is $\sqrt{(\frac{\nu}{\nu-2})}$, where ν represents the number of degrees of freedom, so for 8 data, we have 7 degrees of freedom and the deviation of the corresponding t-distribution is 1.18.

129. ***The t-theorem.*** *Let X be a random variable, not necessarily with a normal distribution. Let $H_o : \mu = \mu_o$, which means that an event shall be equal to μ_o apart from the random noise with mean zero and deviation σ. To test the null hypothesis, we take an experimental sample with n events, mean $\widehat{\mu}_n$ and deviation s. Let us test the null hypothesis with two tails, which means that all we want to know is whether or not the mean is μ_o. In a simulation, we take a random sample of n events and calculate its mean $\bar{X}_n$ and its deviation s. Next, we calculate the discrepancy d between what we see, $\bar{X}_n$, and what we expect or think, μ_o:*

$$d = \frac{\bar{X}_n - \mu_o}{s/\sqrt{n}}$$

then, d will have a t-distribution with $n-1$ degrees of freedom, whose density function is given by

$$f_t(x) = \frac{\Gamma[(\nu+1)/2]}{\Gamma(\nu/2)\sqrt{\nu\pi}}(1 + \frac{x^2}{\nu})^{-\frac{\nu+1}{2}}$$

where the Γ function is defined by

$$\Gamma(x/2) = \begin{cases} (x/2-1)(x/2-2)...3 \times 2 \times 1 \ if \ x \ is \ even. \\ (x/2-1)(x/2-2)...3/2 \times 1/2 \times \sqrt{\pi} \ if \ x \ is \ odd. \end{cases}$$

The mean of the t distribution is zero and the variance is $nu/(nu-2)$ *with* $\nu = n-1$.

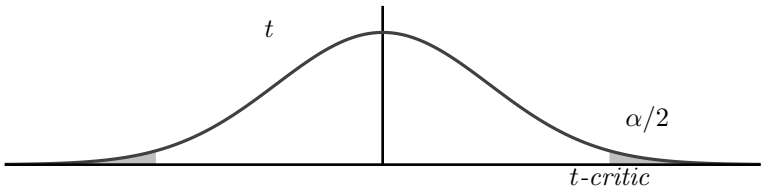


Figura 8.2. The density function of a t looks like a Bell of Gauss although more extended. As the degrees of freedom grow, the t approaches the z. For two tails and for any significance level there are two critical values.

For a confidence level α *and two tails, we find* t_c*, the critical* t*. For instance, for the default value* $\alpha = 0.05$*, two tails and 7 degrees of freedom,* $t_c = 2.365$*. Now, let* $\widehat{\mu}_n$ *be the mean of the sampled values in an experiment, i.e., and instance of* $\bar{X}_n$*. To accept else reject the null hypothesis all we need to do is to compare the discrepancy due to* $\widehat{\mu}_n$

$$d(\widehat{\mu}_n) = \frac{\widehat{\mu}_n - \mu_o}{\sigma/\sqrt{n}}$$

with the critical t*: if* $|d(\widehat{\mu}_n)| \leq t_c$ *accept the null hypothesis: the mechanism that caused the mean to get the value* μ_o *is congruent with experimental observations because the discrepancy between what one sees and what one thinks is small and is explained by the noise. But if* $|d(\widehat{\mu}_n)| > t_c$ *reject the null hypothesis: the discrepancy between what one sees and what one thinks is large enough to suspect that it is not caused by noise but by a systematic effect to elucidate. In consequence, you are invited to formulate and test another explanation to your results or observations.*

130. ***Exercise.*** *Adapt the previous code, that was developed for the central limit theorem, to simulate a t-test, as explained in the t-theorem.* *Answer*

8.3 Hypotheses about variances

The **sample mean** (the mean of a sample) is the fundamental descriptor of a data set in the sense that it maximizes the information content using the minimum of resources. It is also the best estimator of the mean of a source of random data. Now, let us ask: what else shall be added to the mean to maximize the marginal information at minimum cost? The answer is: if we replace the whole data by the mean, the degree of fairness in the representation is not the same for all data, so a measure of the discrepancy between the mean and data is what we shall add to the mean to enrich the information to a maximum at minimum cost. That measure of dissatisfaction is tied to the variability of

the population and is, of course, the variance or its square root, the standard deviation.

It is now understandable why the mean without the variance looks lonesome and poor like the left rail without the right one.

Lets us study now a null hypotheses in regard with one and two variances. In the first case, we have the chi-square-theorem and, in the second, the F-theorem.

8.3.1 One variance: the chi-square theorem

Let us consider a null hypothesis in which we declare that the variance of certain random variable with normal distribution has certain value.

131. ***The chi-square theorem.*** *Let X be a normally distributed random variable. We have a null hypothesis about its variance σ^2, which reads $H_o : \sigma^2 = \sigma_o^2$ and an alternate hypothesis $H_a : \sigma^2 \neq \sigma_o^2$, i.e., we have a two tailed test. Given a confidence level α, that null hypothesis must be tested against a set of n data whose mean is $\widehat{\mu}_n = \bar{X}_n$ and sample variance*

$$\widehat{\sigma}^2 = (\tfrac{1}{n-1}) \sum_1^n (x_i - \widehat{\mu}_n)^2 = s^2$$

where x_i represents datum number i-th and $\widehat{\mu}_n$ is the mean of the sample.

To decide the null hypothesis, we must go over the next procedure: Take in a simulation a random sample of $X \sim N(\mu = \widehat{\mu}_n; \sigma = \sigma_o)$ with n elements and compute its mean $\widehat{\mu}_n$ and variance $\widehat{\sigma}^2$. This sample is an observation in a virtual world that behaves just as commanded by the null hypothesis. Next, we measure the discrepancy between what we observe in the virtual world and what we think. This is done by

$$d = (n-1)\frac{\widehat{\sigma}^2}{\sigma_o^2}$$

This discrepancy is a random variable that distributes itself as a chi-square distribution whose probability density function is given by

$$f_{\chi^2}(x) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} e^{-\frac{x}{2}}\, x^{\frac{\nu-2}{2}} \text{ for } x \geq 0.$$

The mean of a chi-square distribution is $\nu = n - 1$ and the variance is 2ν.

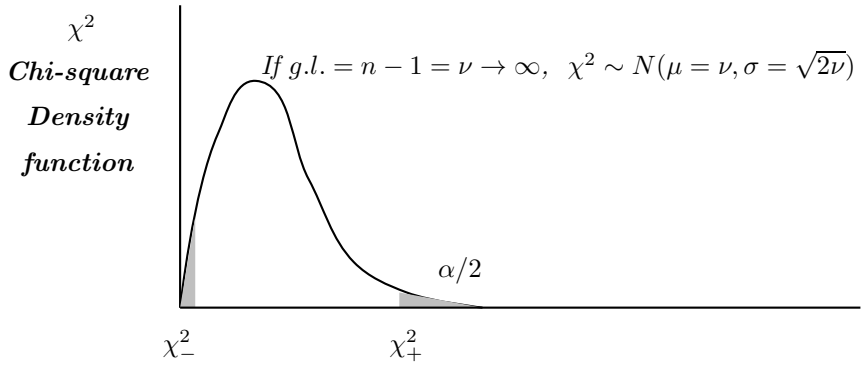


Figure 8.3. The chi-square deals with variances, positive quantities, so it is not defined for negative values.

In plain words, if we repeat the aforementioned procedure some 10000 or more times and make a char bar of d values, the envelope will get a shape that is best fitted by the density function of the chi-square distribution.

For the chosen significance level, compute the critical values, χ^2_- and χ^2_+. This barriers tell us what must be understood as normal and what as outlier. If certain d is in within these two values, it is normal, otherwise it is an outlier. To decide the null hypothesis, we must calculate d_{exp}, the discrepancy originated by the experimental data, whose sample variance is $\widehat{\sigma}^2$:

$$d_{exp} = (n-1)\frac{\widehat{\sigma}^2}{\sigma_o^2}$$

If d_{exp} is a normal value in the virtual world delimited by χ^2_- and χ^2_+, accept the null hypothesis: the discrepancy produced by observed data can be explained by the noise that is inherent to X and that is reflected by σ_o^2 and so the null hypothesis is an acceptable explanation of observed data. But if the discrepancy d_{exp} is classified as outlier, then it is large enough to invite a revision of beliefs: think of a new one and test it on its own.

132. ***Exercise.*** *Compose a code to simulate a χ-square-test as explained in the chi-square-theorem and to show mechanistically the origin of the χ square distribution.* *Answer*

8.3.2 Two variances: the F theorem

Let us consider a null hypothesis in which we assign a value to the ratio between two variances of two given random variables with normal distributions.

133. ***The F-theorem.*** *Let X and Y be normally distributed random variables with variances σ_1^2 and σ_2^2 respectively, which are unknown. We have a null hypothesis about the relation between variances, which reads*

$$H_o : \frac{\sigma_1^2}{\sigma_2^2} = R$$

and an alternate hypothesis

$H_a : \frac{\sigma_1^2}{\sigma_2^2} \neq R$

i.e., we have a two tailed test. Given a confidence level α*, that null hypothesis must be tested against a set of* n_1 *data sampled from* X *and* n_2 *data sampled from* Y*. The means of those samples are* $\widehat{\mu}_1$ *and* $\widehat{\mu}_2$*, while sample variances of those data are respectively* $\widehat{\sigma}_1^2$ *and* $\widehat{\sigma}_2^2$*.*

Let us define the joint deviation $\widehat{\sigma_J}$ *as*

$\widehat{\sigma_J} = \sqrt{\frac{\widehat{\sigma_1}^2(n_1-1)+\widehat{\sigma_2}^2(n_2-1)}{n_1+n_2-2}}$

To decide the null hypothesis, we must go over the next procedure: take in a simulation a random sample of $X \sim N(\mu = \widehat{\mu}_1; \sigma = \sqrt{R}\widehat{\sigma_J})$ *with* n_1 *elements and another random sample of* $Y \sim N(\mu = \widehat{\mu}_2; \sigma = \widehat{\sigma_J})$ *with* n_2 *elements, compute sample variance* $\widehat{\sigma}_1^2 = s_1^2$ *and* $\widehat{\sigma}_2^2 = s_2^2$*. These samples conform an observation in a virtual world that behaves just as commanded by the null hypothesis. Next, we measure the relation between these two variances:*

$r = \frac{s_1^2}{s_2^2}$

The discrepancy between what we observe in the virtual world and what we think is given by

$d = \frac{r}{R}$

This discrepancy is a random variable that distributes itself as a F *distribution whose probability density function is given by*

$f_F(x) = \frac{\Gamma(\frac{\nu_1+\nu_2}{2})}{\Gamma(\frac{\nu_1}{2})\Gamma(\frac{\nu_2}{2})} \nu_1^{\frac{\nu_1}{2}} \nu_2^{\frac{\nu_2}{2}} \frac{x^{\frac{\nu_1-2}{2}}}{(\nu_2+\nu_1 x)^{\frac{\nu_1+\nu_2}{2}}}$ *for* $x \geq 0$*.*

where Γ *is the gamma function defined in the chi-square theorem, the degrees of freedom are* $\nu_1 = n_1 - 1$ *and* $\nu_2 = n_2 - 1$*.*

The mean of the F *distribution is, for* $\nu_2 > 2$*,*

$Mean = \frac{\nu_2}{\nu_2-2}$

and the variance is, for $\nu_2 > 4$*,*

$Variance = (\frac{\nu_2}{\nu_2-2})^2 \times \frac{2(\nu_1+\nu_2-2)}{\nu_1(\nu_2-4)}$

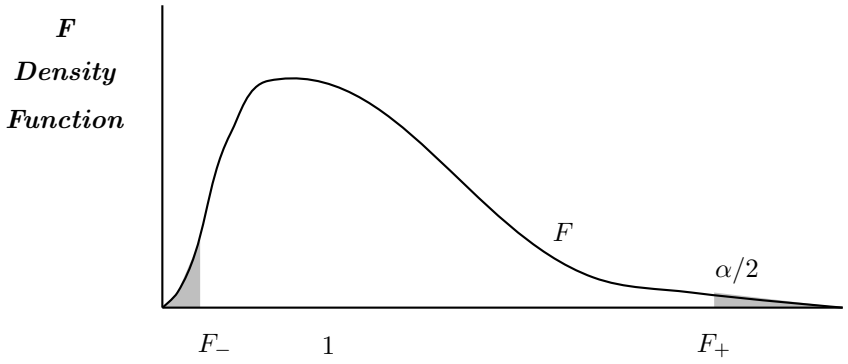


Figure 8.4. General shape of a F density function. For each α and two tails, there are two critical values, F_- and F_+. The mean is close to the mode that is close to one.

In plain words, if we repeat the aforementioned procedure some 10000 or more times and make a char bar of d values, the envelope will get a shape that is best fitted by the density function of the F distribution.

For the chosen significance level, compute the critical values, F_- and F_+. These barriers tell us what must be understood as normal and what as outlier. If certain d is in within these two values, it is normal, otherwise it is an outlier. To decide the null hypothesis, we must calculate d_{exp} the discrepancy between experimental data and expected value according to the considered null hypothesis, whose sample variances are $\widehat{\sigma}_1^2$ and $\widehat{\sigma}_1^2$:

$$r_{exp} = \frac{\widehat{\sigma}_1^2}{\widehat{\sigma}_2^2}$$

Variables are numbered in such a way that $r_{exp} \geq 1$. We compare now the relation of experimental variances with the expected ratio R to get the discrepancy between observed and predicted by the null hypothesis.

$$d_{exp} = \frac{r_{exp}}{R}$$

If d_{exp} is a normal value in the virtual world delimited by F_- and F_+, accept the null hypothesis: the discrepancy produced by observed data can be explained by the noise that is inherent to X and Y and that is reflected by σ_1^2 and σ_2^2 so the null hypothesis is an acceptable explanation of observed data. But if the discrepancy d_{exp} is classified as outlier, then it is large enough to invite a revision of beliefs: think of a new one and test it on its own.

134. ***Exercise.*** *Compose a code to simulate an F-test as explained in the F theorem and to show mechanistically the origin of the F distribution.* Answer

8.3.3 Experiments with two proportions

There was a time when developing software was a distinctive of male intelligence, an option that was inaccessible to females. That was taken by everybody, females included, as a definitive demonstration of the fact that males were superior for the extremely difficult task of software developing. Nevertheless, things are otherwise by these days: some months ago, I visited a company in the trade and I noticed that ladies and gentlemen work shoulder to shoulder with significant proportion of both genders. The difference is now not among male and female but about formation: those that have a professional formation in mathematics were expected to perform better than pure engineers in regard with how to attack very difficult problems. Anyway, once it was known how to solve a problem, engineers are probably better in the implementation of the solution. All these problems are related with proportions and with the statistical problem of deciding when two proportions are equal given scare data. The next is a usual solution, expressed in short words, to this problem:

135. ***Receipt for the comparison of two proportions.***

Test with two tails : $H_o : p - r = s$, $H_a : p - r \neq s$

where p and r are the proportions of trait carriers in two populations given that the corresponding random samples of m and n individuals were f and g. Use the Z distribution with

$$z = \frac{(f-g)-(p-r)}{\sqrt{\frac{p(1-p)}{m}+\frac{r(1-r)}{n}}}$$

136. ***Challenge.*** *Convert the previous receipt into a theorem about a null hypothesis and implement it into a Java program.*

Having gone over simple mandatory exercises about the very core of the scientific method, one would like to give a look to more complex problems. We have chosen various instances, among which we have a test for the comparison of two means and a test for normality.

8.4 Comparison of two means

In a productivity experiment, one compares the average yield of two processes that differentiate one from another by a given factor. Example: two groups learn to program Java, the first uses Vol 1 and the second uses Vol 5. An exam is given to them taking care of choosing themes common to both groups. We compare the mean score to see what material produces better average results. The official and correct way of solving this question appears in various theorems, some of which follow.

8.4.1 Known variances

137. *Theorem.* *Let X and Y be random variables normally distributed with means μ_X and μ_Y and deviations σ_X and σ_Y. Take a random sample of m data from $X \sim N(\mu_x, \sigma_X)$ and a random sample of n data from $Y \sim N(\mu_Y, \sigma_Y)$ and find the corresponding means $\bar{X}_m$ and $\bar{Y}_n$. Let W be the random variable $W = \bar{X}_m - \bar{Y}_n$. Then W has a normal distribution with mean equal to $\mu_X - \mu_Y$ and deviation given by*

$$\sigma_W = \sqrt{\frac{{\sigma_X}^2}{m} + \frac{{\sigma_Y}^2}{n}}$$

In consequence, we have the following receipt to test a null hypothesis about the difference of means, i.e., about the mean of W. Let $H_o : \mu_X - \mu_Y = k$, a belief that must be tested with two tails and significance level α against experimental observations that rendered a mean of X equal to $\widehat{X}_m$ taken from m data and mean of Y equal to $\widehat{Y}_n$ taken from n data.

To test the null hypothesis, take in a simulation a random sample of m data from $X \sim N(\mu = \widehat{Y}_m + k, \sigma = \sigma_X)$ and a random sample of n data from $Y \sim N(\mu = \widehat{Y}_m, \sigma = \sigma_Y)$ and find the corresponding means $\bar{X}_m$ and $\bar{Y}_n$. This is a virtual world that behaves just as commanded by the null hypothesis. Measure the discrepancy between what is expected according to the null hypothesis and what is found in experiment in this virtual world by

$$d = \frac{(\bar{X_m} - \bar{Y_n}) - (\mu_X - \mu_Y)}{\sqrt{\frac{{\sigma_X}^2}{m} + \frac{{\sigma_Y}^2}{n}}}$$

then d has a Z-distribution, i.e., a normal distribution with mean 0 and deviation 1. So, find the critical values of Z for the given significance level. Let them be z_- and z_+. Next, we calculate the discrepancy due to the experiment:

$$d_{exp} = \frac{(\widehat{X_m} - \widehat{Y_n}) - (\mu_X - \mu_Y)}{\sqrt{\frac{{\sigma_X}^2}{m} + \frac{{\sigma_Y}^2}{n}}}$$

Accept the null hypothesis if d_{exp}, the discrepancy associated to the experiment, is in within those bounds, else reject it.

Apply this theorem with confidence even if your data does not correspond to normal distributions.

138. *Challenge.* *Develop a simulation to show mechanistically and automatically how the previous theorem functions.*

8.4.2 Unequal inferred variances

When one does not know the value of the variances of the populations, one must estimate them from data. There are many ways of doing that and it is preferable to use a method extracting the maximal profit with the maximal transparency.

For this reason, one begins the analysis with an F-test to decide whether or not the variances are equal. Let us consider the case in which the F-test renders that the variances are unequal.

139. ***Theorem.*** *Let X and Y be random variables normally distributed with unknown means μ_X and μ_Y and unknown deviations σ_X and σ_Y. Let H_o : $\mu_X - \mu_Y = k$, a belief that must be tested with two tails and significance level α against experimental observations that rendered a mean of X equal to $\widehat{X}_m$ taken from m data and mean of Y equal to $\widehat{Y}_n$ taken from n data. From the same experimental data, we have the sample deviations $\widehat{\sigma_X}$ and $\widehat{\sigma_Y}$ and a previously used F-test rendered different populational variances.*

To test the null hypothesis, take a random sample of m data from $X \sim N(\mu = \widehat{Y}_m + k, \sigma = \widehat{\sigma}_X)$ and a random sample of n data from $Y \sim N(\mu = \widehat{Y}_m, \sigma = \widehat{\sigma}_Y)$ and find the corresponding means $\bar{X}_m$ and $\bar{Y}_n$. Find also the sample variances s_X^2 and s_Y^2. This is a virtual world that behaves just as commanded by the null hypothesis. Measure the discrepancy between what is expected according to the null hypothesis and what is found in experiment in this virtual world by

$$d = \frac{(\bar{X_m} - \bar{Y_n}) - (\mu_X - \mu_Y)}{\sqrt{\frac{{s_X}^2}{m} + \frac{{s_Y}^2}{n}}}$$

then d has a t-distribution with degrees of freedom df given by

$$df = \frac{(\frac{{s_X}^2}{n_X} + \frac{{s_Y}^2}{n_Y})^2}{\frac{(\frac{{s_X}^2}{n_X})^2}{n_X+1} + \frac{(\frac{{s_Y}^2}{n_Y})^2}{n_Y+1}} - 2$$

So, find the critical values of t for the given significance level and the corresponding degrees of freedom. Let them be t_- and t_+. Next, we calculate the discrepancy due to the experiment:

$$d_{exp} = \frac{(\widehat{X}_m - \widehat{Y}_n) - (\mu_X - \mu_Y)}{\sqrt{\frac{\widehat{\sigma_X}^2}{m} + \frac{\widehat{\sigma_Y}^2}{n}}}$$

Accept the null hypothesis if d_{exp}, the discrepancy associated to the experiment, is normal in the virtual world. i.e., if it is in within t_- and t_+, else reject it.

140. ***The corresponding code follows.***

```
//Program E140 TwoMeans
//This program contrasts   a  null hypothesis
//about the difference between  two means
//against experimental facts.
```

```
//Specifically,
//the null hypothesis  is
//that the difference between
//populational means is 5.
//Facts are:
//the set of random data  of X
//1, 2, 5, 8, 7, 9, 8, 16, 14
//with mean 7.7 and variance 24.4;
//the set of random data Y
//3, 4, 5, 3, 5, 5, 4, 3
//with mean 4 and variance 0.85;
//the relation of variances is
//24.4 / 0.85 = 28.8.
//Help given by an F-test: Variances are different.
//The program contains some tests
//for correctness.

package ejvol5p;

import java.util.Random;

public class twoMeans {
    //fact =   experimental samples

    private static final double DATA1[]
            = {1, 2, 5, 8, 7, 9, 8, 16, 14};
    private static final double DATA2[]
            = {3, 4, 5, 3, 5, 5, 4, 3};

    //Means and deviations of experimental samples
    private static double muHat1, sigmaHat1, varHat1;
    private static double muHat2, sigmaHat2, varHat2;
    //Size of  samples DATA1 and DATA2
    private static final int SAMPLESIZE1 = 9;
    private static final int SAMPLESIZE2 = 8;
    //Degrees of freedom
    private static double nu;
    //idea = null hypothesis,
    //expected difference between means
    private static final double MU1MINUSMU2 = 5;
    //Discrepancy due to experiment
    private static double dExp;
    //Number of trials in the virtual world
    private static final int NTRIALS = 74350;
    //Significance level
    private static final double ALPHA = 0.05;
```

```
//Turn on of the random generator
static Random r = new Random();
//A list with a random sample
private static final double VECTSAMPLE[]
        = new double[1000];
//Dimension of arrays
private static final int N = 100000;
//Means   of samples   generated at random
private static final double XBARS1[] = new double[N];
private static final double XBARS2[] = new double[N];
//Variances of samples  generated at random
private static final double VAR1[] = new double[N];
private static final double VAR2[] = new double[N];
//Discrepancy between fact and idea
private static final double DISCREPANCY[] = new double[N];
private static double min;
private static double max;
//The left border of intervals, less or equal to the
//minimum value of data.
private static double infLimit;
//Interval length
private static double intervalLength;
//Max number of allowed classes
private static final int MAXNCLASSES = 1000;
//Borders of the intervals
private static final double BARRIERS[]
        = new double[MAXNCLASSES + 1];
private static final double CLASSMARKERS[]
        = new double[MAXNCLASSES];
private static int nClasses = 250;
//Distribution of the discrepancies
private static final double FREQTABLE[][]
        = new double[MAXNCLASSES][2];

//Sums data in vector with real numbers
private static double sumVector(double Vect[],
        int lim) {
    double sum = 0;
    for (int i = 0; i < lim; i++) {
        sum = sum + Vect[i];
    }
    return sum;
}

//The mean of a list of data
private static double meanVector(double Vect[],
```

```
        int lim) {
    double sum = sumVector(Vect, lim);
    int n = lim;
    double mean = sum / n;
    return mean;
}

//The variance of a list of data
private static double varVector(double Vect[],
        int lim) {
    //Mean
    double mean = meanVector(Vect, lim);
    // Variance is calculated by definition
    int n = lim;
    double sum2 = 0;
    for (int i = 0; i < n; i++) {
        sum2 = sum2 + (Vect[i] - mean) * (Vect[i] - mean);
    }
    double var = sum2 / (n - 1);
    return var;
}

//Generates   sampleSize  random numbers
//with the requested normal distribution.
public static void randomSample(double mean,
        double deviation,
        int sampleSize) {
    //We generate a random sequence
    //of size equal to that of the sample
    double event;
    for (int i = 0; i < sampleSize; i++) {
        //Random numbers with Gaussian distribution
        //mean zero and deviation 1.
        double d = r.nextGaussian();
        //System.out.println(d ) ;
        // Rescale numbers: mean = mean  and
        // deviation = deviation
        event = deviation * d + mean;
        //System.out.println(event ) ;
        VECTSAMPLE[i] = event;
    }
}

//We repeat lim times the following procedure:
//sampleSize random  numbers are generated,
//whose mean and deviation are kept in simXBar[]
```

```
    //and simXS[]. This is done twice for we
    //have two populations.
    //This is a virtual world that obeys the
    //null hypothesis.
    public static void virtualWorld() {
        //We run the virtual world NTRIALS times
        for (int i = 0; i < NTRIALS; i++) {
            //SAMPLESIZE1 random numbers with normal distribution
            randomSample(muHat2 + MU1MINUSMU2, sigmaHat1,
                    SAMPLESIZE1);
            //mean of sample
            double xBar = meanVector(VECTSAMPLE, SAMPLESIZE1);
            //variance
            double xVariance = varVector(VECTSAMPLE, SAMPLESIZE1);
            XBARS1[i] = xBar;
            VAR1[i] = xVariance;
            //SAMPLESIZE2 random numbers with normal distribution
            //Variance2 = R times variance1
            randomSample(muHat2, sigmaHat2, SAMPLESIZE2);
            //mean of sample
            xBar = meanVector(VECTSAMPLE, SAMPLESIZE2);
            //variance
            xVariance = varVector(VECTSAMPLE, SAMPLESIZE2);
            XBARS2[i] = xBar;
            VAR2[i] = xVariance;
        }
    }

    //Measures the discrepancy between observed r
    //and expected R
    public static double discrepancy(double mean1,
            double var1, double mean2, double var2) {
        double deno = Math.sqrt(var1 / SAMPLESIZE1
                + var2 / SAMPLESIZE2);
        double d
                = ((mean1 - mean2) - (MU1MINUSMU2)) / deno;
        return d;
    }

    //Measures the discrepancy between each fact of
    //the virtual world and the idea
    public static void discrepancy(double simVar1[],
            double simVar2[]) {
        double d;
        for (int i = 0; i < NTRIALS; i++) {
            //Measuring of discrepancy
```

```
            //between observed r and thought R
            d = discrepancy(XBARS1[i], simVar1[i],
                    XBARS2[i], simVar2[i]);
            DISCREPANCY[i] = d;
        }
    }

    //The range of discrepancies is evaluated
    //and is used to calculate the length of each
    //interval to group Discrepancies.
    public static double intervalLength() {
        min = 100000;
        max = 0;
        for (int i = 0; i < NTRIALS; i++) {
            if (DISCREPANCY[i] < min) {
                min = DISCREPANCY[i];
            }
            if (DISCREPANCY[i] > max) {
                max = DISCREPANCY[i];
            }
        }
        System.out.println("min = " + min);
        System.out.println("max = " + max);
        double range = max - min;
        double length = range / nClasses;
        return length;
    }

    //Prints the frequency table
    private static void printFreqTable(double FreqTable[][],
            int nClasses) {
        System.out.println("x and its absolute frequency");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

    //Calculates the sum    of absolute frequencies
    //of a frequency table
    private static void
            test(double FreqTable[][]) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][1];
        }
```

```
        System.out.println("\nNumber of events = " + sum);
    }

//The border of each interval is calculated
    private static void getBorders(double Vect[]) {
        BARRIERS[0] = infLimit;
        CLASSMARKERS[0] = infLimit + intervalLength / 2;
        //Calculate maximal value of data
        double myMax = Vect[0];
        for (int j = 1; j < Vect.length; j++) {
            if (Vect[j] > myMax) {
                myMax = Vect[j];
            }
        }
        System.out.println("Max value = " + myMax);
        //Calculates number of classes
        nClasses = (int) ((myMax - infLimit) / intervalLength) + 1;
        System.out.println("Number of classes = " + nClasses);
        //Calculates interval borders and class markers
        for (int j = 1; j <= nClasses; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + intervalLength;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + intervalLength;
        }
        /*System.out.println( "Borders are" );
 printVector(BARRIERS);
 System.out.println( "Class markers are" );
 printVector(CLASSMARKERS);*/
    }

//Data are grouped into interval classes.
    private static void group(double Vect[]) {
        intervalLength = intervalLength();
        infLimit = min - intervalLength / 2;
        getBorders(Vect);
        int nData = NTRIALS;
        //Class markers a re defined
        for (int j = 0; j < nClasses; j++) {
            FREQTABLE[j][0] = CLASSMARKERS[j];
            FREQTABLE[j][1] = 0;
        }
        //Frequencies are calculated.
        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < nClasses; j++) {
                if ((BARRIERS[j] <= Vect[i])
                        & (Vect[i] < BARRIERS[j + 1])) {
                    FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
```

```
            }
        }

    }
}

//The upper critical value with two  tails is calculated
//with significance level = ALPHA = 0.05
private static double criticalValueUp(double Vect[]) {
    double c = 0;
    //Number of events that classify as extreme
    //in either tail
    int e = (int) (NTRIALS * ALPHA / 2);
    System.out.println("Number of extreme events in "
            + "each tail = " + e);
    //Clon of Vect[]
    double Clon[] = new double[NTRIALS];
    /*
    for (int i = 0; i < NTRIALS; i++) {
    Clon[i] = Vect[i];
     */
    System.arraycopy(Vect, 0, Clon, 0, NTRIALS);
    //Discrepancies are sorted
    //in decreasing order until completion of
    //e events.
    int Champ;
    for (int i = 0; i < e; i++) {
        Champ = 0;
        for (int j = 1; j < NTRIALS; j++) {
            if (Clon[j] >= Clon[Champ]) {
                Champ = j;
            }
        }

        /* System.out.println(  i + "th ind. is  No "
             + Champ + " Discrep =  " + Clon[Champ]);*/
        c = Clon[Champ];
        Clon[Champ] = 0;
    }
    return c;
}

//The lower critical value with two  tails is calculated
//with significance level = ALPHA = 0.05
private static double criticalValueDown(double Vect[]) {
    double c = 0;
```

```
        //Number of events that classify as extreme
        //in either tail
        int e = (int) (NTRIALS * ALPHA / 2);
        //Clon of Vect[]
        double Clon[] = new double[NTRIALS];
        /*
        for (int i = 0; i < NTRIALS; i++) {
        Clon[i] = Vect[i];
        }
         */
        System.arraycopy(Vect, 0, Clon, 0, NTRIALS);
        //Discrepancies are sorted
        //in increasing order until completion of
        //e events.
        System.out.println("\nDiscrepancies are sorted in "
                + "\ndecreasing order until completion of demanded"
                + "\nnumber of extreme  events.");
        int Champ;
        for (int i = 0; i < e; i++) {
            Champ = 0;
            for (int j = 1; j < NTRIALS; j++) {
                if (Clon[j] <= Clon[Champ]) {
                    Champ = j;
                }
            }
            /* Instruction to print outliers
   System.out.println(  i + "th ind. is  No "
                 + Champ + " Discrep =  " + Clon[Champ]);*/
            c = Clon[Champ];
            Clon[Champ] = 10000000;
        }
        return c;
    }

    /*Test with mean and deviation of events*/
//The mean and variance of a list of data
    private static void meanVarVector(double DataVect[],
            int lim) {
        System.out.println("Sampled data");
        System.out.println("Data length = " + lim);
        double SampleMean = meanVector(DataVect, lim);
        System.out.println("Mean = " + SampleMean);
        double SampleVar = varVector(DataVect, lim);
        System.out.println("Variance = " + SampleVar);
        double SampleDeviation = Math.pow(SampleVar, 0.5);
        System.out.println("Deviation = " + SampleDeviation);
```

```
        double coeffVar = SampleDeviation / SampleMean;
        System.out.println("Coefficient of variation = "
                + coeffVar);
    }

    /*Test with mean and deviation of frequency table*/
//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][],
            int nClasses) {
        double n = 0;
        for (int i = 0; i < nClasses; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

//Calculates the sum XF   of
//a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the sum X2F   of
//a table of absolute frequencies
    private static double
            sumX2FFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][0]
                    * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][],
            int nClasses) {
```

```
        double n = nFreqTable(FreqTable, nClasses);
        double mean = sumXFFreqTable(FreqTable, nClasses) / n;
        return mean;
    }

//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][],
            int nClasses) {

        double n = nFreqTable(FreqTable, nClasses);
        double Sxx = sumX2FFreqTable(FreqTable, nClasses)
                - Math.pow(sumXFFreqTable(FreqTable, nClasses), 2) / n;
        double var = Sxx / (n - 1);
        return var;
    }

// Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][],
            int nClasses) {
        double var = varFreqTable(FreqTable, nClasses);
        double dev = Math.pow(var, 0.5);
        return dev;
    }

// Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][],
            int nClasses) {
        double n = nFreqTable(FreqTable, nClasses);
        System.out.println("Number of measured objects = " + n);
        double sumXF = sumXFFreqTable(FreqTable, nClasses);
        System.out.println("sum xF  = " + sumXF);
        double mean = meanFreqTable(FreqTable, nClasses);
        System.out.println("Mean = " + mean);
        double sumX2F = sumX2FFreqTable(FreqTable, nClasses);
        System.out.println("sum x2F  = " + sumX2F);
        double var = varFreqTable(FreqTable, nClasses);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable, nClasses);
        System.out.println("Deviation =  " + deviation);
    }

//Test: grouping must produce a typical chi-square
//distribution
    private static void tableForExcel() {
```

```
        System.out.println("\nPaste next numbers to Excel, "
                + "OpenOffice or Gnumeric."
                + "\nMake a bar char."
                + "\nThe typical t shape  shall appear "
                + "\nelse there is a bug."
                + "\n");
        System.out.println("Absolute frequency "
                + "of discrepancies:");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FREQTABLE[i][1]);
        }
    }

//Tests for correctness
    private static void testZone() {
        System.out.println("\n\nTest zone");
        System.out.println("\n\nFirst random variable");
        //Mean and variance of events
        System.out.println("\nMean and Variance "
                + "of sample mean:");
        meanVarVector(XBARS1, NTRIALS);
        double h = muHat2 + MU1MINUSMU2;
        System.out.println("Expected mean = " + h);
        //Central limit theorem
        System.out.println("Expected deviation = "
                + sigmaHat1 / Math.sqrt(SAMPLESIZE1));
        //Mean and variance of events
        System.out.println("\n\nSecond random variable");
        System.out.println("\nMean and Variance "
                + "of sample mean:");
        meanVarVector(XBARS2, NTRIALS);
        System.out.println("Expected mean = " + muHat2);
        //Central limit theorem
        System.out.println("Expected deviation = "
                + sigmaHat2 / Math.sqrt(SAMPLESIZE2));
        //Degrees of freedom of theoretically expected
        //t distribution
        System.out.println("\nDegrees of freedom");
        double nx = SAMPLESIZE1;
        double ny = SAMPLESIZE2;
        double num = Math.pow(varHat1 / nx + varHat2 / ny, 2);
        double den1 = (varHat1 / nx) * (varHat1 / nx) / (nx + 1);
        double den2 = (varHat2 / ny) * (varHat2 / ny) / (ny + 1);
        double den = den1 + den2;
        nu = num / den - 2;
```

```
        System.out.println("num  = " + num);
        System.out.println("den1 = " + den1);
        System.out.println("den2 = " + den2);
        System.out.println("den = " + den);
        System.out.println("gl = " + nu);
        //Theoretical mean and deviation of the
        //t-distribution
        double meant = 0;
        double vart = nu / (nu - 2);

        System.out.println("\nThe table of absolute "
                + "frequencies of discrepancies"
                + " \nmust represent"
                + " a t-distribution (like a bell) \nwith mean " + meant
                + " and variance " + vart);
        //Mean and variance of frequency table
        System.out.println("Found values in simulation");
        meanVarTable(FREQTABLE, nClasses);
        //Prints the absolute frequencies of discrepancies
        //they can be pasted to Excel, OpenOffice or Gnumeric
        tableForExcel();
    }

    public static void meansAndVars() {
        //Mean of experimental sample1
        muHat1 = meanVector(DATA1, SAMPLESIZE1);
        //Variance of experimental sample
        varHat1 = varVector(DATA1, SAMPLESIZE1);
        //Deviation
        sigmaHat1 = Math.sqrt(varHat1);
        //Mean of experimental sample1
        muHat2 = meanVector(DATA2, SAMPLESIZE2);
        //Variance of experimental sample
        varHat2 = varVector(DATA2, SAMPLESIZE2);
        //Deviation
        sigmaHat2 = Math.sqrt(varHat2);
        System.out.println(" ");
        System.out.println("Mean of X = " + muHat1);
        System.out.println("Var of  of X = " + varHat1);
        System.out.println("Deviation  of  of X = " + sigmaHat1);
        System.out.println("Mean of Y = " + muHat2);
        System.out.println("Var of  of Y = " + varHat2);
        System.out.println("Deviation  of  of Y = " + sigmaHat2);
    }

    public static void main(String[] args) {
```

```
        meansAndVars();
        virtualWorld();
        System.out.println("Discrepancies");
        discrepancy(VAR1, VAR2);
        group(DISCREPANCY);
        System.out.println("Frequency table of discrepancies");
        printFreqTable(FREQTABLE, nClasses);
        //Minor test for correct grouping
        test(FREQTABLE);
        double criticalValueUp = criticalValueUp(DISCREPANCY);
        double criticalValueDown
                = criticalValueDown(DISCREPANCY);
        System.out.println("Experimental sample variance 1 = "
                + varHat1);
        System.out.println("Experimental sample variance 2 = "
                + varHat2);
        //Discrepancy due to the experiment
        dExp = discrepancy(muHat1, varHat1, muHat2, varHat2);
        testZone();
        System.out.println("Discrepancy due  to the "
                + "experimental event   = " + dExp);
        //Test for critical value
        System.out.println("The expected upper critical "
                + "value for \n"
                + nu + " degrees of freedom  is 2.26"
                + "\nThe lower bound is -2.26."
        );
        System.out.println("Found upper critical value  = "
                + criticalValueUp);
        System.out.println("Found lower critical value  = "
                + criticalValueDown);
        if ((dExp > criticalValueUp)
                || (dExp < criticalValueDown)) {
            System.out.println("The event is extreme "
                    + " according to Ho: invent another theory.");
        } else {
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }
    }
}//End of Program E140 TwoMeans
```

141. ***Exercise.*** *Run the program and play with the code.*

142. ***Research.*** *Develop a simulation for the comparison of two means given*

that an F-test renders equality of variances.

8.5 The KS test for normality

Some popular statistical tests explicitly demand to work with normally distributed random variables because they are very sensitive to departures from normality, i.e., they can render the wrong result. One example is in regression: if residuals are not normally distributed, a change of model is the next try to follow. Thus, if one claims that certain distribution has a normal distribution, one must prove that. To that aim, various procedures has been invented. The next example illustrates how one of them, the Kolgomorov-Smirnov test, operates.

143. ***Example*** *The following are the tallness in centimeters of 72 young men:*

145, 164, 171, 183, 177, 184, 168, 163, 157, 172,
179, 168, 172, 170, 172, 157, 162, 159, 164, 158,
173, 181, 162, 159, 167, 162, 166, 147, 163, 152,
156, 150, 170, 149, 180, 148, 163, 168, 169, 158,
150, 158, 174, 179, 168, 167, 182, 174, 168, 160,
180, 152, 160, 160, 148, 155, 173, 151, 149, 169,
161, 155, 160, 176, 177, 170, 169, 165, 146, 175,
156, 166

We group data beginning from 150 and in intervals of length 10. The associated frequency table corresponds to the first two columns of the table below.

Let us decide whether or not a distribution of tallness given by the table below fits a normal distribution. So, the H_o in a Kolmogorov-Smirnov test is that data come from a normally distributed random variable. The mean height is 165 and the standard deviation is 9.5743 (calculated from row data). So the $z(x)$ associated to a tallness x is $z(x) = \frac{x-1.65}{9.5743}$. Tallness is represented by x, the absolute frequency of x is $N(x)$, the cumulative absolute frequency is $F(x)$, the relative cumulative frequency of x is $\Phi(x)$, the distribution function of $z(x)$ is $\Psi(z(x))$ and the absolute value of the difference between the theoretical expected distribution and the experimental one is $|\Phi(x) - \Psi(z(x))|$. The null hypothesis in this test is that these differences are due to mere randomness. The test is calculated with the help of the next table:

Fitting of tallness in meters to a normal distribution						
x	$N(x)$	$F(x)$	$\Phi(x)$	z(x)	$\Psi(z(x))$	$\lvert\Phi(x) - \Phi(z(x))\rvert$
1.50	12	12	0.167	-1.57	0.059	0.108
1.60	24	36	0.5	-0.522	0.301	0.199
1.70	24	60	0.833	0.522	0.699	0.134
1.80	12	72	1	1.57	0.941	0.058

The observed maximum value of $|\Phi(x)-\Phi(z(x))|$ is $D = 0.199$ but, according to the Kolmogorov Smirnov theory and if the null hypothesis is correct, the maximum allowed value would be 0.192 (Marques, [67] 1990). Since the observed difference is greater than the allowed one, we reject the null hypothesis: our data do not fit a normal distribution. The value 0.192 was calculated as $1.63/\sqrt{n} = 1.63/\sqrt{72} = 1.63/8.48 = 0.192$ an approximation that is valid when the number of events is greater than 35. The constant 1.63 is the value associated to the significance $\alpha = 0.01$ and n is the total number of data. For $\alpha = 0.05$, that constant is 1.36 and the corresponding maximum allowed difference would be $1.36/\sqrt{72} = 1.36/8.48 = 0.16$. For this value of α, the null hypothesis also would be rejected (of course).

144. ***Research.*** *The theoretical distribution of discrepancies between observed and expected according to the null hypothesis is described in terms of Wiener processes that describe Brownian movement. Inquire about this. Hint: check Wikipedia with the following key expressions: Kolmogorov Smirnov test, Brownian motion, stochastic process, Brownian bridge process, Wiener-process.*

Let us implement our own version of the Kolmogorov-Smirnov test, which works not with density but with distribution functions. Let us pay attention to the fact that simulations are more easy to understand and to implement than general but abstract mathematics, which exist only for a handful of cases. Anyway, it is intelligent to keep in mind that we have tested the results of our previous simulations against mathematically expected values. The verified compatibility has inspired confidence in our procedures and in Java. This confidence is a great treasure because of which we feel eternally indebted with abstract Mathematics.

145. ***Simulation to decide a null hypothesis about normality.*** *Let H_o : A given data set that is grouped in a table of absolute frequencies T comes from a sample of a normally distributed random variable. We must decide this null hypothesis given a certain level of significance. Our procedure is as follows: We use table T to calculate the mean, variance and deviation of the sample. Next, we generate a random sequence of exactly the same size as that of the experimental sample and with a normal distribution with mean and deviation equal to the mean and deviation of table T. We group simulated data following the same protocol that was made to produce table T. Next, we construct a comparative table as that of the previous example where we have on one hand the cumulative function of data $\Phi(x)$ together with the cumulative distribution of the standard normal distribution $\Psi(z(x))$, which is the null hypothesis. We measure the discrepancy between observed $\Phi(x)$ and expected $\Psi(z(x))$ according to $|\Phi(x)-\Psi(z(x))|$ and take the maximal value as the measure of the discrepancy, D. We repeat this procedure many times to estimate the distribution of D and to define in agreement with the chosen level of significance what is normal and what is alien, extreme, outlier. Our test is one tailed since we work with absolute values and are interested in large discrepancies only. We calculate D_{exp}, the discrepancy due to the experimental value, and we judge it: if it is normal, we accept the null hypothesis and if it is outlier, we reject it.*

146. ***The code for a simulation of the Kolmogorov-Smirnov test follows.***

```
/*Program E146 KolSmirTest
 We simulate a Kolmogorov-Smirnov test
 in which a null hypothesis for normality is checked out.
 Procedure:
 We group raw experimental data into a frequency table T.
 We  use  table T to calculate
 the mean,  variance and deviation of the sample.
 We calculate the discrepancy due to the experimental
 value DExp as follows:
 we construct   the
 cumulative function of data Phi(x) together with
 the cumulative distribution of the standard normal
 distribution Psi(z(x)), which is the null hypothesis.
 We measure the discrepancy between observed Phi(x) and
 expected Psi(z(x)) according to
 | Phi(x)-Psi(z(x)) |
 and take the maximal value as the measure of the
 discrepancy DExp.
 Next, we generate a random sequence of exactly
 the same size as that of the experimental sample
 and with a normal distribution with mean and
 deviation equal  to the mean and deviation of
 table T. We group simulated data following the
 same protocol that was made to produce table  T.
 Next, we measure the discrepancy between observed
 in the virtual world and expected under Ho by
 exactly the same procedure used to calculate DExp.
  We repeat  this procedure many times
 to estimate the distribution of  D  and to define in
 agreement with the chosen level of significance
 what is normal and what is   outlier.
 Our test is one tailed since we take absolute value
 and are interested in large discrepancies only.
 At last, we judge the discrepancy due to the experiment:
 if it normal, we accept the null hypothesis
 but  if it is outlier, we reject it.
 */


package ejvol5p;

import java.util.Random;
```

```
public class KolSmirTest {
    //Experimental data

    private static final double EXPDATA[] = {
        145, 164, 171, 183, 177, 184, 168, 163, 157, 172,
        179, 168, 172, 170, 172, 157, 162, 159, 164, 158,
        173, 181, 162, 159, 167, 162, 166, 147, 163, 152,
        156, 150, 170, 149, 180, 148, 163, 168, 169, 158,
        150, 158, 174, 179, 168, 167, 182, 174, 168, 160,
        180, 152, 160, 160, 148, 155, 173, 151, 149, 169,
        161, 155, 160, 176, 177, 170, 169, 165, 146, 175,
        156, 166
    };

//The left border of intervals, less or equal to the
//minimum value of data.
    private static double infLimit = 145;
//Interval length
    private static double intervalLength = 10;
//Max number of allowed classes
    private static final int MAXNCLASSES = 1000;
//Borders of tallness intervals
    private static final double BARRIERSTALLNESS[]
            = {145, 155, 165, 175, 185};
    private static final double CLASSMARKERSTALLNESS[]
            = {150, 160, 170, 180};
    private static int nClasses;
    private static final double FREQTABLE[][]
            = new double[MAXNCLASSES][2];
//Observed cumulative distribution
    private static final double PHI[]
            = new double[MAXNCLASSES];
//Expected cumulative distribution
    private static final double PSI[]
            = {0.059, 0.301, 0.699, 0.941};
//Parameters according to Ho
    private static double muHat;
    private static double sigmaHat;
//Size of  sample
    private static final int SAMPLESIZE = 72;
//number of trials in the virtual world
    private static final int NTRIALS = 87650;
//Significance level
    private static final double ALPHA = 0.05;
//Turn on of the random generator
```

```
    static Random r = new Random();
//A list with a random sample
    private static final double VECTSAMPLE[]
            = new double[1000];
//Dimension of arrays
    private static final int N = 100000;
//Means   of simulated samples,   generated at random
    private static final double SIMXBAR[] = new double[N];
//Variances of samples  generated at random
//    private static final double SIMVAR[] = new double[N];
//Discrepancy due to experiment
    private static double dExp;
//Discrepancy between fact and idea
    private static final double DISCREPANCY[] = new double[N];
//Borders of the intervals
    private static final double BARRIERS[]
            = new double[MAXNCLASSES + 1];
    private static final double CLASSMARKERS[]
            = new double[MAXNCLASSES];
    private static double min;
    private static double max;
    private static boolean print;

//Sums data in vector with real numbers
    private static double sumVector(double Vect[],
            int n) {
        double sum = 0;
        for (int i = 0; i < n; i++) {
            sum = sum + Vect[i];
        }
        return sum;
    }

//The mean of a list of data
    private static double meanVector(double Vect[], int n) {
        double sum = sumVector(Vect, n);
        double mean = sum / n;
        return mean;
    }

//The variance of a list of data
    private static double varVector(double Vect[],
            int n) {
        //Mean
        double mean = meanVector(Vect, n);
        // Variance is calculated by definition
```

```
        double sum2 = 0;
        for (int i = 0; i < n; i++) {
            sum2 = sum2 + (Vect[i] - mean) * (Vect[i] - mean);
        }
        double var = sum2 / (n - 1);
        return var;
    }

//The mean and variance of a list of data
    private static void meanVarVector(double DataVect[]) {
        System.out.println("Data length = " + DataVect.length);
        double mean = meanVector(DataVect, DataVect.length);
        System.out.println("Mean = " + mean);
        double var = varVector(DataVect, DataVect.length);
        System.out.println("Variance \t " + var);
        double deviation = Math.pow(var, 0.5);
        System.out.println("Deviation = " + deviation);
        double coeffVar = deviation / mean;
        System.out.println("Coefficient of variation = "
                + coeffVar);
    }

//Prints data in vector
    private static void printVector(double Vect[], int n) {
        for (int i = 0; i < n; i++) {
            System.out.println(i + "   " + Vect[i]);
        }
    }

//Prints the frequency table
    private static void printFreqTable(double FreqTable[][],
            int nClasses) {
        System.out.println("x and its absolute frequency");
        for (int i = 0; i <= nClasses; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][]) {
        double n = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            n = n + FreqTable[i][1];
```

```
        }
        return n;
    }

//Calculates the sum XF   of
//a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][]) {
        double sum = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the sum X2F   of
//a table of absolute frequencies
    private static double
            sumX2FFreqTable(double FreqTable[][]) {
        double sum = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][0]
                    * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][]) {
        double n = nFreqTable(FreqTable);
        double mean = sumXFFreqTable(FreqTable) / n;
        return mean;
    }

//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][]) {

        double n = nFreqTable(FreqTable);
        double Sxx = sumX2FFreqTable(FreqTable)
                - Math.pow(sumXFFreqTable(FreqTable), 2) / n;
        double var = Sxx / (n - 1);
        return var;
```

```
    }

// Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][]) {
        double var = varFreqTable(FreqTable);
        double dev = Math.pow(var, 0.5);
        return dev;
    }

//Calculates the coefficient of variation of
//a table of absolute frequencies
    private static double coefficient(double FreqTable[][]) {
        double c = devFreqTable(FreqTable)
                / meanFreqTable(FreqTable);
        return c;
    }

// Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][]) {
        printFreqTable(FreqTable, nClasses);
        double n = nFreqTable(FreqTable);
        System.out.println("Number of measured objects = " + n);
        double sumXF = sumXFFreqTable(FreqTable);
        System.out.println("sum xF  = " + sumXF);
        double mean = meanFreqTable(FreqTable);
        //Mean is captured
        muHat = mean;
        System.out.println("Mean = " + mean);
        double sumX2F = sumX2FFreqTable(FreqTable);
        System.out.println("sum x2F  = " + sumX2F);
        double var = varFreqTable(FreqTable);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable);
        System.out.println("Deviation =  " + deviation);
        //Deviation is captured
        sigmaHat = deviation;
        double coefficient = coefficient(FreqTable);
        System.out.println("Coefficient of variation =  "
                + coefficient);
    }

//***********Grouping*****************************
//Data are grouped into interval classes.
    private static void group(double Vect[],
            double classMarkers[],
```

```
            double BARRIERS[],
            int nClasses, int nEvents) {

//Class markers are defined
        for (int j = 0; j < nClasses; j++) {
            FREQTABLE[j][0] = classMarkers[j];
            FREQTABLE[j][1] = 0;
        }
//Frequencies are calculated.
        for (int i = 0; i < nEvents; i++) {
            for (int j = 0; j < nClasses; j++) {
                if ((BARRIERS[j] <= Vect[i])
                        & (Vect[i] < BARRIERS[j + 1])) {
                    FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
                }
            }
        }
    }

//***********Measuring of discrepancy***********
//Cumulative distribution of observed frequency table
    public static void Phi(double FreqTable[][],
            int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][1] / SAMPLESIZE;
            PHI[i] = sum;
        }
    }

//Measures the discrepancy between observed Phi(x)
//and expected PSI(z(x))
    public static double discrepancy(double Phi[],
            int nClasses, String s) {
        double D = 0;
        double d;
        if (print) {
            System.out.println("\nLine per line discrepancies");
        }
        for (int i = 0; i < nClasses; i++) {
            d = Math.abs(Phi[i] - PSI[i]);
            if (print) {
                System.out.println(i + "   " + d);
            }
            if (d > D) {
                D = d;
```

```
            }
        }
        if (print) {
            System.out.println("Discrepancy due to "
                    + s + " =  " + D);
        }
        return D;
    }

//*************Simulation zone***************
//Generates   SAMPLESIZE  random numbers
//with the requested normal distribution.
    public static void randomSample(double mean,
            double deviation,
            int sampleSize) {
        //We generate a random sequence
        //of size equal to that of the sample
        double event;
        for (int i = 0; i < sampleSize; i++) {
            //Random numbers with Gaussian distribution
            //mean zero and deviation 1.
            double d = r.nextGaussian();
            //System.out.println(d ) ;
            // Rescale numbers: mean = mean  and
            // deviation = deviation
            event = deviation * d + mean;
            //System.out.println(event ) ;
            VECTSAMPLE[i] = event;
        }
    }

//Extreme events are associated to available extreme
//classes and output data are grouped
    public static void GroupAndCutTails(double vectSample[],
            int sampleSize) {
        //Extreme events are smeared out
        double Min = infLimit;
        double Max = infLimit + nClasses * intervalLength;
        for (int i = 0; i < sampleSize; i++) {
            if (vectSample[i] < Min) {
                vectSample[i] = Min;
            }
            if (vectSample[i] > Max) {
                vectSample[i] = Max;
            }
        }
```

```
        //Data are grouped
        group(vectSample, CLASSMARKERSTALLNESS, BARRIERSTALLNESS,
                4, sampleSize);
    }

    /*We repeat NTRIALS times the following procedure:
sampleSize random  numbers are generated.
Next, those numbers are grouped
in a frequency table T with the same form as
experimental data. This implies taking care of outliers.
Cumulative distribution for T is calculated
and the discrepancy that is expected by the null hypothesis
is measured. Result is kept in DISCREPANCY[].
     */
    public static void virtualWorld() {
        //We run the virtual world NTRIALS times
        for (int i = 0; i < NTRIALS; i++) {
            //sampleSize1 random numbers with normal distribution
            randomSample(muHat, sigmaHat, SAMPLESIZE);
            //mean of sample to be used in a test
            //double xBar = meanVector(VECTSAMPLE, SAMPLESIZE);
            //variance
            //double xVariance = varVector(VECTSAMPLE, SAMPLESIZE);
            //SIMXBAR[i] = xBar;
            //SIMVAR[i] = xVariance;
            GroupAndCutTails(VECTSAMPLE, SAMPLESIZE);
            Phi(FREQTABLE, nClasses);
            double D = discrepancy(PHI, nClasses, " simulation");
            DISCREPANCY[i] = D;
        }
    }

//The range of discrepancies is evaluated
//and is used to calculate the length of each
//interval to group Discrepancies.
    public static double intervalLength() {
        min = 100000;
        max = 0;
        for (int i = 0; i < NTRIALS; i++) {
            if (DISCREPANCY[i] < min) {
                min = DISCREPANCY[i];
            }
            if (DISCREPANCY[i] > max) {
                max = DISCREPANCY[i];
            }
        }
```

```
        System.out.println("min = " + min);
        System.out.println("max = " + max);
        double range = max - min;
        double length = range / nClasses;
        return length;
    }

//Calculates the sum    of absolute frequencies
//of a frequency table
    private static void
            test(double FreqTable[][]) {
        double sum = 0;
        for (int i = 0; i <= nClasses; i++) {
            sum = sum + FreqTable[i][1];
        }
        System.out.println("Number of events = " + sum);
    }

//The border of each interval is calculated
    private static void getBorders(double Vect[]) {
        BARRIERS[0] = infLimit;
        CLASSMARKERS[0] = infLimit + intervalLength / 2;
//Calculate maximal value of data
        double mmyMax = Vect[0];
        for (int j = 1; j < Vect.length; j++) {
            if (Vect[j] > mmyMax) {
                mmyMax = Vect[j];
            }
        }
        System.out.println("Max value = " + mmyMax);
//Calculates number of classes
        nClasses = (int) ((mmyMax - infLimit) / intervalLength) + 1;
        System.out.println("Number of classes = " + nClasses);
//Calculates interval borders and class markers
        for (int j = 1; j <= nClasses; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + intervalLength;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + intervalLength;
        }
        /*System.out.println( "Borders are" );
printVector(BARRIERS);
System.out.println( "Class markers are" );
printVector(CLASSMARKERS);*/
    }

//Data are grouped into interval classes.
    private static void groupMake(double Vect[],
```

```
            int nClasses, int NTRIALS) {
        intervalLength = intervalLength();
        infLimit = min - intervalLength / 2;
        getBorders(Vect);
        group(Vect, CLASSMARKERS, BARRIERS, nClasses + 1, NTRIALS);
    }

//The upper critical value with two  tails is calculated
//with significance level = ALPHA = 0.05
    private static double criticalValueUp(double Vect[]) {
        double c = 0;
        //Number of events that classify as extreme
        //in either tail
        int e = (int) (NTRIALS * ALPHA);
        System.out.println("Number of extreme events in "
                + "upper tail = " + e);
        //Clon of Vect[]
        double Clon[] = new double[NTRIALS];
        /*
        for (int i = 0; i < NTRIALS; i++) {
        Clon[i] = Vect[i];
        }
         */
        System.arraycopy(Vect, 0, Clon, 0, NTRIALS);
        //Discrepancies are sorted
        //in decreasing order until completion of
        //e events.
        System.out.println("\nDiscrepancies are sorted in "
                + "\ndecreasing order until completion of demanded"
                + "\nnumber of extreme  events.");
        int Champ;
        for (int i = 0; i < e; i++) {
            Champ = 0;
            for (int j = 1; j < NTRIALS; j++) {
                if (Clon[j] >= Clon[Champ]) {
                    Champ = j;
                }
            }
            //Instruction that prints outliers
            /* System.out.println(  i + "th ind. is  No "
               + Champ + " Discrep =  " + Clon[Champ]);*/
            c = Clon[Champ];
            Clon[Champ] = 0;
        }
        return c;
    }
```

```
    /*Test with mean and deviation of events*/
//The mean and variance of a list of data
    private static void meanVarVector(double DataVect[],
            int lim) {
        System.out.println("Sampled data");
        System.out.println("Data length = " + lim);
        double SampleMean = meanVector(DataVect, lim);
        System.out.println("Mean = " + SampleMean);
        double SampleVar = varVector(DataVect, lim);
        System.out.println("Variance = " + SampleVar);
        double SampleDeviation = Math.pow(SampleVar, 0.5);
        System.out.println("Deviation = " + SampleDeviation);
        double coeffVar = SampleDeviation / SampleMean;
        System.out.println("Coefficient of variation = "
                + coeffVar);
    }

    /*Test with mean and deviation of frequency table*/
//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][],
            int nClasses) {
        double n = 0;
        for (int i = 0; i < nClasses; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

//Calculates the sum XF   of
//a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the sum X2F   of
//a table of absolute frequencies
    private static double
```

```
            sumX2FFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][0]
                    * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][],
            int nClasses) {
        double n = nFreqTable(FreqTable, nClasses);
        double mean = sumXFFreqTable(FreqTable, nClasses) / n;
        return mean;
    }

//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][],
            int nClasses) {

        double n = nFreqTable(FreqTable, nClasses);
        double Sxx = sumX2FFreqTable(FreqTable, nClasses)
                - Math.pow(sumXFFreqTable(FreqTable, nClasses), 2) / n;
        double var = Sxx / (n - 1);
        return var;
    }

// Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][],
            int nClasses) {
        double var = varFreqTable(FreqTable, nClasses);
        double dev = Math.pow(var, 0.5);
        return dev;
    }

// Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][],
            int nClasses) {
        double n = nFreqTable(FreqTable, nClasses);
        System.out.println("Number of measured objects = " + n);
        double sumXF = sumXFFreqTable(FreqTable, nClasses);
```

```
        System.out.println("sum xF  = " + sumXF);
        double mean = meanFreqTable(FreqTable, nClasses);
        System.out.println("Mean = " + mean);
        double sumX2F = sumX2FFreqTable(FreqTable, nClasses);
        System.out.println("sum x2F  = " + sumX2F);
        double var = varFreqTable(FreqTable, nClasses);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable, nClasses);
        System.out.println("Deviation =  " + deviation);
    }

//Test: grouping must produce a central tendency
//distribution
    private static void tableForExcel() {
        System.out.println("\nPaste next numbers to Excel, "
                + "OpenOffice or Gnumeric."
                + "\nMake a bar char."
                + "\nA central tendency is  expected "
                + "\nelse there is a bug."
                + "\n");
        System.out.println("Absolute frequency "
                + "of discrepancies:");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FREQTABLE[i][1]);
        }
    }

//Tests for correctness
    private static void testZone() {
        System.out.println("\n\nTest zone");
        //Mean and variance of events
        System.out.println("\nMean and Variance "
                + "of sample mean:");
        meanVarVector(SIMXBAR, NTRIALS);

        System.out.println("Expected mean = " + muHat);
        //Central limit theorem
        System.out.println("Expected deviation = "
                + sigmaHat / Math.sqrt(SAMPLESIZE));
        //Prints the absolute frequencies of discrepancies
        //they can be pasted to Excel, OpenOffice or Gnumeric
        tableForExcel();
    }

    public static void main(String[] args) {
        System.out.println("**** EXPERIMENT ****");
```

```
System.out.println("\nUngrouped  data: \n");
meanVarVector(EXPDATA);
System.out.println("\nGrouped experimental data: \n");
nClasses = 4;
intervalLength = 10;
min = infLimit;
group(EXPDATA, CLASSMARKERSTALLNESS,
        BARRIERSTALLNESS, 4, SAMPLESIZE);
meanVarTable(FREQTABLE);
//Cumulative distribution for experiment
Phi(FREQTABLE, nClasses);
System.out.println(
        "\nObserved cumulative Distribution");
printVector(PHI, nClasses);
System.out.println("\n**** Ho: data come from a normal distribution ****");
System.out.println("\n**** EXPECTED UNDER Ho ****");
System.out.println(
        "\nExpected Cumulative Distribution.");
printVector(PSI, nClasses);
//Discrepancy due to the experiment
dExp = discrepancy(PHI, nClasses, "Experiment");
System.out.println("\n**** OBSERVED DISCREPANCY ****");
System.out.println("\nDiscrepancy due  to the "
        + "experimental event   = " + dExp);

//**********SIMULATION*******
System.out.println("\n**** SIMULATION ****");
print = false;
//Discrepancies in the virtual world
virtualWorld();
//MODIFY THIS PARAMETER
nClasses = 20;
System.out.println("NTRIALS = " + NTRIALS);
groupMake(DISCREPANCY, nClasses, NTRIALS);
System.out.println("Frequency table of discrepancies");
printFreqTable(FREQTABLE, nClasses);
meanVarTable(FREQTABLE, nClasses);
//Minor test for correct grouping
test(FREQTABLE);
double criticalValueUp = criticalValueUp(DISCREPANCY);
testZone();
System.out.println("Discrepancy due  to the "
        + "experimental event   = " + dExp);
//Test for critical value
System.out.println("The expected upper critical "
        + "\nvalue  (with infinitely many classes) = 0.16"
```

```
        );
        System.out.println("Found upper critical value with "
                + nClasses + " classes  = " + criticalValueUp);

        if ((dExp > criticalValueUp)) {
            System.out.println("The event is extreme "
                    + " according to Ho: invent another theory.");
        } else {
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }

    }

}//End of Program E146 KolSmirTest
```

147. ***Exercise.*** *Run the program and play with the code.*

148. ***Exercise.*** *The previous program was developed by reusing of previous material. So, we pasted one piece from here and other form there taking care of good performance. In spite of the fact that the program has been carefully refined, it still conserves a track of its origin: it has two distinct versions of procedures to calculate mean and variances of frequency tables. To see this more easily, activate the menu* `Window` $\rightarrow$ `Navigator`*: you will see that there are many repeats of procedures to calculate means and variances. Simplify the program to remove such tracks and to gain professional cleanness and simplicity. make sure that the new program fulfills exactly the same function as the original redundant one. Reuse is the key to evolution: based on your personal experience with this exercise, would you dare to say that reuse must or must not left clear tracks in the fossil record?* *Answer*

149. ***Challenge.*** *Run the program using diverse values of the number of classes of the frequency table of discrepancies in the virtual world. Recommended values: 20, 100, 500. That value can be changed in line 694 (or so). Observe that what appears under low resolution to be a bell is really a superimposing of 3 or more bells with possible different means and deviations. Actually, these distributions are not normal because they have a sting at zero. Design and run a research program to test the next theory to explain this superposition phenomenon:*

We work with frequency tables with four classes. That setting is enough to estimate the mean, the variance and the degree of asymmetry of generated distributions. Now, a sample generated at random can differ from the true distribution by the mean, by the variance, by the degree of asymmetry or by mixture of these deviations. So, there are many ways to approach the expected distribution. Each way generates its own trend of approximation and its own distribution of results with its own mean and variance, i.e., with its own stinged bell. We have

a prediction: with more classes and higher resolution, more superimposed bells shall appear although it could be difficult to disentangle them.

150. ***Research.*** *The Kolmogorov-Smirnov test for normality is possibly the simplest but certainly it is not the better. Inquire about other tests and implement the corresponding simulation. Hint: check Wikipedia with the next key expressions: normality test, Shapiro-Wilk test, Anderson-Darling test, normality tests in the R package.*

8.6 Independence of attributes

The purpose of a test of independence of attributes is to decide whether or not two attributes are statistically correlated. The attribute must not be necessarily quantitative and so one works with frequency arrangements in the form of a **contingency table**. Children understand quite well what this test is about when they are around eight: they perceive that blue eyes and light skin might come together with much more frequency than blue eyes and obscure skin. This generalization is correct in most places of the world but it might be false in some coastal touristic zones.

151. ***Research.*** *Consult statistical bibliography about the theme, formulate the appropriate theorem and implement the corresponding simulation. Compare your results with those of established receipts. If you find a strong discrepancy, keep in mind that theory is for this problem only an approximation.*

8.7 Simulation of anyone distribution

152. ***Motivation and purpose.*** *We have considered natural questions that have meaning for the generality of distributions, be they normal o not. Some procedures can be applied to many diverse types of distributions, such as those that hang from the central limit theorem about means. In any case, how can we make sure that a procedure can be applied to distributions other than normal ones? The only answer is to run the same procedures over non normally distributed random variables to see what happens. And, what shall we do if anomalous distributions render anomalous behavior? To dwell into these questions, we must be able to simulate a random variable with whatever distribution. Our purpose in this section is precisely to show how this is done. The rest is just more of the same and will be not pursued here.*

Let us learn how one can program a source of random numbers to fit **whatever distribution** one desires.

153. ***Fitting a discrete distribution.*** *To fix ideas, let us suppose that one has the next table, which registers the relative frequencies of numbers 0,1,2 and 3. Our purpose is to program a random generator that follows exactly the distribution given by this frequency table.*

A table of relative frequencies	
numbers	Frequency
0	3/10
1	2/10
2	0/10
3	5/10

The next is the graphic of this table:

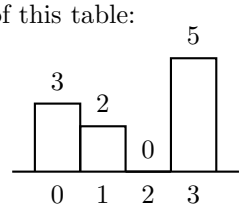


Figure 8.6. Our task is to design a source of random numbers with this distribution.

The idea to solve the problem is the following: we generate numbers from 0 to 9 (included). Next, we divide the range zero to nine in four parts in proportion to the relative frequencies given by the frequency table. Next, we rename the output of the source in agreement with the events given by the frequency table. Say, our first sub-range covers 0,1,2. If the random generator happens to output any one of this numbers, we define our output as 0. The next sub-range covers 3 and 4. If anyone of this numbers is produced, we report 1.The next sub-range covers 5, 6, 7, 8 and 9. For any one of those cases, one answers 3. As a result, 2 is never output. This is done in the next code:

```
//Program E153 WhateverD
//Fixes a bug in Program A213
//The program generates random integers
//with a given distribution.

package ejvol5p;

import java.util.Random;

public class WhateverD {
    // Absolute frequencies are defined:
    //The frequency of 0 is 3, that of 1 is 2,
    //that of 2 is zero and that of four is 5.

    static int Dist[] = {3, 2, 0, 5};

    // Counters are in count
    static int count[] = new int[4];
```

```
    // Turn on of the random generator
    static Random r = new Random();

    public static void main(String[] args) {
        for (int i = 0; i < Dist.length; i++) {
            count[i] = 0;
        }
        System.out.println("Generated numbers: ");
        for (int i = 0; i < 100; i++) {
            //A random integer less than 10 is generated
            //with a uniform distribution
            int n = r.nextInt(10);
            //Outputs are renamed
            if (n < 3) {
                count[0] = count[0] + 1;
                n = 0;
            }
            if ((3 <= n) & (n < 5)) {
                count[1] = count[1] + 1;
                n = 1;
            }
            if ((5 <= n) & (n < 10)) {
                count[3] = count[3] + 1;
                n = 3;
            }
            System.out.println(n);
        }
        System.out.println("\nEvents and their frequencies: ");
        for (int i = 0; i < Dist.length; i++) {
            System.out.println(i + "\t" + count[i]);
        }
        System.out.println("\nFrequencies: ");
        for (int i = 0; i < Dist.length; i++) {
            System.out.println(count[i]);
        }
    }//End of main
}//End of Program E153 WhateverD
```

154. ***Exercise.*** *Run the program, play with the code, say, modify the number of simulated events, and intuitively assess the accuracy of the fitting. Hint: help yourself with the graphics of Excel, LibreOffice or Gnumeric.*

155. ***Exercise.*** *Complement the previous code with a chi-square test over its output to rigorously and automatically assess the accuracy of the fitting. The chi-square test works as follows. Let us suppose that we have a distribution*

given by a frequency table Exp, which is the expected distribution under the null hypothesis. We also have a distribution of observed data given by Obs. We measure the discrepancy between the observed distribution and the reference one by

$$d_{exp} = \sum \frac{(Obs[i]-Exp[i])^2}{Exp[i]}$$

We must answer the next question: Can the registered discrepancy be explained by random effects or is it better to propose the existence of an alien systematic effect? The answer to this question depends on the chosen level of confidence and on the degrees of freedom which equals the number of classes of the distribution minus one. The test is decided with one tail for we are interested in large discrepancies. For a confidence level of 0.05, 3 degrees of freedom and one tail, the critical chi-square is 7.8. Answer

156. ***Challenge.*** *To make the previous exercise, we have applied blindly a well known receipt. Let us notice that the test assumes that for any distribution, be it normal or otherwise, the distribution of discrepancies follows a chi-square distribution. In reality, that is approximately correct and in some cases more than in others. So, we are in plain right of claiming that a simulation is more reliable. In consequence, design and develop a Java simulation whose aim is to study the distribution of discrepancies calculated exclusively for the expected distribution such as it is without further assumptions or approximations. After that, decide what is normal and what is outlier, a classification that will allow you to judge the discrepancy due to the experiment and to accept else reject the null hypothesis.*

157. ***Fitting a continuous distribution***

Let us suppose that we have a table of relative frequencies that describes a certain continuous distribution, such as in the next table:

A table of relative frequencies	
Interval	Frequency
0-0.999	3/10
1-1.999	2/10
2-2.999	0/10
3-3.999	5/10

Our task is to device a source of decimal random numbers that takes values in within 0 and 4 to fit the distribution given by the proposed histogram of relative frequencies but with the condition that the decimal numbers must have a uniform distribution in within each one of the four intervals. With a slight modification, we can use the same idea of the previous program to solve this problem:

```
//Program E157 WhateverC
//Fixes a bug in Program A215
//A source generates random numbers of type double
//with a given continuous distribution
//that is piecewise constant, i.e.,
//it has  a uniform distribution in within each one
//of given intervals.

package ejvol5p;

import java.util.Random;

public class WhateverC {

    //Absolute frequencies are defined:
    //The frequency of 0 is 3, that of 1 is 2,
    //that of 3 is zero and that of four is 5.
    private static final int DIST[] = {3, 2, 0, 5};
    private static final int NCLASSES = DIST.length;
    private static double sizeSample;
    private static final int SIZESIMULATION = 100000;
    private static final double EVENTS[]
            = new double[SIZESIMULATION];

    //Counters of simulated events are in count
    private static final double COUNT[] = new double[NCLASSES];
    //Predicted distribution
    private static final double PREDICTED[] = new double[NCLASSES];
    //Discrepancies line per line
    private static final double CRITICALCHI2 = 7.8;
    private static final double DISCREPANCY[] = new double[NCLASSES];
    //Turn on of the random generator
    private static final Random R = new Random();
    private static boolean printAll = true;

    //Prints data in vector
    private static void printVector(double Vect[]) {
        for (int i = 0; i < Vect.length; i++) {
            System.out.println(i + "    " + Vect[i]);
        }
    }


    private static void printTwoVectors(double Vect1[], double Vect2[]) {
        for (int i = 0; i < Vect1.length; i++) {
            System.out.println(Vect1[i] + "   " + Vect2[i]);
```

```
    }
}


//Returns the size of the sample
private static double sizeSample() {
    double s = 0;
    for (int i = 0; i < NCLASSES; i++) {
        if (DIST[i] > 0) {
            s = s + DIST[i];
        }
    }
    return s;
}

//PREDICTEDs what must be observed
private static void prediction() {
    for (int i = 0; i < NCLASSES; i++) {
        double probSubI = DIST[i] / sizeSample;
        PREDICTED[i] = probSubI * SIZESIMULATION;
    }
}

//Runs a chi^2-test to check the accuracy
//of our source of random numbers
//that must fit the distribution given by DIST
private static double chiTest() {
    double d = 0;
    for (int i = 0; i < NCLASSES; i++) {
        if (PREDICTED[i] > 0) {
            DISCREPANCY[i]
                    = Math.pow((COUNT[i] - PREDICTED[i]), 2)
                    / PREDICTED[i];
        }
        d = d + DISCREPANCY[i];
    }
    return d;
}

public static void main(String[] args) {
    for (int i = 0; i < DIST.length; i++) {
        COUNT[i] = 0;
    }
    for (int i = 0; i < SIZESIMULATION; i++) {
        //A random decimal number in (0,10) is generated
        //r.nextDouble() generates a random decimal number
```

```
            //with a uniform distribution in within 0 and 1.
            double e = 10 * R.nextDouble();
            if (printAll)
                System.out.println(e);
            //Outputs are renamed
            if (e < 3) {
                e = e / 3;
                COUNT[0] = COUNT[0] + 1;
            }
            if ((3 <= e) & (e < 5)) {
                e = 1 + (e - 3) / 2;
                COUNT[1] = COUNT[1] + 1;
            }
            if ((5 <= e) & (e < 10)) {
                e = 3 + (e - 5) / 5;
                COUNT[3] = COUNT[3] + 1;
            }
            EVENTS[i] = e;
        }
        printAll = false;
        if (printAll) {
            System.out.println(" Simulated events");
            printVector(EVENTS);
        }

        sizeSample = sizeSample();
        prediction();
        System.out.println("Observed and predicted values");
        printTwoVectors(COUNT, PREDICTED);
         double dExp = chiTest();
        System.out.println("Ho: observed values fit "
                + "predicted ones");
        System.out.println("Discrepancies");
        printVector(DISCREPANCY);
        System.out.println("Experimental discrepancy = " + dExp);
        System.out.println("Critical chi2 = " + CRITICALCHI2);
        if (dExp > CRITICALCHI2) {
            System.out.println("The event is extreme "
                    + " according to Ho: invent another theory.");
        } else {
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }

    }//End of main
}//End of Program E157 WhateverC
```

158. ***Exercise.*** *Run the program and play with the code.*

159. ***Exercise.*** *Test somehow the claim that the program produces random numbers with a uniform distribution in within, say, 0 and 1, 1 and 2 and so on.* *Answer*

160. ***Exercise.*** *A graphical study of the accurateness of our simulation reveals a border defect, whose corresponding frequencies seem to be depressed. Is this effect caused by our programs or is it a bug in the Java generator of random numbers? How can it be fixed?* *Answer*

161. ***Fitting a mathematical formula***

Our next step is to achieve a total degree of generality to simulate whatever distribution, say, one that is given in the form of a mathematical formula for a frequency distribution or for a density function or for a cumulative distribution. After some theoretical digression and to fix ideas, we will simulate a Pareto distribution.

To begin with, let us explain how must we proceed to design an evolvable generator of random numbers X for a step-wise function, such as that of the two previous programs, whose probability density function is the idealization of a bar chart of relative frequencies. The idea is the following: we have a Java inbuilt generator of random numbers with a uniform distribution in within zero and one. To reuse that generator to design a new random generator X with a probability density function such as a step function, we need to solve the next question:

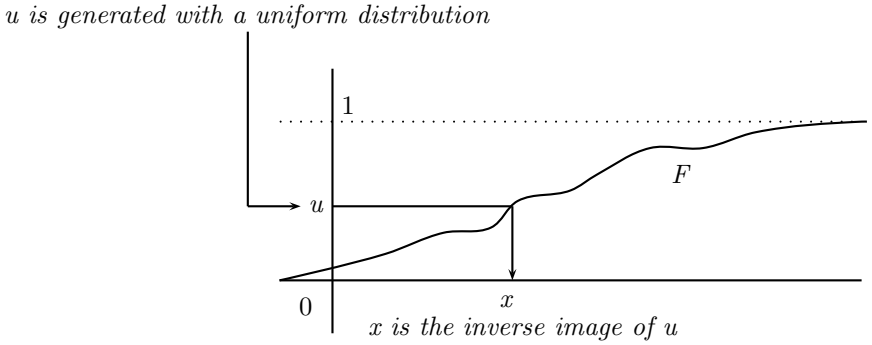


Figure 8.7.If we want x to have a predefined distribution, we need to adjust the exact and appropriate shape of F.

Which must be the shape of function F, as in the graphic, in order to have x with the predefined distribution X?

Let us solve this question for the simplest case of a probability density function represented by a step function as the following:

A table of relative frequencies	
Intervals	Frequency
0-0.999	3/10
1-1.999	2/10
2-2.999	0/10
3-3.999	5/10

whose bar chart is the following:

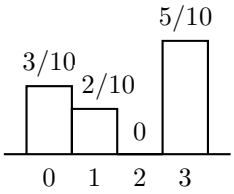


Figure 8.8. Our task is to design a source of random numbers X with this continuous distribution.

The problem can be easily solved if we guide ourselves by the table of cumulative frequencies

A table of relative an cumulative frequencies		
Intervals	Frequency	Cumulative function
0-0.999	3/10	3/10
1-1.999	2/10	5/10
2-2.999	0/10	5/10
3-3.999	5/10	10/10

Now, we get the following graphic:

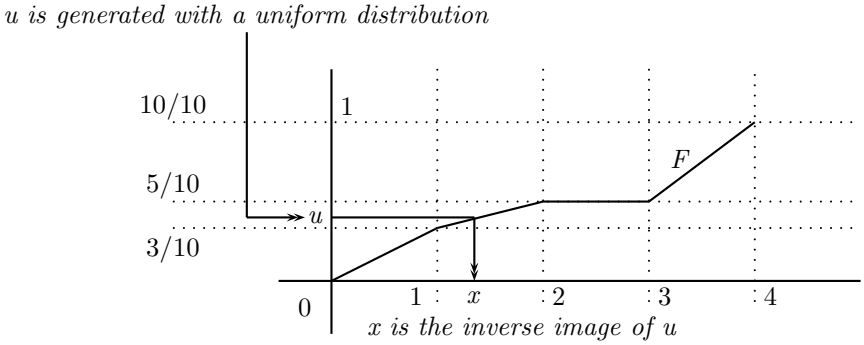


Figure 8.9. If u has a uniform distribution in within zero and one (over the vertical axis), then x has the requested distribution if we define $x = F^{-1}(u)$, the F-inverse image of u.

The reason of our procedure is that F transmits the probabilities from the vertical axis towards the horizontal axis in consonance with intervals and their relative frequencies. In fact, to the interval (0,1), this procedure assigns the probability 3/10. To the interval (1,2) corresponds the probability 5/10 - 3/10 = 2/10. To the interval (2,3) corresponds nothing. To the interval (3,4) corresponds half of the vertical axis, i.e., 5/10.

Our immediate guess is that $F(x)$ shall always be the cumulative distribution of $f(x)$, the probability density function of X and that idealizes the relative frequencies. For the case of a general probability density function $f(s)$, the cumulative function is $F(x) = \int_{-\infty}^{x} f(s)ds$ as in the next graphic:

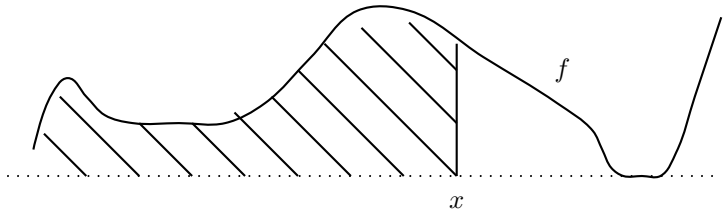


Figure 8.10. The cumulative function $F(x)$ of a probability density function $f(x)$ is the area accumulated from $-\infty$ to x, i.e., $F(x) = \int_{-\infty}^{x} f(s)ds$.

The relation among the random variable X, its density function f that is supposed to be continuous, and its cumulative function F is as follows:

- $p(a < x < b) = \int_a^b f(s)ds$
- $F(x) = \int_{-\infty}^{x} f(s)ds$
- $F'(x) = f(x)$
- $p(a < x < b) = F(b) - F(a)$
- $p(a < x < a + dx) = \int_a^{a+dx} f(s)ds = f(a)dx$
- $F(x + dx) = \int_{-\infty}^{x+dx} f(s)ds = \int_{-\infty}^{x} f(s)ds + \int_x^{x+dx} f(s)ds$
  $= F(x) + \int_x^{x+dx} f(s)ds = F(x) + f(x)dx$

To say that $F'(x) = f(x)$ is the same as saying that the slope of the tangent line to the graphic of $F(x)$ is $f(x)$. This is also equivalent to saying that $F(x + dx) = F(x) + f(x)dx$ or that $dF = F(x + dx) - F(x) = f(x)dx$.

162. ***Theorem for the design of generators of random numbers. Let*** X ***be a random variable with continuous probability density function*** $f(s)$ ***and cumulative function*** $F(x) = \int_{-\infty}^{x} f(s)ds$. ***Let*** u ***be an event of a random variable*** U ***with a uniform distribution in within zero and one, then*** $x = F^{-1}(u)$ ***is an event of a random variable whose cumulative function is*** F ***and its density function is*** $f(s)$.

Proof. To say that U obeys a uniform distribution in within zero and one is equivalent to saying that the probability of an interval, in the vertical axis of the graphic below, is equal to the length of the interval:

$p(du) = du$

and that this happens no matter in which point du is located. So, let us imagine that it is at $G(a)$, where G is a non decreasing function to be specified.

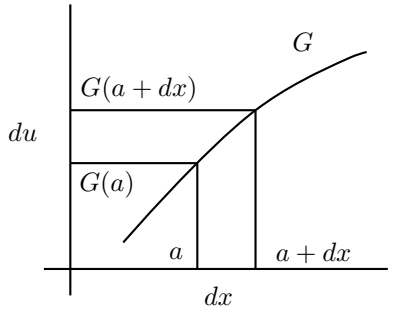


Figure 8.11. The probability of du is exactly equal to the probability of dx.

Thus, we have that $p(du) = du = G(a + dx) - G(a)$.

Our procedure takes u with a uniform distribution and reports $G^{-1}(u)$ as an event of X, whose cumulative function is F. So, we assign the same probability to du and to dx:

$p(du) = p(dx)$

Now,

$p(du) = du = G(a + dx) - G(a) = p(dx) = p(a < X < a + dx)$.

In short, we have proved that

$p(a < X < a + dx) = G(a + dx) - G(a)$

In other words, function G is a cumulative function of X. If we assume that the density function of X is continuous, it has only one cumulative function, so $G = F$, as demanded. For more complex instances, G and F might have some small differences.

163. ***Example. Fitting a Pareto distribution.***

We can notice that some few of our programs are large and that many are small and that the distribution does not begins from zero but from certain

minimal value, let it be m. The same happens with the size of the genome: relatively small genomes abound in the many species of bacteria while large genomes in plants and animals are relatively scare. One mathematical model of this situation is the Pareto distribution, whose probability density function is

$$f(x) = \begin{cases} 0, & \text{if } x < m, \\ \frac{rm^r}{x^{r+1}}, & \text{if } x \geq m, \end{cases}$$

The density function looks like this:

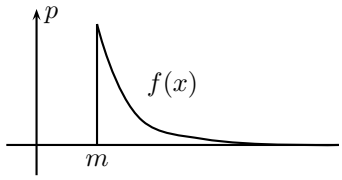


Figure 8.12. The general shape of the density function of a Pareto distribution for $r > 1$.

and its corresponding cumulative distribution, which is found by integration from $-\infty$ to x, is

$$F(x) = \begin{cases} 0, & \text{if } x < m, \\ 1 - (\frac{m}{x})^r, & \text{if } x \geq m, \end{cases}$$

The cumulative distribution looks like this

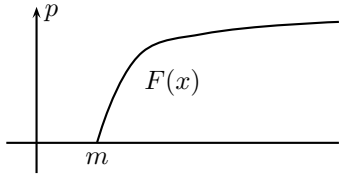


Figure 8.13. The general shape of the cumulative function of a Pareto distribution.

The inverse function of the cumulative distribution is found as follows:

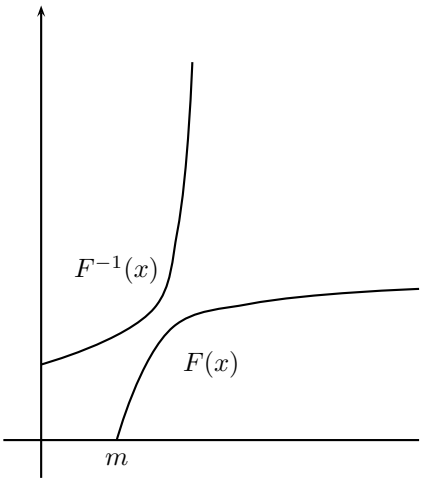


Figure 8.14. The general shape of the inverse of the cumulative function of a Pareto distribution.

To find the mathematical expression for F^{-1} we go over the next procedure:

$u = 1 - (\frac{m}{x})^r$

$u - 1 = -(\frac{m}{x})^r$

$1 - u = (\frac{m}{x})^r$

$(1 - u)^{1/r} = \frac{m}{x}$

$x = \frac{m}{(1-u)^{1/r}}$

164. ***The code that generates random numbers with a Pareto distribution.*** *We use the Java generator of random numbers to produce u with a uniform distribution in within zero and one and next we take*

$x = \frac{m}{(1-u)^{1/r}}$

We predict that x follows a Pareto distribution, a claim that we check with a chi-2-test. The code follows:

```
//Program E164 Pareto
//A  generator of random number of type double
//with a Pareto distribution.
//A test for this claim is included.
```

```
package ejvol5p;

import java.util.Random;

public class Pareto {
    //Parameters of the Pareto distribution
    //Minimal value
    private static final double M = 3;
    //Power
    private static final double R = 2;
    //Max nuMber of allowed classes
    private static final int MAXNCLASSES = 1000;

    private static int nClasses;
    private static final int SIZESIMULATION = 1234;
    private static final double EVENTS[]
            = new double[SIZESIMULATION];
    //Predicted distribution
    private static final double PREDICTED[] = new double[MAXNCLASSES];
    //Critical chi2 for 8 classes
    private static final double CRITICALCHI2 = 14.067;
    //Discrepancies between observed and predicted
    private static final double DISCREPANCY[] = new double[MAXNCLASSES];
    //Turn on of the random generator
    private static final Random RANDOM = new Random();
    private static final boolean PRINTALL = false;
    private static double min;
    private static double max;
    //The left border of intervals,   equal to the
    //minimum possible value.
    private static final double INFLIMIT = M;
    //Interval length
    private static double intervalLength;
    //Borders of the intervals
    private static final double BARRIERS[]
            = new double[MAXNCLASSES + 1];
    private static final double CLASSMARKERS[]
            = new double[MAXNCLASSES];

    //Table of frequencies
    private static final double FREQTABLE[][]
            = new double[MAXNCLASSES][2];

    //Prints data in vector
    private static void printVector(double Vect[], int lim) {
        for (int i = 0; i < lim; i++) {
```

```
            System.out.println(i + "    " + Vect[i]);
        }
    }

    //Prints the frequency table
    private static void printFreqTable(double FreqTable[][],
            int nClasses) {
        System.out.println("x and its absolute frequency");
        for (int i = 0; i < nClasses + 1; i++) {
            System.out.println(FreqTable[i][0]
                    + "     " + (int) (FreqTable[i][1]));
        }
    }

    //Calculates the sum    of absolute frequencies
    //of a frequency table
    private static void
            sum(double FreqTable[][], int nClasses) {
        double sum = 0;
        for (int i = 0; i <= nClasses + 1; i++) {
            sum = sum + FreqTable[i][1];
        }
        System.out.println("Registered numb of events = " + sum);
    }

    //Generates a random decimal number
    //with a Pareto distribution
    //with parameters m and r.
    private static double xPareto(double m, double r) {
        //a.nextDouble() generates a random decimal number
        //with a uniform distribution in within 0 and 1.
        double u = RANDOM.nextDouble();
        //Outputs are transformed by F inverse into x
        double one = 1;
        double x = m / Math.pow(1 - u, one / r);
        return x;
    }

    //Theoretical cumulative distribution
    //of the Pareto distribution
    private static double F(double x) {
        double c = 1 - Math.pow(M / x, R);
        return c;
    }

    //Predicted frequencies according to
```

```
    //theoretical distribution
    private static void prediction() {
        System.out.println("\n****TEST ZONE****");
        //System.out.println("Predicted frequencies");
        double sum = 0;
        double sum2 = 0;
        for (int i = 0; i < nClasses; i++) {
            double probSubI = F(BARRIERS[i + 1]) - F(BARRIERS[i]);
            /*System.out.println("xValue = " + BARRIERS[i] +
                      " probOfX = " + probSubI);*/
            PREDICTED[i] = probSubI * SIZESIMULATION;
            sum = sum + probSubI;
            sum2 = sum2 + PREDICTED[i];
        }
        System.out.println("\nPrediction:");
        System.out.println("Prob of everything = " + sum);
        System.out.println("Sum of frequencies = " + sum2);
    }

    //Runs a chi^2-test to check the accuracy
    //of our source of random numbers
    //that must fit the distribution given by Predict[].
    //First 8 classes are included.
    private static double chiTest() {
        double d = 0;
        for (int i = 0; i < 8; i++) {
            if (PREDICTED[i] > 0) {
                DISCREPANCY[i]
                        = Math.pow((FREQTABLE[i][1] - PREDICTED[i]), 2)
                        / PREDICTED[i];
            }
            d = d + DISCREPANCY[i];
        }
        return d;
    }

    //The range of discrepancies is evaluated
    //and is used to calculate the length of each
    //interval to group Discrepancies.
    public static double intervalLength(double Vect[],
            int nData,
            int nClasses) {
        min = 100000;
        max = 0;
        for (int i = 0; i < nData; i++) {
            if (Vect[i] < min) {
```

```
                min = Vect[i];
            }
            if (Vect[i] > max) {
                max = Vect[i];
            }
        }
        System.out.println("min = " + min);
        System.out.println("max = " + max);
        double range = max - min;
        double length = range / nClasses;
        return length;
    }

    //The border of each interval is calculated
    private static void getBorders(double Vect[]) {
        BARRIERS[0] = INFLIMIT;
        CLASSMARKERS[0] = INFLIMIT + intervalLength / 2;
        //Calculate maximal value of data
        double myMax = Vect[0];
        for (int j = 1; j < Vect.length; j++) {
            if (Vect[j] > myMax) {
                myMax = Vect[j];
            }
        }
        System.out.println("Max value = " + myMax);
        System.out.println("Number of classes = " + nClasses);
        //Calculates interval borders and class markers
        for (int j = 1; j <= nClasses + 1; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + intervalLength;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + intervalLength;
        }
        /*System.out.println( "Borders are" );
        printVector(BARRIERS);
        System.out.println( "Class markers are" );
        printVector(CLASSMARKERS);*/
    }

    //Data are grouped into interval classes.
    private static void group(double Vect[],
            int nData, int nClasses) {
        intervalLength = intervalLength(Vect, nData, nClasses);
        getBorders(Vect);
        //Class markers a re defined
        for (int j = 0; j <= nClasses + 1; j++) {
            FREQTABLE[j][0] = CLASSMARKERS[j];
            FREQTABLE[j][1] = 0;
```

```
        }
        //Frequencies are calculated.
        for (int i = 0; i < nData; i++) {
            for (int j = 0; j <= nClasses + 1; j++) {
                if ((Vect[i] >= BARRIERS[j])
                        & (Vect[i] < BARRIERS[j + 1])) {
                    FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
                }
            }

        }
    }

    //Grouping of events
    private static void grouping(double Vect[],
             int nClasses) {
        group(Vect, SIZESIMULATION, nClasses);
        System.out.println("Frequency table of events. ");
        printFreqTable(FREQTABLE, nClasses);
        //Minor test for correct grouping
        sum(FREQTABLE, nClasses);
        System.out.println("Simulated numb of events = "
                + SIZESIMULATION);
        System.out.println("\nFrequencies   of events"
                + "\nPaste it to Excel, OpenOffice or Gnumeric"
                + " and make a scatter char."
                + "\nIn OpenOfficer choose paste special, +"
                + " unformatted text, space separation + scatter chart.");
        for (int i = 0; i <= nClasses + 1; i++) {
            System.out.println(FREQTABLE[i][1]);
        }
    }

    public static void main(String[] args) {
        //Number of classes for simulated events
        nClasses = 40;
        System.out.println(" Simulated events");
        for (int i = 0; i < SIZESIMULATION; i++) {
            //A random decimal number with a Pareto
            //distribution is generated
            EVENTS[i] = xPareto(M, R);
        }
        if (PRINTALL) {
            printVector(EVENTS, SIZESIMULATION);
        }
        grouping(EVENTS,   nClasses);
```

```
        prediction();
        double dExp = chiTest();
        System.out.println("Predicted and observed values,"
                + "  chi2 discrepancy for 8 classes");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(PREDICTED[i] + "\t" + FREQTABLE[i][1]
                    + "\t" + DISCREPANCY[i]);
        }
        System.out.println("Ho: observed values fit "
                + "predicted ones");
        System.out.println("Experimental discrepancy for "
                + "8 classes = " + dExp);
        System.out.println("Critical chi2 = " + CRITICALCHI2);
        if (dExp > CRITICALCHI2) {
            System.out.println("The event is extreme "
                    + " according to Ho: invent another theory.");
        } else {
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }
    }//End of main
}//End of Program E164 Pareto
```

165. ***Exercise.*** *Run the program and play with the code.*

166. ***Challenge.*** *Let us measure the size of programs by the number of lines of its code. Does the size of programs of this volume fit a Pareto distribution?*

167. ***To know more.*** *Simulation of whatever distribution is by some reason considered to be a topic for experts and so they have developed professional software to fit in their needs. The R project ([82], 2010) is one answer ready to be used for free. To gave a look at another solution, also ready to use and free, inspect the ROOT site (Root, [87], 2010). Both projects are very similar, although ROOT is implemented over C while the R project has its R language on its own.*

8.8 Conclusion

The scientific method is built upon a very simple procedure: contrast what you see with what you think and if you notice a discrepancy that is small enough to be explained by uncontrolled factors, retain your thoughts else reject them and look for a better explanation. We have seen how this program is implemented over very simple but illustrative instances in which noise is included. The procedure is as follows: Step one: choose a given measure of the discrepancy between what you see and what you think and carry the measure out to get D_{exp}. Step

two: study the distribution of discrepancies between what you see in a virtual world fueled by your belief or null hypothesis and what you expect in that world. The purpose of that study is to determine what is normal and what is outlier in the space of discrepancies. Step three: judge D_{exp}, the discrepancy caused by experimental result: if it classifies as normal, accept the null hypothesis but invent and test new forms of measuring the discrepancy between what you see and what you think. But if D_{exp} classifies as outlier, reject the null hypothesis, invent a new theory and test it over and over.

Chapter 9

Evolvable software

`The style of life: evolution`

168. ***Purpose.*** *Development of software is a hard, very complex enterprise. So it is natural to try to alleviate it. The first recommendation is to reuse it: do not invent the round twice and much less trice. Simple as this advice could be it is one of the most powerful. Now, the crude reality is that software to be reusable needs a special style of design, a characteristic that nowadays is called evolvability because the paradigm for reuse is the genome and evolution is the usual biological term for smart reuse. Our purpose in this chapter is to propose and discuss some realizable and simple advises to achieve evolvability.*

9.1 Five different styles

There are many diverse programs that execute the same function. We know this because when one gives a task to a group of students, they come back with programs that in general are different. Or, if one loses a program and develops it again, the resultant version is different than the former one. It is also apparent that one can classify diverse versions according to the style, the manner as they have been composed. To understand this, let us compare the next five isofunctional programs.

169. ***The code for an entangled version.***

```
//Program A169 Entangled
//Simple task leading to entangled,
//difficult to understand code.
package ejvol5v2p;

public class Entangled {
```

```
    public static void main(String[] args) {
        int x = 2;
        System.out.print(" The sum of x = " + x + " and ");
        int y = 3;
        int z = x + y;
        System.out.println(" y = " + y + " is " + z);
    }
}//End of Program A169 Entangled
```

170. ***The code for a disentangled version.***

```
//Program A170 Disentangled
//A code that is clear.

package ejvol5v2p;

public class Disentangled {

    public static void main(String[] args) {
        int x = 2;
        int y = 3;
        int z = x + y;
        System.out.print(" The sum of x = " + x + " and ");
        System.out.println(" y = " + y + " is " + z);
    }

}//End of Program A170 Disentangled
```

171. ***The code for a semi structured version.***

```
//Program A171 SemiStructured
//Some tasks are isolated into methods.

package ejvol5v2p;


public class SemiStructured {

    public static int sum(int x, int y) {
        return x + y;
    }

    public static void main(String[] args) {
        //Declaration and initialization
```

```
        // of variables
        int x = 2;
        int y = 3;
        //Invocation of a method
        int z = sum(x, y);
        System.out.print(" The sum of x = " + x + " and ");
        System.out.println(" y = " + y + " is " + z);
    }
}//End of Program A171 SemiStructured
```

172. ***The code for a structured version.***

```
//Program A172 Structured2
//Each task into its own method.

package ejvol5v2p;

public class Structured2 {

    public static int sum(int x, int y) {
        return x + y;
    }

    public static void report(int x, int y, int z) {
        System.out.print(" The sum of x = " + x + " and ");
        System.out.println(" y = " + y + " is " + z);
    }

    public static void main(String[] args) {
        int x = 2;
        int y = 3;
        int z = sum(x, y);
        report(x, y, z);
    }

}//End of Program A172 Structured2
```

173. ***The code for a documented version.***

```
//Program  E173 Documented
//Same as Program E23
//Fully structuring and documenting a program

package ejvol5v2p;
```

```
public class Documented {
    //This is a method.
    // Its function is to sum two numbers
    // of type int, one is x, the other is y.
    //The result is declared as int.
    //The output of the method appears after
    // the reserved word "return".

    public static int sum(int x, int y) {
        return x + y;
    }

    //This method reports to the console
    // some information about x,y,z.
    // It does not report a result to the main
    // method, so its output is void.
    public static void report(int x, int y, int z) {
        System.out.print(" The sum of x = " + x + " and ");
        System.out.println(" y = " + y + " is " + z);
    }

    //This is the main method.
    //This is the head of the class.
    public static void main(String[] args) {
        //Declaration and initialization
        // of variables
        int x = 2;
        int y = 3;
        //Invocation of a method
        int z = sum(x, y);
        // Report to the console
        report(x, y, z);
    }
}//End of Program  E173 Documented
```

174. ***Exercise.*** *Run the 5 programs and verify that they fill exactly in the same function.*

175. ***Crucial and difficult exercise.***

a) Agree else disagree with the appropriateness of the names given to previous 5 programs.

b) Defend the thesis that we have here 5 different styles of software development.

c) Agree or disagree that it is possible to develop fully functional software using a tremendously entangled style, such as that of the first example. Nevertheless, no genome is like that: why? How?

d) Discuss the proposition that software cannot exist if not as the expression of a given style.

e) Does the number of styles grow with the complexity of programs, measured, say, by the number of lines of code?

f) Can we automatically distinguish among diverse styles?

g) Prove else refute that styles can be distinguished by their cost of implementation, say, in additional lines of code and in care to keep some rules.

h) Software can be developed by means of evolution. This is officially called ***genetic programming*** *(Koza, [61] 1996, [62] 2007). Propose a plausible description of the style that the resultant software will have.*

9.2 Styles and reusability

No matter which style one might adopt, one always look for **reusability**, for the possibility to use and reuse already developed software to fill in the needs that one finds along the way. In this regard, it is reasonable to think that the genome is and will be the epitome for reusability of complex software. Thus, reusability and **evolvability** can be understood as interchangeable terms, as the next paragraph shows.

176. ***The style of the genome***

Given that the genome is software, which is its style?

Our proposal begins by noticing that it is impossible to separate in modern science the genome from evolution. So, we claim that the genome is evolvable by its very nature and style. We say that **evolvable software** is one that is distinguished by its smart reuse, in which their parts are recurrently employed with or without modifications that include recombination to produce diversity and fine adaptation to specific tasks.

177. ***Engineering evolvability***

How to produce reusable, evolvable software is a matter of intense investigation with extensive research (Mens, [69] 2008; Higuchi et al, [46] 1997). All that sophisticated material produces changes and trends that are seen only with the course of decades and that possible boil down to very simple conclusions, as the next parable shows:

We use pipelines to conduct water, gas, oil. By instinct, we know that pipelines must be simple and direct. The very same is expected from software, which is a conduct for instructions, for commands, for directive information.

At this moment in time, it is cogent to notice that the expected qualities of a pipeline are not a duty, they correspond to a style, to free choices made by the designer, which not always is constrained by monetary or product restrictions.

Pipelines are made of tubes, whose full function can be inferred with a single sight an which are easy to concatenate. With units like those, a pipeline can stream out for thousand of kilometers, say, as those that join Siberia with Europe or with China. Using the same style in software development, we will be able to compose programs as large as needed, to fulfill in the most complex and restrictive imaginable functions.

Our elementary reflection orients us to enumerate some basic qualities of software to be evolvable, reusable, as follows:

178. ***Manageable, simple modules.***

The equivalent of a tube is a software unit, a subroutine, a procedure, a method, that is preferentially short, simple, fully parameterized and with an easy to read function which has been thoroughly tested, with commented strokes. We see here an invitation to work with structured documented software that shall be free from global variables.

179. ***Easy to concatenate modules***

The key to reuse is the easiness to concatenate units, of whatever hierarchy, into superunits. All these problems can be solved at once by the **function paradigm**: any program is a recursive function and must be nothing else (Akhmechet, [3] 2006; Jones, [52] 1987); Harper, [45] 2011). To be a function means that upon the same input always produces the same output. To be recursive means, colloquially speaking, that it belongs into an evolvable environment, fueled by reuse. All this is easy to say, but impossible to fulfill. To see why, let us consider the next method

```
public static int sum(int x, int y)
{
   return  x+y;
}
```

This method has a very clear input, two integers, x and y, and a very clear output, their sum, which is also an integer. Its concatenation into a greater unit is immediate:

```
int a= 2;
int b = 3;
int z = sum(a,b);
System.output.println(z);
```

This example, shows that, in principle, the function paradigm is the simplest and most complete synonym of evolvable software and therefore it is powerful for

the rapid achievement of every possible purpose. In consequence, this paradigm was early incorporated into programming languages as, say, FORTRAN, which means Formula Translating System. Nevertheless, this language was an implementation of the recursive paradigm at a recursive level. This means that one can compute a formula using one line of code with the same facility as one can concatenate various formulas to achieve a single great compound formula in a function that can be invoked from whatever place.

The next paragraph contains a correction to a misconception of the first version, in which Functional Programming was identified with Object oriented programming. These terms have official definitions that are not subject of personal interpretations.

To enable the function paradigm in programming beyond procedures, modules or methods, to the level of whole programs is a further achievement that has a glorious and terrifying name: **Functional Programming** that originally was the style of those that want to know more than everybody but that is gaining terrain in the industry because of its high level of security. The Java implementation can be learned from our `Vol XVI Functional Java`. Java arrived late to the functional paradigm (in 2015) because Java was built on another directive: complexity is the distinctive of software just because it must reflect the nature of real problems but this comes structured around objects, their interactions and modifications. This gave raise to the **OOP, Object oriented Programming**. Below, we will find a natural introduction to this theme. Warning: Java arrived late to Functional Programming but it was implemented on top of the OOP, which is native to Java. The final result is simply amazing.

Inexperts might think at this place that we are done. No. Things come at a high price: a great dose of suffering fills the gap in within beginners and experts. And, which is the lesson?

180. *The spaghetti code jungle*

Given the industrial importance of accelerating techniques for software development, many people try to understand and assimilate what evolvable software means. But experience shows that one usually ends with a program that resembles a 'big ball of mud', which 'is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle'(Foote and Yoder, [35] 1995).

Spaghetti jungles has nothing bad in themselves. The problem is that the design of software is very difficult a fact that operationally represents a lot of lost time in fixing **bugs**, i.e., errors against the purpose of the program. So, if one commends oneself to software development, one must know that bugs' fixing might represent 10% of wasted time in a small project a cipher that can grow up to 99% in some very complex projects. It is here that one begins to appreciate how good it will be to work outside those jungles and to experience the freedom for movement.

Now, why does everyone end revolving him or herself in the spaghetti jungle in spite of all efforts to the contrary?

The fundamental reason is very simple, ubiquitous and inescapable. It happens that the function paradigm to be human friendly needs the possibility to include void input and/or void output. This means that the given method lacks a return and so it produces no reusable output, as in the next example:

```
public static void report(int x, int y, int z)
{
 System.out.print(" The sum of x =" + x + " and ");
 System.out.println( " y = " + y + " is " + z);
}
```

or that it works upon global variables as the following:

```
public static void report()
{
 //x,y and z are globally defined variables
 System.out.print(" The sum of x =" + x + " and ");
 System.out.println( " y = " + y + " is " + z);
}
```

Now, the problem resides in the use of global variables. Why? Because a global variable, that is not explicitly declared in the input or output, plays the same role as free glue. Thus global variables are called **slack**. Global variables appear everywhere as in the next procedure:

```
 //Data are grouped into interval classes.
private static void group(double Vect[])
{
 getBorders(Vect);
 int nData = Vect.length;
 //Class markers a re defined
 for(int j=0; j < nClasses; j++)
 {
 FreqTable[j][0] = classMarkers[j];
 FreqTable[j][1] = 0;
 }
 //Frequencies are calculated.
 for(int i=0; i< nData; i++)
 {
for(int j=0; j < nClasses; j++)
{
  if ((Vect[i] >= Barriers[j]) & (Vect[i] < Barriers[j+1]))
  FreqTable[j][1] =FreqTable[j][1] +1;
}
```

```
 }
}
```

The function of this method is very clear: it groups some data that is conveyed by Vect[] into a frequency table. Its output is also clear: to produce a frequency table. Some global variables also appear: the number of classes of the frequency table and the form to partition the range of data.

Aforementioned global variables are like free ends that can accidentally get tied to whatever one less expects, entangling everything as in the spaghetti jungle. The remedy is simple: to declare everything, i.e., to produce a straight pipe with one single entrance and one single exit, as follows:

```
//Data are grouped into interval classes.
private static void group(double Vect[],
                          double FreqTable[][],
                          double Barriers[],
                          int nClasses)
{
 getBorders(Vect);
 int nData = Vect.length;
 //Class markers a re defined
 for(int j=0; j < nClasses; j++)
 {
 FreqTable[j][0] = classMarkers[j];
 FreqTable[j][1] = 0;
 }
 //Frequencies are calculated.
 for(int i=0; i< nData; i++)
 {
for(int j=0; j < nClasses; j++)
{
  if ((Vect[i] >= Barriers[j]) & (Vect[i] < Barriers[j+1]))
  FreqTable[j][1] =FreqTable[j][1] +1;
}

 }
}
```

Simple as this might be, it is most of times not done. Why? The reason is that one automatically divides things in two classes: those that must remain in focus and those that must run in background. This is a strategy to manage complexity and it is very fruitful. The eye is designed over that principle: the resolution of the retina is greater at the place where the eye is focused than sidewards.

So, we love a strategy that always leads into a spaghetti jungle. Can we minimize losses? There possibly exist many criteria to divide a world in a primary goal plus a background. The problem is that no one is interested in

knowing about them or about the best. One actually proceeds on intuitive grounds and one makes a revision in the only case of been forced by bugs.

But suppose that somebody is strange enough to faithfully follow the pipeline or function paradigm: does he or she get rid of complexity and bugs? A negative answer is expected by most people: to fit real problems that might be extremely complex, on must get involved in mutual recursive calls of elementary subunits whose complexity might go beyond human comprehension.

So, we have a great hero, evolution, and a great archenemy, complexity. They both run hand in hand and for your own evil you will forget one in favor of the other.

9.3 Intrigues about the genome

Our experience and considerations about the software produced by humans prompts some intriguing questions regarding the style of the genome.

181. ***The genome is documented***

The production of tryptophan is encoded in many bacteria by a group of genes that are used, or transcribed, together, so they conform an operon structure. It happens that this operon also encodes for very short peptide sequence that is rich in tryptophan. Something similar happens with phenylalanine and histidine (Darnell et al, [16] 1986). From our perspective, this looks as software documentation. How can this be explained?

182. ***The genome follows the function paradigm***

(This part of the chapter has been rewritten for version 2 of this volume because the original version contained a serious mistake: it was preached that an enzyme can contain one and only one active center. This is false. By the same reason, the following chapter was retouched.)

We have seen that the **function paradigm** is the dream of clean computing science, a dream that is transformed by every developer into a nightmare because of global variables and by the entanglement that ensues the abundance of mutual calls among methods. So, one might think that the functional paradigm cannot surpass the realm of abstract mathematics and sophisticated applications. But that is false: let us argue that the genome obeys the function paradigm without one single exception and not by some constraint but by a choice of style. The defense of our proposal is as follows:

Catalysis in the cell is to great extend executed by the enzymes. They are, in general, functionally compound assemblies and their subunits are called modules or domains. Enzymes are targeted to catalyze those reaction whose activated complexes dynamically fit the active site of the enzyme. Now, physic

chemical laws cannot interdict the existence of various active centers in enzymes. More to the point, there exist enzymes with various active centers, for example polyketide synthases:

"Polyketides form a large and structurally diverse class of natural product, mainly produced by soil-based bacteria, notably Streptomyces spp. They include clinically useful drugs such as the macrolide antibiotic erythromycin A, the immunosuppressants FK506 and rapamycin as well as antiparasitic and anticancer compounds. Many polyketides are produced by modular polyketide synthases. An increasing number of the gene clusters encoding these enzymes have been sequenced. By altering the DNA sequence of the genes encoding for the polyketide synthases we can produce polyketides of differing but predicted structure and enable the engineering of novel therapeutic drugs. Such interventions, to harness the cellular machinery for a specific novel purpose, is an important part of the emerging field of Synthetic Biology.

[...]Type I modular polyketide synthases (PKS) were identified in 1990. These giant catalytic enzymes are molecular assembly lines which contain multiple active sites on a single polypetide. In the case of erythromycin biosynthesis the polyketide macrocycle is produced by three enzymes DEBS1, DEBS2, and DEBS3, which function as a complex of molecular weight 2 MDa. Each protein contains numerous domains, each possessing catalytic activity to extend and alter the structure of the polyketide as it passes along the protein. The domains are grouped into extension modules. Each module specifies the chemical structure added to the growing polyketide at each stage." (Leadlay, [66] 2011)

Thus, we have enzymes that contain various active centers but each one of them in different modules. The point is that no one has ever reported an enzymatic module with more than one active site. If we additionally take into account that enzymes are in general highly specific, the overall result is that each enzymatic domain catalyze only one reaction under normal circumstances. That is how pathways arise and with them the biological identity of each species.

Warning: an enzyme can catalyze one reaction while processing different molecules, say proteases, which digest proteins by disrupting specific bonds but can do this in very different proteins.

So, the function paradigm reads: each one method, procedure or subroutine exactly implements one recursive function. By the same token, the catalysis paradigm for enzymes reads: if a domain catalyzes a reaction, it is unique with input and output compounds clearly specified in such a way that under normal conditions, the enzyme univocally determines as the inputs as the output compounds. Thus, one can design pathways by resorting to concatenation. This is precisely the function paradigm but applied not to the genome, which is software, but to the modules of the enzymes they encode for.

Now, where did this implementation of the function paradigm in molecular biology result from?

One can immediately claim that the function paradigm became the actual

style of the genome thanks to evolution through a race for high turnover that immediately leads to high specificity. That option might be appropriate for the international literature but not here in our community. By contrast, we must proceed like this to test the idea that the function paradigm becomes dominant in an evolutionary environment:

1. Make a synthesis of enzymatic modules with two or more active centers. This can be done in silico (in computer simulations), in vitro or in biotechnology with or without the help of evolution. There is no law against this possibility, therefore it can be done.
2. Prove that in a population relying on enzymatic modules with various active centers, those that rely on enzymatic modules with just one center become winners.
3. Prove that the evolutionary process leading to the dominance of enzymatic modules with one active center is instantaneous and determinant in geological time and so it left no tracks neither in the fossil record nor in remnants in actual populations.

183. ***We need quantitative characterization***

We have been driven by the feeling that every developer has his or her own personal style and that the number of different styles can be unbounded. We need automatic methods to register and recognize all that variability. Next, we will apply this methodology to the study of the genome, which looks to be highly organized, highly transparent, extremely evolvable, guided by the function paradigm and also documented. Such a style is one in a billion. Where does it come from?

9.4 Conclusion

The function paradigm in computing science teaches that any procedure or method in a program is the implementation of a recursive function. Functions can be concatenated, composed. So, this paradigm produces software that is naturally evolvable, reusable. But it is not a panacea because it has in itself its own poison: recursiveness immediately leads to extremely entangled systems of mutual calls that hinder evolvability. In general, we feel unable to recognize the existence of a remedy against complexity. Moreover, the function paradigm is never applied in practice because one usually works with globally defined variables. And this gives rise to untied ends that lead to a spaghetti jungle, the usual style of programming in real life. On the other hand, we have argued that the function paradigm is faithfully followed by the genome through the specificity of enzymatic domains and not by physic chemical constraints but by an election of style. In short, the genome looks to be highly organized, highly transparent, extremely evolvable, guided by the function paradigm and also documented. Such a style is one in a billion. Where does it come from? We hope to have posited very concrete and powerful challenges to our community.

Chapter 10

The chloroplast strategy

`Complexity comes structured`

184. ***Motivation and purpose.*** *We have been promoting the pipeline paradigm, our cartoon of Functinal Programming, which to be complete must include the possibility of accepting void inputs and/or outputs, a fact that allows to work upon globally defined variables. Nevertheless, we have arrived to the conclusion that our paradigm becomes weak against complexity if one allows that extension because one ends in a spaghetti jungle that interdicts an efficient fixing of bugs and further maintenance. In regard with the input, we know how to proceed: one makes a full and complete declaration of all inputs in the argument of the method. But what about complex outputs? All we have considered is outputs with integer and decimal numbers. But an output might be an array or matrix A together with a matrix B, a vector C, five integers and three strings. What do we propose for a case like this? In the present chapter, we discuss the wrong solution to our problem and next we introduce a smart one that has been adopted by Java and by other modern languages: it is the* ***OOP (object oriented programming)****.*

10.1 The wrong solution

We have been claiming that **loose, slack variables** that are globally defined converts programs into a spaghetti jungle that will eat you alive because of caused problems in the moment of fixing bugs or when slight modifications are attempted at.

In our language: slack variables are by their very essence contrary to directed evolution. Let us emphasized the adjective *directed*, which means that we are not dealing with natural evolution but with an evolution that serves the purpose of design and that ordinarily is really the design in itself.

We see here how to deal with slack variables. Those at the input are readily dealt with:

185. ***Clean inputs***

The next method contains a slack, globally defined, variable `FreqTable[][]`:

```
 //Calculates the mean   of
//a table of absolute frequencies
//The input FreqTable[][] is globally defined
private static double meanFreqTable()
{
 double mean;
 double sum = 0;
 double n = 0;
 int m = FreqTable.length;
 for ( int i = 0; i < m; i++ )
  {
 //Class marker times frequency
 sum = sum + FreqTable[i][0]*FreqTable[i][1];
     n  = n + FreqTable[i][1];
  }
 mean = sum / n;
 return mean;
}
```

To neutralize the globally defined variable `FreqTable[][]`, we declare it explicitly in the argument of the method:

```
 //Calculates the mean   of
//a table of absolute frequencies
private static double meanFreqTable(double FreqTable[][])
{
 double mean;
 double sum = 0;
 double n = 0;
 int m = FreqTable.length;
 for ( int i = 0; i < m; i++ )
  {
 //Class marker times frequency
 sum = sum + FreqTable[i][0]*FreqTable[i][1];
     n  = n + FreqTable[i][1];
  }
 mean = sum / n;
 return mean;
}
```

We see that this method fits perfectly into the bug free function paradigm because there is no slack variable neither in the input nor in the output. But, can this be repeated when outputs are not that simple?

186. ***Complex outputs***

In relation with the previous method, let us suppose that one wants not only the `mean` but also the value of the variable `sum` and also that of `n`. The wrong solution would be to extend the language with the capability to cover complex outputs as in the next would-be example:

```
//Java lacks the option for
//declaring outputs as indicated.
//This method calculates the mean,
//the pondered sum and the number of objects
//for a table of absolute frequencies.
//Input = FreqTable [][]
//Output: mean, sum,  n.
private static (double mean,
                double sum,
                double n)
                msnFreqTable(double FreqTable[][])
{
 double mean;
 double sum = 0;
 double n = 0;
 int m = FreqTable.length;
 for ( int i = 0; i < m; i++ )
  {
 //Class marker times frequency
 sum = sum + FreqTable[i][0]*FreqTable[i][1];
     n  = n + FreqTable[i][1];
  }
 mean = sum / n;
 return (mean, sum, n);
}
```

Now, this idea has not been implemented in Java. Why? The reason is that it is wrong. To understand why, let us think of a problem in the kitchen with a water filtration. One tries to solve the problem in most cases with an appropriate patchwork. But in old houses this will not work because the pipeline has gotten filtering prone. The only solution is to change the whole pipeline. By the same token, we can try patches here and there in our task of dealing with complex inputs, outputs and other problems to come. What is then the equivalent of a full replacement of the pipeline? It is to devise a technology that is definitively robust against the complexities associated with input, output and slack variables, a technology that is evolution-friendly.

10.2 A right solution: OOP(Object Oriented Programming)

To deal with that complexity, we have tried, very successfully, the modular approach and so we divided a whole program into small modules or methods. Now, there are infinitely many criteria to define modularity divisions, but we never have discussed even a single one: we proceed by instinct just following the natural structure of tasks. So, our intuitive principle reads: the structure inherent to a problem must be precisely reflected by a solution algorithm.

Our principle is possibly a reformulation of a systemic approach, whose purpose is to describe the world, real in the study of nature or virtual in problems of design, respecting its natural unity and inner structure with given divisions and subdivisions.

Let us remark: a correct management of complexity must reflect our intuitive perception of the problem in regard with unity and inner divisions.

We see in the next piece of code how this directive has been implemented in Java and and in all modern languages. We make our job in two steps. In the first, we solve the problem with the tools known to us and, in the second, we introduce a new technology which is the solution adopted by modernity.

187. ***Code to show the natural structure of a problem***

```
//Program E187 Substructure
//computes the mean,
//the pondered sum and the number of objects
//of a frequency table.
//The program shows
//the inner structuring of the problem,
//which is perfectly reflected in
//solution algorithm, in the code.

package ejvol5v2p;


public class Substructure {
    //Declaration of data:
    //observed value in the first coordinate
    //Absolute frequency in the second.

    private static final double FREQTABLE[][] = {
        {4, 3},
        {7, 6},
        {10, 10},
```

```
        {13, 15},
        {16, 4},
        {19, 2}};

    //Prints the frequency table
    private static void printFreqTable(double FreqTable[][]) {
        System.out.println("x and its absolute frequency");
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

    //Calculates the number of objects registered in
    //the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][]) {
        double n = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

    //Calculates the sum XF   of
    //a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][]) {
        double sum = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

    //Calculates the mean  of
    //a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][]) {
        double n = nFreqTable(FreqTable);
        double mean = sumXFFreqTable(FreqTable) / n;
        return mean;
    }

    public static void main(String[] args) {
```

```
        double n = nFreqTable(FREQTABLE);
        double sum = sumXFFreqTable(FREQTABLE);
        double mean = meanFreqTable(FREQTABLE);
        printFreqTable(FREQTABLE);
        System.out.println("Number of objects = " + n);
        System.out.println("the pondered sum is = " + sum);
        System.out.println("The mean is = " + mean);
    }
}//End of Program E187 Substructure
```

188. ***Exercise.*** *Run the program and play with the code.*

189. ***A route to paradise.*** *Revise each method to inspect for slack, globally defined variables. If you find someone, shunt it by declaring it explicitly in the argument of the method. Think of a better division of tasks that will result in a more natural and robust modularity. Work out a perfect congruency among names of methods and their function. Improve the documentation to fit in your personal needs and those of your students. Rest only when a full professional product is accomplished: it must be free of slack variables, divided in simple modules with a simple and clear functions and with appropriate names that are moreover clearly and concisely documented, with every method thoroughly tested, ready for use and for reuse. In one word, your product must be evolution friendly, evolution prone.*

The previous code, possibly improved by the work of the Reader, is pretty good. It would have served in the 1980's as a perfect example of professional software. But for the today standards, this piece of software suffers from a deficiency: the unity of the problem is lack and so the whole program is not ready for use nor for reuse, so it is not evolution friendly. Now, if the problem is because of separation, the solution is **encapsulation**, to gather pieces into a single container. The immediate implementation of this idea is just to posit a program in within another.

The idea of encapsulation of programs is as old as the genome: the chloroplast and the mitochondria have DNA on their own which represent computer programs but they are organelles of an exterior cell, which has its own DNA. The net fact is that we have a computer program in within another one.

The Java implementation of encapsulation is very sophisticated, so let us learn it in some few steps.

190. ***The chloroplast strategy, step one: just posit one class in within another.***

```
//Program E190 Cell
//We include a program or class in within another.
//Just as a box is put inside another one.
//The outer class is Cell. Chloroplast is the inner class.
//Elementary mandatory adjustments are made
//that the new program could be free of syntax errors
//but it is not, as yet, accepted by the compiler.


package ejvol5v2p;

public class Cell {

    //Chloroplast is an inner class.
    //Their variables and methods cannot be static.
    public class Chloroplast {
        //Declaration of data:
        //observed value in the first coordinate
        //Absolute frequency in the second.

        private final double FREQTABLE[][]
                = {{4, 3},
                {7, 6},
                {10, 10},
                {13, 15},
                {16, 4},
                {19, 2}};

        //Prints the frequency table
        private void printFreqTable(double FreqTable[][]) {
            System.out.println("x and its absolute frequency");
            int m = FreqTable.length;
            for (int i = 0; i < m; i++) {
                System.out.println(FreqTable[i][0]
                        + "\t " + FreqTable[i][1]);
            }
        }

        //Calculates the number of objects registered in
        //the table of absolute frequencies
        private double nFreqTable(double FreqTable[][]) {
            double n = 0;
            int m = FreqTable.length;
            for (int i = 0; i < m; i++) {
                n = n + FreqTable[i][1];
            }
```

```
            return n;
        }

        //Calculates the sum XF   of
        //a table of absolute frequencies
        private double
                sumXFFreqTable(double FreqTable[][]) {
            double sum = 0;
            int m = FreqTable.length;
            for (int i = 0; i < m; i++) {
                sum = sum + FreqTable[i][0] * FreqTable[i][1];
            }
            return sum;
        }

        //Calculates the mean   of
        //a table of absolute frequencies
        private double meanFreqTable(double FreqTable[][]) {
            double n = nFreqTable(FreqTable);
            double mean = sumXFFreqTable(FreqTable) / n;
            return mean;
        }

        //This main is in the inner class
        public void main(String[] args) {
            double n = nFreqTable(FREQTABLE);
            double sum = sumXFFreqTable(FREQTABLE);
            double mean = meanFreqTable(FREQTABLE);
            printFreqTable(FREQTABLE);
            System.out.println("Number of objects = " + n);
            System.out.println("the pondered sum is = " + sum);
            System.out.println("The mean is = " + mean);
        }
    }//end of inner class

}//end of main class and Program E190 Cell
```

191. ***Exercise.*** *Observe this piece of code: it points to our intention of converting a whole class or program into a unity that is ready for use and reuse from within a greater program. Verify that the resulting code is free from syntax errors. Explain what changes were made. Verify that the new program is as yet good for nothing because it is not accepted by the compiler.* *Answer*

192. ***Exercise.*** *Our project is to transform a given class into an inner one that must be used and reused form the greater class. So, the inner class does not make merits for having a main method. Remove it from the inner class and place it in its exterior, as part of the outer class. See what happens: is that change accepted by the Java compiler? Invent and make appropriate adjustments to achieve a fully functional program.* *Answer*

193. ***Exercise.*** *To achieve functionality, some part of the code of the inner class migrated to the outer one. Check the rationality of the done division. Prove, by trying other divisions, that there are many forms of doing that but that a best one possibly exists. Relate this with the corresponding problem about the chloroplast as an organelle of their hosting cells.* *Answer*

The developers of the language implemented a fully evolvable language: Java is not a finished language, instead, it can be extended in a personalized way, according to the desires and needs of the user. How was this achieved? By paying attention to a bit of abstraction that passes from an instance to a prototype. In biology this is quite simple: Are you hearing how that dog barks? It is a dog, it is an instance of the prototype Dog, which represents a species. In computing science, a species is called a `type`. So, to posit a class into another is equivalent to the definition of a new type that to be used needs to be instantiated in concrete objects or specimens. Let us see how this is done:

194. ***The chloroplast strategy, step three: prototypes that serve as new Java types.*** *We show here three things.*

1. *How one class is encapsulated into another, a possibility that enables rapid evolution towards complexity.*

2. *The naturalness and simplicity of communication among diverse classes. The idea is that different classes must communicate as cells in a biological organism. So, classes can be organized shoulder to shoulder, each one in its own file, but we have preferred to include one into another to facilitate learning. Classes has ears to acquire information, declared and specified in the constructors, and have mouths in each return of its inner methods.*

3. *Every class can be made into a prototype, a species. The world can be populated with individuals of the species in as many individuals as desired. This is done through instantiation. If a class is no instantiated, it is an* ***abstract class****.*

```
//Program E194 Cell3

package ejvol5v2p;

import java.util.Random;
```

```
//We include a program or class, Chloroplast,
//in within another, Cell3.
//The ensemble is appropriately coordinated
//to perfectly function.
//This is fully evolvable software,
//ready for use and for reuse,
//as tubes with respect to a pipeline.
//Full use of constructor methods is made.

public class Cell3 {



    //An outer variable is declared and initialized
    private static final double T1[][]
            = {{4, 3},
            {7, 6},
            {10, 10},
            {13, 15},
            {16, 4},
            {19, 2}};

    //*****************
    //This is a class in within another one,
    //it is an inner class.
    //It is prototype or a type.
    //Their variables and methods cannot be static.
    private static final class Chloroplast {

        //Variable in the inner class is declared
        //and initialized as a constant
        double FreqTable2[][] = {{20, 2},
        {32.5, 5},
        {37.5, 8},
        {42.5, 10},
        {47.5, 3},
        {60, 2}};

        //Variable is declared but not initialized
        //It will be used to connect the inner class
        //with the output of other methods of the outer class.
        //This is reuse, evolvability to the full
        double[][] FreqTable = new double[1000][2];

        //It is mandatory to initialize variables.
        //A non initialized variable is slack.
```

```
//A process of initialization is called a constructor.
//There could be various constructors.
//A constructor is designed for reuse,
//for initialization from varied sources,
//each one of which must be implemented on its own.
//A constructor represents a way as the class
//can acquire information.
//The class can hand over information through
//the return of their methods.
//Here we have a zeroed initialization:
Chloroplast() {
    for (int j = 0; j < 2; j++) {
        for (int i = 0; i < 1000; i++) {
            FreqTable[i][j] = 0;
        }
    }
}

//Initialization by cloning from matrix A
//with r rows and 2 columns
Chloroplast(double A[][], int r) {
    for (int j = 0; j < 2; j++) {
        for (int i = 0; i < r; i++) {
            FreqTable[i][j] = A[i][j];
        }
    }
}

//*****Methods of the inner class
//Prints the frequency table with m rows and 2 columns
private void printFreqTable(double FreqTable[][], int r) {
    System.out.println("\n***************\n");
    System.out.println("x and its absolute frequency");
    for (int i = 0; i < r; i++) {
        System.out.println(FreqTable[i][0]
                + "\t " + FreqTable[i][1]);
    }
}

//Calculates the number of objects registered in
//the table of absolute frequencies
//with m rows and 2 columns
private double nFreqTable(double FreqTable[][], int r) {
    double n = 0;
    for (int i = 0; i < r; i++) {
        n = n + FreqTable[i][1];
```

```
            }
            return n;
        }

        //Calculates the sum XF   of
        //a table of absolute frequencies
        //with m rows and 2 columns
        private double
                sumXFFreqTable(double FreqTable[][], int r) {
            double sum = 0;
            for (int i = 0; i < r; i++) {
                sum = sum + FreqTable[i][0] * FreqTable[i][1];
            }
            return sum;
        }

        //Calculates the mean   of
        //a table of absolute frequencies
        //with m rows and 2 columns
        private double meanFreqTable(double FreqTable[][], int r) {
            double n = nFreqTable(FreqTable, r);
            double mean = sumXFFreqTable(FreqTable, r) / n;
            return mean;
        }

    }//end of inner class

    //***** Outer class continues *************
    //*****New instance of Chloroplast*********
    //An object with a  freqTable
    //is instantiated by cloning from T1, in the outer class.
    private static final int J = 6;
    private static Chloroplast MYCHLOROPLAST =  new Chloroplast(T1,J);

    //The methods of the inner class are reused
    //by the outer class.
    private static void nsmFreqTable(double FreqTable[][], int r) {
    //a method of the inner class s is used
        double n = MYCHLOROPLAST.nFreqTable(FreqTable, r);
        double sum = MYCHLOROPLAST.sumXFFreqTable(FreqTable, r);
        double mean = MYCHLOROPLAST.meanFreqTable(FreqTable, r);
        MYCHLOROPLAST.printFreqTable(FreqTable, r);
        System.out.println("Number of events = " + n);
        System.out.println("the pondered sum is = " + sum);
        System.out.println("The mean is = " + mean);
    }
```

```
    //Main of outer class
    //Must be public and static
    public static void main(String[] args) {
        //An object with
        //a  freqTable is birth by zeroed initialization
        Chloroplast t =  new Chloroplast();
        //A zeroed FreqTable is processed
        int j = 10;
        nsmFreqTable(t.FreqTable, j);
        //A non zero FreqTable is processed
        j = MYCHLOROPLAST.FreqTable2.length;
        nsmFreqTable(MYCHLOROPLAST.FreqTable2, j);
        //Another specimen is declared and instantiated
        double[][]  myArray = new double[5][2];
        for (int i = 0; i < 5; i++)
            myArray[i][0] = i;
        for (int i = 0; i < 5; i++)
            myArray[i][1]  = new Random().nextInt(10);
        Chloroplast t2 =  new Chloroplast(myArray, 5);
        nsmFreqTable(t2.FreqTable, 5);
    }

}//end of main class and  Program E194 Cell3
```

195. ***Exercise.*** *Run the program and play with the code.*

196. ***Overloading.*** *The previous program contains two methods of initializing, one that fills the frequency table in zeros and the second by cloning from a given table. This possibility is native to Java and is called* ***overloading****: it is the possibility to use a family of different methods that process different inputs to fill in the same generic function. Example: we have three sums, one for integers, other for decimals numbers and another for strings. But actually we never have perceived that splitting because Java deals with all three types with an overloaded operator.*

197. ***Exercise.*** *Work out the general scheme of the chloroplast strategy. It must serve as a template for your own programs.* *Answer*

198. ***Research.*** *The just introduced technology is the renown* ***OOP****, Object Oriented Programming. It has two important extensions that respond to usual needs. First: one can divide a large program using many objects or classes in many files, so, one always has a tiny portion of code what to concentrate in and compilation goes swiftly because non modified files are not recompiled. Second: with the possibility to define new prototypes, we are producing speciation of our programming language, and so there is an automatic mechanism to deal with subspeciation. The official name for that process is* ***inheritance****.*

Let us recall now that we had a concrete problem: how to deal with complex outputs without generating slack variables. We forgot that problem for the sake of a new paradigm to solve a greater problem, that of full evolvability that resulted from encapsulation and prototyping. It is time to show that our new paradigm also solves the problem of dealing with complex outputs and allows for nice solution of complex inputs.

199. ***The code that shows encapsulation of complex inputs and outputs.***

```
//Program E199 ComplexOutput
//Encapsulation of complex inputs and outputs

package ejvol5v2p;

public class ComplexOutput {


    //Data
    private static final double L[] = {1, 2, 3, 4, 5};

    //******Inner class definition*******
    //This inner class defines a new type: d1Vector
    //It converts a 1d-array into a class or object.
    //d1Vector contains two items:
    //the length of the array, length
    //and the array itself, F[].
    private static final class d1Vector {

        int length;
        double F[] = new double[2000];

        //An instance of d1Vector can be
        //initialized in various ways:
        //first: Automatic zeroed initialization
        d1Vector(int l) {
            length = l;
            for (int i = 0; i < length; i++) {
                F[i] = 0;
            }
        }

        //Second: Initialization from  a 1d- array
        d1Vector(double[] A, int m) {
            length = m;
```

```
            /*
            for (int i = 0; i < length; i++) {
            F[i] = A[i];
            }
             */
            System.arraycopy(A, 0, F, 0, length);
        }

        //Third: Initialization by cloning from a d1Vector
        d1Vector(d1Vector A) {
            length = A.length;
            /*
            for (int i = 0; i < A.length; i++) {
            F[i] = A.F[i];
             */
            System.arraycopy(A.F, 0, F, 0, A.length);
        }

        //The class d1Vector is essentially  endowed
        //with a printing facility
        private void printVector(d1Vector Vect, String s) {
            System.out.println(s);
            for (int i = 0; i < Vect.length; i++) {
                System.out.println(Vect.F[i]);
            }
        }
    }//end of inner class

    //****Outer class continues**********
    //Initialization of a d1Vector from an array
    private static final d1Vector LIST1 =  new d1Vector(L, L.length);

    //Initialization of a d1Vector from a d1Vector
    private static d1Vector list2 =  new d1Vector(LIST1);

    //Every entry of a d1Vector is squared.
    //The input is a d1Vector, the ComplexOutput is also  d1Vector.
    //
    private static d1Vector V2(d1Vector V) {
        //Zeroed initialization of a d1Vector
        d1Vector list =  new d1Vector(V.length);
        //The square of each entry is calculated
        for (int i = 0; i < V.length; i++) {
            list.F[i] = V.F[i] * V.F[i];
        }
        return list;
```

```
    }

    //Main of outer class
    //Must be public and static
    public static void main(String[] args) {
        System.out.println();
        LIST1.printVector(LIST1, "Original data");
        list2 = V2(LIST1);
        System.out.println();
        list2.printVector(list2, "Squared data");
    }
}//end of outer class and Program E199 ComplexOutput
```

200. ***Exercise.*** *Run the program and play with the code.*

201. ***Exercise.*** *Modify the previous code to develop a program without slack or globally defined variables to calculate the variance of a list of data. Use the next formulas for the mean and variance:*

$$\bar{X} = \frac{\Sigma x_i}{n} = \frac{\Sigma x}{n}$$

$$s^2 = \frac{\Sigma (x_i - \bar{X})^2}{n-1} = \frac{\Sigma (x - \bar{X})^2}{n-1}$$ *Answer*

10.3 Conclusion

We have identified evolution with smart reuse. The top-down version of reuse, in reference to encapsulation, is that it is the style of problem solving by recursively dividing a task in sub-tasks that might be solved by specific encapsulated packages of code. The bottom-up view is that one can encapsulate pieces of code into modules that in their turn can be encapsulated in higher level packages. The whole business is to enable tinkering to the full. So, evolution runs at two levels. In the top one, an evolutionary environment is designed as powerful as possible. At the low level, evolution runs by randomness and self-organization by tinkering of the elements of the evolutionary environment. The enabling technology is the very famous OOP, Object Oriented Programming. While we have highlighted the importance of correct packing, OOP is also famous for its emphasis in communications: objects must communicate among themselves just as cells in an organisms. Java is an object oriented programming language and this means that the object approach is both **enabled and necessary** for complex projects. We have clearly shown why evolution is in computing science a style of problem solving that has been engineered and consciously crafted for complete recursiveness. This paradigm implements the biological concepts of species and specimens quite naturally.

Chapter 11

Numerical analysis over the Z

Introduction to numeric integration

202. ***Introduction and motivation.*** *A critical value separates normal events form extreme ones when a distribution and a significance level are given. We know how to use simulations to determine them but they were targeted to understand the innermost structure of science but not to make fast calculations. For this new purpose, we have mathematical formulas for the respective probability density functions. To use those functions to calculate critical values, we need numerical integration. This is a quite well developed and sophisticated discipline that has nevertheless some points that can be easily grasped and that we will learn to use in this chapter. We change the study of critical values for that of p-values, which is more powerful.*

11.1 p-values

The formalism of p-values is simple and more suitable to take decisions than critical values.

203. ***Definition.*** *The* ***p-value*** *of an event is the probability to find an event more extreme than it. The p-value can be calculated with one tail or with two.*

204. ***Example.*** *We have that the pH of the blood is neutral and that it is strongly buffered for stability, with a normal distribution in healthy people of mean 7 and deviation 0.01. John feels himself in bad condition and it was found to have a blood with pH of 7.03, which was measuring after 12 hours of fasting. Can we say that its bad condition is possible due to a departure of the pH of his blood away from the used neutrality? The question is formulated with two tails because the interest is directed to consider a departure form neutrality, a situation that can happen when the pH increases as well as when it decreases.*

The z associated to 7.03 is $z = (7.03-7)/0.01 = 3$ *and by consulting a table, one arrives to the conclusion that the probability of being more extreme with two tails than this event is 0.0026. This means that in 10000 people, only 26 have a pH of their blood more extreme than John. Because this probability is considered small, and in particular less than 0.05, most doctors would decide to treat his blood to stabilize its pH before going further.*

A critical value separates normal events form extreme ones when a distribution and a significance level are given. So, to decide a null hypothesis, all we need to know is whether or not the considered event is more extreme that the critical one according to the significance level and the distribution of the used statistic. We also can use p-values to decide null hypothesis:

205. ***Receipt for accepting or rejecting a null hypothesis using p-values.*** *The null hypothesis is rejected if and only if an event is extreme with respect to a significance level* α*. Now, the p-value associated to an event represents the proportion of events that are more extreme than it. So, if* $p-value < \alpha$*, our event is more extreme than the critical value associated to* α *and the null hypothesis shall be rejected. Otherwise, one is allowed to accept the null hypothesis.*

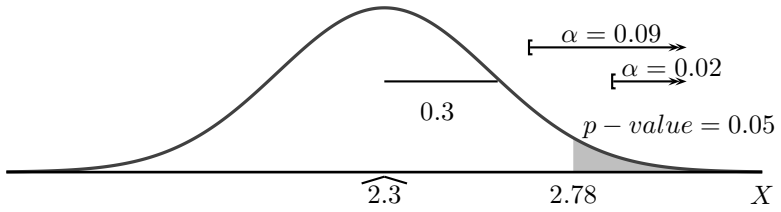


Figure 11.0. The bell depicts the probability density function of a normal distribution with mean 2.3 and deviation 0.3. The p-value of the event 2.78 with one tail is the area of the shadowed region corresponding to the proportion of all events that in the upper tail are more extreme than 2.78. For this case and the upper tail, the p-value is 0.05. So, for the event 2.78 and for the significance level 0.02, the null hypothesis $H_o : \mu = 2.3$ *must be accepted because in that case the event would be a normal one, but it must be rejected for* $\alpha = 0.09$ *because in that case the event would be extreme. We see that the p-value can be used to decide a given hull hypothesis no matter what significance level one might choose.*

To use probability density functions to calculate p-values, we need to calculate areas under them, a task that is called integration. In general, there is no analytical procedure to calculate resultant integrals and so we need to recur to numerical analysis and trickery, an introduction to which follows.

11.2 The Riemann and trapezoidal rules

There is no close formula to calculate a definite integral under a Gauss Bell or the other density functions t, χ-square or F. Therefore, we need approximation schemes to get rid of this and related problems. This thematic is a very complex affair that is part of a profession: numerical analysis. Our purpose is to implement some simple to understand but effective methods.

206. ***The method of Riemann sums***

In this method, one approximates the target area by blocks of the same width but with varying height that is adjusted to the curve, say, at the left border. In that case, we have the next approximation:

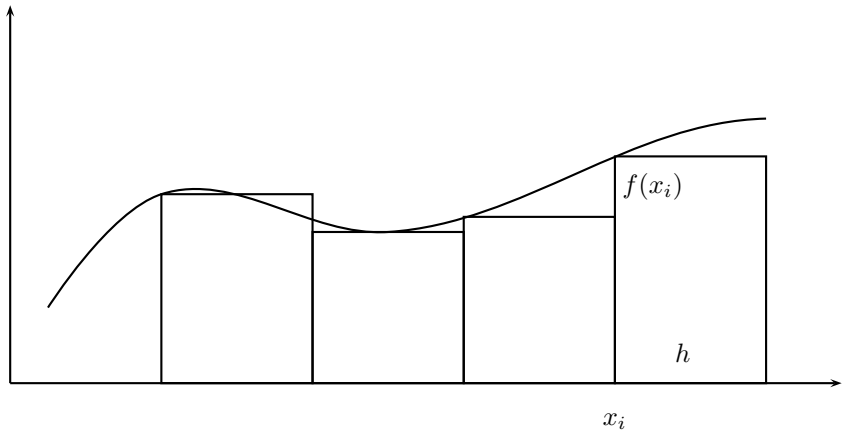


Figure 11.2. The area under a curve can be approximated by the sum of the areas of those blocks whose heights are determined by left evaluation.

$$\int_0^x f(s)ds \simeq \sum_{i=1}^n f(x_i)h = h\sum_{i=1}^n f(x_i)$$

We have a very simple and easy to implement formula. To improve precision, one might increase n. Going that way, one can attain, in principle, absolute precision.

207. ***Exercise.*** *Develop a Java program to calculate a definite integral using Riemann sums. Run appropriate tests to make sure that you can improve the precision as much as desired.* *Answer*

The method of the Riemann sums is simple to understand but the corresponding program can take some time to be run. What else do we have?

A simple but very effective variation is to approximate the figure not by blocks but by trapezes.

208. ***The method of the trapezoidal rule***

Let us pay attention to another method that is direct both to calculate and to understand: the trapezoidal rule (Burden and Faires, [12] 1985) and that is expected to produce better results in less time than the method of Riemann sums:

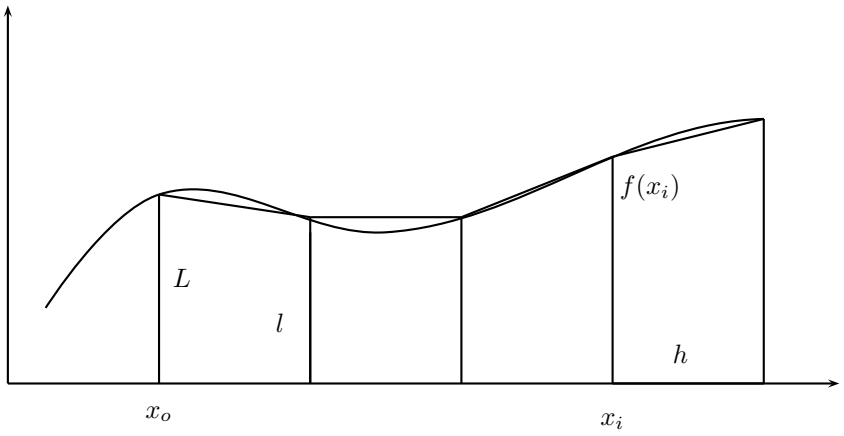


Figure 11.3. The area under a curve can be approximated by the sum of the areas of trapezes, all with the same width, h, and whose heights are determined by a partition set.

Let us number the coordinates of the division points along the X-axis as x_o,.. x_i.. .. x_n.

The area of a trapeze is

$a = h(L + l)/2 = h(f(x_{i-1}) + f(x_i))/2$

where L and l are the heights defined by the function at the extremes of the interval whose length is h and that begins in x_{i-1} and ends in x_i.

We have that the total area equals

$A = h(f(x_0) + f(x_1))/2 + h(f(x_1) + f(x_2))/2 + ... + h(f(x_{n-1}) + f(x_n))/2.$
$A = (h/2)[f(x_0)+f(x_1)+f(x_1)+f(x_2)+f(x_2)+f(3)+....+f(x_{n-1})+f(x_n)]$
$A = (h/2)[f(x_o) + 2\sum_{j=1}^{n-1} f(x_j) + f(x_n)]$

So, we get the very nice formula:

$$\int_{a=x_o}^{b=x_n} f(s)ds \simeq (h/2)[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)]$$

11.3 Lab over the z-distribution

Let us apply our methodologies to calculate the integral from zero to certain x below the standardized Gauss Bell, whose density function is

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

(the minus sign is a correction)

Therefore,

$\int_0^z f(x)dx = \frac{1}{\sqrt{2\pi}} \int_0^z e^{-x^2/2}dx$

which is approximated according to Riemann by

$h(\frac{1}{\sqrt{2\pi}}) \sum_{j=0}^{n-1} f(hj) = h(\frac{1}{\sqrt{2\pi}}) \sum_{j=0}^{n-1} e^{-(jh)^2/2}$

and according to the Trapezoidal rule as

$(h/2)[f(0) + 2\sum_{j=1}^{n-1} f(x_j) + f(z)]$
$= (h/2)[\frac{1}{\sqrt{2\pi}} + 2\frac{1}{\sqrt{2\pi}} \sum_{j=1}^{n-1} e^{-(jh)^2/2} + \frac{1}{\sqrt{2\pi}} e^{-z^2/2}]$
$= \frac{h}{2}(\frac{1}{\sqrt{2\pi}})[1 + 2\sum_{j=1}^{n-1} e^{-(jh)^2/2} + e^{-z^2/2}]$

209. ***The code*** *The Riemann and Trapezoid rules are implemented. We calculate the integral under the standardized Gauss Bell from zero to certain positive z. This area is called zHalfBody because the bell is symmetric and we take half of it without tails. The code follows:*

```
//Program E209 TheZeta
//Computes the  integral under the standard Gauss bell
//in within 0 and a positive z.
//Two methods are implemented: Riemann sums and
//the trapezoidal rule.

package ejvol5v2p;

public class TheZeta {

    private static double PI = 3.14159265358979323846;
    private static double z;

    //Returns the integral under the standard Gauss bell
    //in within 0 and z. Precision 1/N.
    //Riemann sums.
    private static double zHalfBodyRie(double z, long N) {
```

```
    /**
     * With N = 10^5, and for z=1 the algorithm achieves 5 correct ciphe
     */

    double h = z / N;
    double k = 1 / Math.pow(2 * PI, 0.5);
    N = (long) (N * z);
    double sum = 0;
    for (int j = 0; j < N; j++) {
        sum = sum + Math.exp(-h * h * j * j / 2);
    }
    sum = sum * k * h;
    return sum;
}

//Returns the integral under the z-density function
//in within 0 and z. Precision=1/N.
//Trapezoidal rule.
private static double zHalfBodyTrap(double z, long N) {
    /**
     * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciph
     */

    double h = z / N;
    double k = 1 / Math.pow(2 * PI, 0.5);
    double sum = 0;
    for (int j = 1; j < N; j++) {
        sum = sum + Math.exp(-h * h * j * j / 2);
    }
    sum = 2 * sum + 1 + Math.exp(-z * z / 2);
    sum = sum * k * h / 2;
    return sum;
}

//We compare here two methods: Riemann sums
//and the trapezoidal rule.
private static void demo() {
    System.out.println("Area under the standardized  "
            + "bell in within zero and 1.");
    System.out.println("N in 10^N, Riemann sums, "
            + "        Trapezoidal rule");
    int N;
    z = 1;
    double answer1, answer2;
    for (int i = 4; i < 7; i++) {
        N = (int) Math.pow(10, i);
```

```
            answer1 = zHalfBodyRie(z, N);
            answer2 = zHalfBodyTrap(z, N);
            System.out.println(i + "\t" + answer1
                    + "\t" + answer2);
        }
        System.out.println("The answer of Gnumeric  is     "
                + " 0.34134474606854 (for z= 1)");
    }

    public static void main(String[] args) {
        demo();
    }
}//End of Program E209 TheZeta
```

210. ***Exercise.*** *Play with the code and verify that it is correctly implemented. Decide which method is better: Riemann sums else the Trapezoidal rule.* *Answer*

211. ***Exercise.*** *Rewrite the code such that it could be reusable at once to calculate other distributions.* *Answer*

212. ***Exercise.*** *Reuse previous program to calculate the p-value associated to the event z. Produce answers for both one and two tails.* *Answer*

Part of the game is to compare our results with those produced by professional and famous packages. Gnumeric is one of them, which is renown by its accurateness and is moreover free. To better enjoy this game, let us learn how to use Gnumeric to calculate desired functions.

11.4 Comparing with Gnumeric

Gnumeric is spreadsheet calculator that functions just like Calc of LibreOffice or like Excel of Microsoft Windows. It can be readily downloaded from its official website (Gnumeric, [42] 2017). To calculate a given function, one goes to Insert, next to Function, next to statistical and then one looks for the desired function. Else, one can learn the next few commands:

1. For z: the instruction

   =normsinv(1- 0.4)

   returns the critical z for the significance $\alpha = 0.4$ with the upper tail. In this case, the returned value is

   1.75068607125217.

   The instruction

   =1-normsdist(zExp)

returns the significance of the experimental z_{exp} in the upper tail, i.e, the probability to find a z that is greater that the z_{exp}. Example: the significance in the upper tail of $z_{exp} = 1.75068607125217$ is 0.4.

2. The t depends on the parameter df, the degrees of freedom. To find the critical t with two tails for the significance 0.05 and 500 degrees of freedom, one types

   =tinv(0.05,500)

   and one receives in return 1.96471983746779. We see that with 500 degrees of freedom, the t functions just like the z.

   The instruction

   =tdist(1.96,50,2)

   returns the significance of the experimental t_{exp} 1.96, with 50 df and with 2 tails. The ouput values is 0.05558087405522, which means that with 50 df, the deviation of the sample can be considered as a wonderful estimation of the deviation of the population and therefore the t gives a value very similar to that given by the z. The instruction

   =tdist(1.96,50,1)

   gives the experimental t_{exp} 1.96, with 50 df and the upper tail and that is

   0.02779043702761.

3. The F:

   =finv(0.01,7,5)

   returns the critical value at the upper tail for the significance 0.01 with 7 df in the numerator and 5 in the denominator. The answer is 10.4555108917609.

   The instruction

   =fdist(10.4,7,5)

   gives the significance at the upper tail of the experimental value $F_{exp} = 10.4$ with 7 df in the numerator and 5 in the denominator. The answer is 0.01.

4. The chi-square χ^2: To find the critical value at the right tail with significance 0.01, and 30 df, one types

   =chiinv(0.01,30)

   which outputs 50.892.

   The instruction

   =chidist(50.892,30)

   gives the significance at the right tail of the experimental value χ_{exp} equal to 50.892 and 30 df, which is 0.01.

To know the left value of the critical chi-square value at the left tail with significance level of 0.025 and 8 df one types

=chiinv(0.975,8)

whose return is 2.17973074725265.

213. ***Challenge.*** *Gnumeric functions better in Linux than under Windows. This is not an isolated case because many programs that are native to Linux function better there. If someone considers that it is time to put an end to the sweet slavery to Microsoft and its potent products, he or she has a good opportunity now: there are various Linux versions, which are free and very accessible to the lay public. A good distribution for all levels is Ubuntu. Very accessible for beginners. For more experienced users we have OpenSUSE, which has an application, Yeast2, that helps the user to install packages. It is powerful and easy to use for every common computer task and is under permanent improvement.*

214. ***Exercise and challenge.*** *Verify that Gnumeric is faster than our program to calculate results. Hint: divide the region of integration in* 10^{15} *subintervals to get the precision of Gnumeric. Notice that while Gnumeric answers immediately, our methods must work during extremely long periods of time. This means that there must be other numerical receipts apart from the trapezoidal rule. Would you imagine one?*

11.5 Conclusion

Some few ideas of geometric origin have allowed us to swiftly develop an algorithm for calculating p-values. While we have worked with the Z distribution, our code is easily reusable for the study of other distributions.

Chapter 12

A quest for perfection

`High accurateness + instant calculation`

215. ***Motivation and purpose.*** *We have seen how easy and fun was to achieve accurateness in the task of calculating p-values by resorting to numerical integration of probability density functions. But we have discovered that our methods are very slow as compared with modern standards: results must be calculated with high accuracy and displayed in least than the twinkling of an eye. In fact, that is the way as Gnumeric functions. Our purpose here is to try out our own path to perfection, to professional software. Nevertheless, we find suffering in our own flesh a martyrdom. So, we abstract a general rule: to achieve incipient or even good quality is something rather easy but to achieve perfection is extremely complicated for the generality of interesting tasks. This generalization is certainly true for us, human beings. Would yo dare to extend it to cover evolution?*

Having working with the trapezoidal rule, we begin our quest for accuracy and velocity trying a further refinement, which is also a geometry oriented one.

12.1 The Simpson's rule

The trapezoidal rule uses lines to approximate the required integral. It is natural to ask: what can we achieve if we approximate the curve by parables, quadratic polynomials? A very beloved solution is known to us, whose ideas are as follows.

216. ***Interpolating functions***

A quadratic polynomial is uniquely defined by 3 points with 3 different x-coordinates. A natural choice of those coordinates for a given interval $[a, b]$ is composed of points a, b and the middle point $m = (a + b)/2$. We ought to Lagrange the discovery that the asked polynomial can be found by inspection:

$$p(x) = f(a)\frac{(x-m)(x-b)}{(a-m)(a-b)} + f(m)\frac{(x-a)(x-b)}{(m-a)(m-b)} + f(b)\frac{(x-a)(x-m)}{(b-a)(b-m)}$$

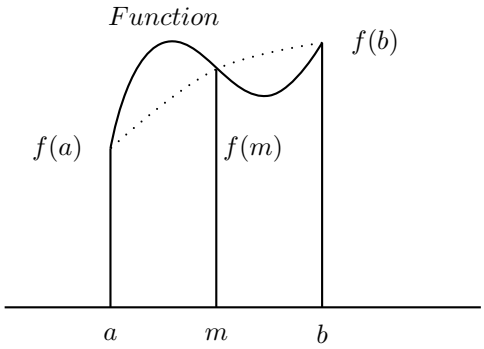


Figure 12.1. To ease the integration procedure, a complex curve, in solid line, is approximated by a 2-degree polynomial, a parable, punctured line.

In fact, this function takes the value $f(a)$ over $x = a$, $f(b)$ over b and $f(m)$ over m and is a sum of second degree polynomials.

Let us find the area or integral of $p(x)$ over the interval $[a, b]$. To that aim, let use the next assignations.

$A = f(a)\frac{1}{(a-m)(a-b)}$
$M = f(m)\frac{1}{(m-a)(m-b)}$
$B = f(b)\frac{1}{(b-a)(b-m)}$

The polynomial reads now:

$p(x) = A(x-m)(x-b) + M(x-a)(x-b) + B(x-a)(x-m)$
$= A(x^2 - bx - mx + bm) + M(x^2 - ax - bx + ab) + B(x^2 - ax - mx + am)$

The integration yields:

$\int_a^b p(x)dx = [A(x^3/3 - bx^2/2 - mx^2/2 + bmx) + M(x^3/3 - ax^2/2 - bx^2/2 + abx) + B(x^3/3 - ax^2/2 - mx^2/2 + amx)]_a^b$

$= [A((b^3-a^3)/3 - b(b^2-a^2)/2 - m(b^2-a^2)/2 + bm(b-a)) + M((b^3-a^3)/3 - a(b^2-a^2)/2 - b(b^2-a^2)/2 + ab(b-a)) + B((b^3-a^3)/3 - a(b^2-a^2)/2 - m(b^2-a^2)/2 + am(b-a))]$

$= (b-a)[A((b^2+ab+a^2)/3 - b(b+a)/2 - m(b+a)/2 + bm) + M((b^2+ab+a^2)/3 - a(a+b)/2 - b(a+b)/2 + ab) + B((b^2+ab+a^2)/3 - a(b+a)/2 - m(b+a)/2 + am)]$

Let us work these expressions separately. Recalling that $m = (b + a)/2$, we get for the first expression:

$A((b^2+ab+a^2)/3 - b(b+a)/2 - m(b+a)/2 + bm)$
$= A((b^2+ab+a^2)/3 - b(b+a)/2 - (b+a)^2/4 + b(b+a)/2)$
$= A((b^2+ab+a^2)/3 - (b+a)^2/4)$
$= A(4b^2+4ab+4a^2-3b^2-3a^2-6ab)/12)$
$= A(b^2+a^2-6ab)/12 = A(b-a)^2/12.$

Since

$A = f(a)\frac{1}{(a-m)(a-b)}$ and $m = \frac{a+b}{2}$, we get $a - m = a - \frac{a+b}{2} = \frac{2a-a+b}{2} = \frac{a-b}{2}$

Therefore

$A(b^2+a^2-6ab)/12 = A(b-a)^2/12 = f(a)\frac{1}{(a-m)(a-b)}(b-a)^2/12$
$= f(a)\frac{1}{(\frac{a-b}{2})(a-b)}(b-a)^2/12 = f(a)/6.$

Now,

$M((b^2+ab+a^2)/3 - a(a+b)/2 - b(a+b)/2 + ab)$
$= M((2b^2+2ab+2a^2-3a(a+b)-3b(a+b)+6ab)/6)$
$= M((2b^2+2ab+2a^2-3a^2-3ab-3ab-3b^2+6ab)/6)$
$= M(-a^2+2ab-b^2)/6 = M(-(a+b)^2/6) = f(m)\frac{1}{(m-a)(m-b)}(-(a+b)^2/6)$
$= f(m)\frac{1}{(\frac{b-a}{2}\frac{a-b}{2})}(-(a+b)^2/6) = 4f(m)/6.$

Similarly

$B((b^2+ab+a^2)/3 - a(b+a)/2 - m(b+a)/2 + am) =$
$= B((b^2+ab+a^2)/3 - ab/2 - a^2/2 - mb/2 - am/2 + am)$
$= B((b^2+ab+a^2)/3 - ab/2 - a^2/2 - (a+b)b/4 - a(a+b)/4 + a(a+b)/2)$
$= B((b^2+ab+a^2)/3 - ab/2 - a^2/2 - ab/4 - b^2/4 - a^2/4 - ab/4 + a^2/2 + ab/2)$
$= B(4b^2+4ab+4a^2-6ab-6a^2-3ab-3b^2-3a^2-3ab+6a^2+6ab)/12)$
$= B(b^2-2ab+a^2)/12 = f(b)\frac{1}{(b-a)(b-m)}(b-a)^2)/12$
$= f(b)\frac{1}{(b-a)(\frac{b-a}{2})}(b-a)^2)/12 = f(b)/6$

Adding up all 3 expressions, we end with:

$\int_a^b p(x)dx = \frac{b-a}{6}[f(a) + 4f(m) + f(b)] = \frac{b-a}{6}[f(a) + 4f(\frac{a+b}{2}) + f(b)].$

The astonishing fact is that the integration of the quadratic approximation does not contain quadratic or cubic terms.

So, if we apply this formula to approximate the area under a curve $f(x)$ in within $a = x_i$ and $b = x_{i+2}$ with $m = x_{i+1}$, we get: $b - a = 2h$, $f(a) = f(x_i)$, $f(m) = f(x_{i+1})$ and $f(b) = f(x_{i+2})$. So:

$\int_{x_i}^{x_{i+2}} p(x)dx = \frac{2h}{6}[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})]$

$$= \frac{h}{3}[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})]$$

Therefore, if we aggregate all terms in a given partition with an even number of intervals, we get:

$$\begin{aligned}\int_b^a p(x)dx \sim \frac{h}{3}[&f(x_0) + 4f(x_1) + f(x_2) + \\ &f(x_2) + 4f(x_3) + f(x_4) + ... + \\ &f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)].\end{aligned}$$

$$\begin{aligned}\int_b^a p(x)dx \sim \frac{h}{3}[&f(x_0) + 4f(x_1) + 2f(x_2) + \\ &+4f(x_3) + 2f(x_4) + ... + \\ &+4f(x_{n-1}) + f(x_n)].\end{aligned}$$

We can rewrite this expression in compact form for $n = 2m$ as :

$$\int_b^a p(x)dx \sim \frac{h}{3}[f(x_0) + 4\sum_{j=1}^{m} f(x_{2j-1}) + 2\sum_{j=1}^{m-1} f(x_{2j}) + f(x_{2m})]$$

217. ***Exercise.*** *Implement in Java the rule of Simpson to calculate p-values of the Z distribution. Compare it with the trapezoidal rule. Test whether or not the method devised by Simpson competes with Gnumeric.* *Answer*

By comparing our results with those of Gnumeric, we arrive to the conclusion that the rule of Simpson is enough to produce very accurate results that are calculated in less than the twinkling of an eye. The problem is that if we stop here, we would be like an explorer that finds a grain of rock with incrustations of gold and then returns to his home and give the grain to their children forgiving to look ahead for the deposit from which the grain fall down.

So, let us look at our progress from the following **`creative principle for design`: if something has been done, it can be done much better**. The motivation for this principle is that problems associated to design are in general very complex and therefore found solutions are just very rough approximations to the possible optimum. The rest of this chapter is an exploration that tries to apply this principle.

Now that we have achieved the same degree of precision of Gnumeric, we stand on the same level of precision or maybe or imprecision. For our honor, it would be desirable to run a test to decide that. Our test is the most elementary: the integration of a probability density function along the whole real axis must be one. For the case of the bell, this is equivalent to saying that the integral from zero to infinite must be 0.5. The problem is that we cannot go until infinite. To remedy this trouble, we help ourselves with a **scale transformation** that converts the whole axis into the interval $(-1, 1)$ and we run the integration over that interval. Or, given that we work with a symmetric function, we can integrate from 0 to 1.

A simple and suitable transformation (Wikipedia (2011) on the entry `numeric integration`) is the following:

$x = \frac{t}{1-t^2}$, for $-1 < t < 1$.

This transformation has the following properties:

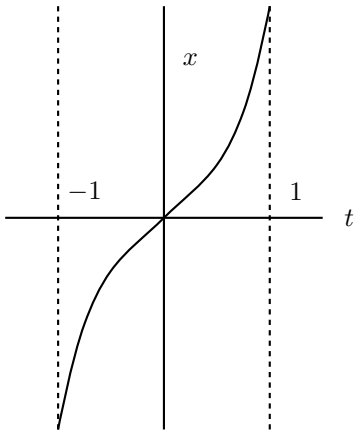


Figure 12.2. If t runs from -1 to 1, x runs from $-\infty$ to ∞.

1. If $t \in (-1, 1)$ then $x \in (-\infty, +\infty)$.
2. It is growing, one to one and onto over $(-1, 1)$
3. If $t = 0$, $x = 0$. If $t \to -1^+$ then $x \to -\infty$. If $t \to 1^-$ then $x \to \infty$.
4. Its inverse is $t = \frac{\sqrt{1+4x^2}-1}{2x}$
5. If $x = \frac{t}{1-t^2}$, $dx = \frac{1+t^2}{(1-t^2)^2}dt$
6. $\int_{-\infty}^{\infty} f(x)dx = \int_{-1}^{1} f(\frac{t}{1-t^2})\frac{1+t^2}{(1-t^2)^2}dt$
7. $\int_{0}^{\infty} f(x)dx = \int_{0}^{1} f(\frac{t}{1-t^2})\frac{1+t^2}{(1-t^2)^2}dt$
8. $\int_{0}^{z} f(x)dx = \int_{0}^{\frac{\sqrt{1+4z^2}-1}{2z}} f(\frac{t}{1-t^2})\frac{1+t^2}{(1-t^2)^2}dt$

218. ***Testing code for the prediction that for the bell*** $\int_0^\infty p(x) = 0.5$**.**

```
//Program E218 SimpTest
//Verifies the prediction that
//the upper half of bell has area 0.5 exactly.
//Method: Simpson

package ejvol5v2p;
```

```
public class SimpTest {

    private static double PI = 3.14159265358979323846;
    private static double z;

    //****************Style: reuse*************
    //Returns the transformed density function of the standardized
    //normal distribution.
    private static double zFunction(double t) {
        z = t / (1 - t * t);
        double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2)
                * (1 + t * t) / ((1 - t * t) * (1 - t * t));
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double z) {
        double f = zFunction(z);
        return f;
    }

    //Returns the integral under the z-density function
    //in within 0 and z. Precision=1/N.
    //Simpson's rule adapted for reuse.
    private static double zHalfBodySimpson(double z, long N) {

        double h = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * h);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * h);
        }
        sum = sum + f((2 * m) * h);
        sum = (h / 3) * sum;
        return sum;
    }

    //The area of half bell is calculated
    //for various degrees of precision.
    private static void work(double z) {
```

```
        System.out.println("N in 10^N  Simpson's rule,"
                + " Area of half bell and error");
        int N;
        double answer;
        for (int i = 1; i < 10; i++) {
            N = (int) Math.pow(10, i);
            answer = zHalfBodySimpson(z, N);
            double error = 0.5 - answer;
            System.out.println(i + "\t" + answer + "\t" + error);
        }
    }

    public static void main(String[] args) {
        double lim = 0.999999999999999;
        work(lim);
    }
}//End of Program E218 SimpTest
```

219. ***Exercise.*** *Run the program and play with the code. Agree else disagree with the appreciation of the Author that claims that the code is precise up to* 1×10^{-14} *which is attained for a division of the interval (0,1) in 1000 subintervals.*

220. ***Challenge.*** *Explain why with finer subdivisions than 1000 the error increases. In general, there are too many mysteries of this type in numerical analysis. One learns to survive with them else one runs into the possibility of getting mad or superexpert.*

221. ***Exercise.*** *We have worked with low to moderate inputs of* z*. For large inputs, we must face the conflict between precision and velocity. Use the change of scale introduced in the last program to calculate the integral under the bell in within zero and any* z*.* *Answer*

12.2 Taylor's theorem

We have observed that approximating a curve by quadratic polynomials produces faster algorithms than approximating it by straight lines. How did that happen? The reason is that we need less subdivisions with 2 degree polynomials than with polynomials with 1-degree. The inference is now immediate: if we increase the degree of approximating polynomials, we eventually can compute the exact integral in just one step with no partition at all apart from the original interval which is the region of integration.

The good news is that that program can be effectively done when we have a infinitely differentiable density function, i.e., one that can be derived indefinitely. The density function of the Gauss Bell is infinitely differentiable all along the real axis and so one can approximate that function by a polynomial without partitions. How to do that was solved by Taylor and its method is knows as Taylor's polynomial.

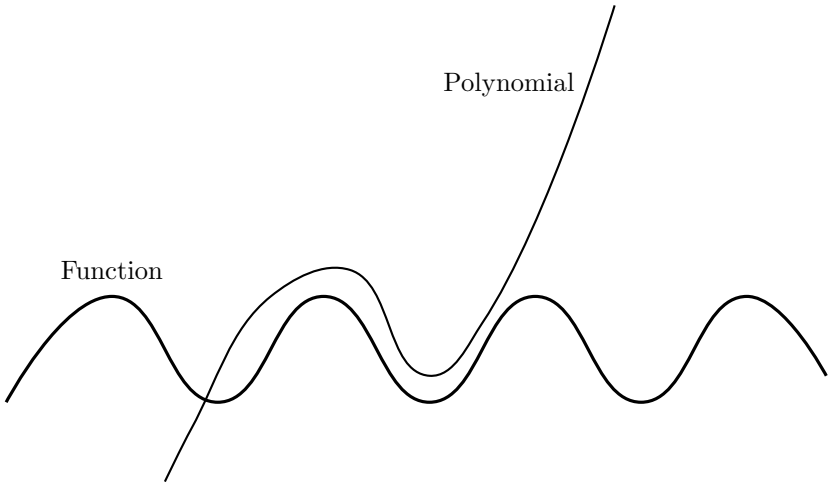


Figure 12.3. A complex function can be approximated very well by polynomials in within certain limits. Taylor found the way to do that in a simple and automatic way. Arisen polynomials are known as Taylor's expansions or Taylor's polynomials, which are calculated for the vicinity of a point that is given beforehand (Frey, [39] 2005)

So, we find an expansion around zero that can be integrated by simple formulas, whose results can be given directly to the computer. This method is expected to produce results in phantoseconds. So, let us implement it. The mathematical part is as follows:

A line obeys the equation $y = b + mx$. If we want the line to approximate a function f near the point with coordinate $(0, f(0))$, we must demand that both the line and the function coincide in that point, i.e., $y(0) = f(0)$, and that the tangent line to both function also coincide. This means that $f'(0) = y'(0) = m$. So

$f(0) = y(0) = b$
$f'(0) = y'(0) = m$

Hence, the searched line has the following equation:

$y = y(0) + y'(0)x.$

If we look now for a polynomial of degree 2, $y = a+bx+cx^2$, that best fit the curve near the point $(0, f(0))$, we must demand three conditions: $y(0) = f(0)$, $y'(0) = f'(0)$ and $y''(0) = f''(0)$. The last condition says that both curves must have the same tangent parable near the point $(0, f(0))$. Now,

$y(0) = f(0)$ implies that $y(0) = a = f(0)$

$y'(0) = f'(0)$ implies that $y'(x) = b + 2cx$ evaluated at zero must be equal to $f'(0)$, In short, $y'(0) = b$

$y''(0) = f''(0)$ implies that $y''(x) = 2c$ evaluated at zero must be equal to $f''(0)$ i.e., $c = f''(0)/2$.

Gluing everything, we get the equation of the polynomial:

$$y = f(0) + y'(0)x + y''(0)x^2/2.$$

Let us try now a third degree polynomial $y = a + bx + cx^2 + dx^3$. We demand $y(0) = f(0)$, $y'(0) = f'(0)$, $y''(0) = f''(0)$ and $y'''(0) = f'''(0)$. The first three conditions render the very same results for coefficients a, b and c. The last condition implies:

$y'(x) = b + 2cx + 3dx^2$

$y''(x) = 2c + 2 \times 3dx$,

$y'''(x) = 2 \times 3d$, which evaluated at zero must coincide with $f'''(0)$; i.e., $d = f'''(0)/(2 \times 3)$.

Gluing everything, we get the equation of the polynomial:

$$y = f(0) + y'(0)x + y''(0)x^2/2 + y'''(0)x^3/(2 \times 3).$$

and so on.

222. ***Important challenge.*** *Our geometric motivation is very primitive because we have no means to estimate the error that we done if we approximate a function by the Taylor's polynomial. So, a must virtuous deduction must exist in which such estimation is done. Please, study the literature to look for it. Hint: probe the web with the expression* `proof of Taylor's theorem`.

223. ***The Taylor expansion of the cumulative function of the standardized normal distribution***

The exponential function $f(x) = e^x$ complies with $(e^x)' = e^x$ and so $(e^x)^{(n)} = e^x$, where the power (n) stands for the $n - th$ derivative.

Henceforth

$f^{(n)}(0) = 1$ since $e^0 = 1$.

The Taylor series for the exponential is, therefore,

$e^s = 1 + s + s^2/2 + s^3/(2 \times 3) + s^4/(2 \times 3 \times 4) + ...$
$e^s = 1 + s + s^2/(2!) + s^3/(3!) + s^4/(4!) + ...$

In short

$e^s = \sum \frac{s^k}{k!}$

where the index of the sum runs from zero to infinite and $0! = 1! = 1$ and $k! = (k-1)!k$. This convention for the sum will be observed everywhere in this section.

Since the exponential series converges for every s, we can replace s by $-s^2/2$ to get:

$$e^{-\frac{s^2}{2}} = \sum \frac{(-\frac{s^2}{2})^k}{k!} = \sum (-1)^k \frac{s^{2k}}{2^k k!}$$

Now, we can integrate from 0 to x:

$$\int_0^x e^{-\frac{s^2}{2}} = \int_0^x \sum (-1)^k \frac{s^{2k}}{2^k k!} = \sum (-1)^k \frac{x^{2k+1}}{2^k k!(2k+1)}$$

where we have taken into account that the last sum evaluated at $x = 0$ is zero and so it does not appear. Now, our move is to use the next approximation for the integral from zero to x under the Gauss bell:

$$\frac{1}{\sqrt{2\pi}} \int_0^x e^{-\frac{s^2}{2}} ds \simeq \frac{1}{\sqrt{2\pi}} \sum_0^M (-1)^k \frac{x^{2k+1}}{2^k k!(2k+1)}$$

The final sub index, M, can be calculated before hand if one uses the Theorem of Taylor for the residue but we prefer to match our accuracy to fit known to us methods.

224. ***Code that uses Taylor's polynomials to integrate under the bell***

```
//Program E224 ZTaylor
//Computes the  integral under the standard Gauss bell
//in within 0 and a positive z.
//Method one: Simpson's rule.
//Method two: Taylor's polynomials.

package ejvol5v2p;

public class ZTaylor {

    private static final double PI = 3.14159265358979323846;
    private static double z;

    //****************Method: Simpson's rule*************
    //Returns the density function of the standardized
    //normal distribution.
    private static double zFunction(double z) {
        double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2);
        return f;
    }
```

```
//To use this method for another distribution,
//instead of  zFunction(z), write and make a call for your
//new distribution.
private static double f(double z) {
    double f = zFunction(z);
    return f;
}

//Returns the integral under the z-density function
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
private static double zHalfBodySimpson(double z, long N) {

    double h = z / (2 * N);
    long m = N;
    double sum = f(0);
    for (int j = 1; j <= m; j++) {
        sum = sum + 4 * f((2 * j - 1) * h);
    }
    for (int j = 1; j < m; j++) {
        sum = sum + 2 * f((2 * j) * h);
    }
    sum = sum + f((2 * m) * h);
    sum = (h / 3) * sum;
    return sum;
}

//*****************Method: Taylor's polynomials**************
//Returns numb!
private static long factorial(int numb) {
    if (numb <= 1) {
        return 1;
    } else {
        return numb * factorial(numb - 1);
    }
}

//Calculates the j-th term of the Taylor's
//expansion around zero of the cumulative function of
//the standardized normal distribution
//evaluated at z
private static double zTerm(double z, int j) {
    double den = (2 * j + 1) * Math.pow(2, j) * factorial(j);
    double zt = Math.pow(-1, j) * Math.pow(z, 2 * j + 1) / den;
    /*System.out.println("j = " + j + " den = "
                    + den + "zt = " + zt);
```

```
         */
        return zt;
    }

    //To use this method for another distribution,
    //instead of  zterm(z,j), write and make a call for your
    //new term.
    private static double jTerm(double z, int j) {
        double t = zTerm(z, j);
        return t;
    }

    //Returns the integral under the z-density function
    //in within 0 and z.
//Taylor's polynomials.M terms included.
    private static double zHalfBodyTaylor(double z, int M) {
        /**
         * With M =12, and for z=1 the algorithm achieves 13 correct ciphers
         */

        double k = 1 / Math.pow(2 * PI, 0.5);
        double sum = 0;
        for (int j = 0; j < M; j++) {
            sum = sum + jTerm(z, j);
        }
        return k * sum;
    }

    //Here we use the trapezoidal rule to
    //estimate the number of terms that must be included
    //in the Taylor's expansion to achieve professional
    //accurateness.
    private static void demo() {
        System.out.println("Area under the standardized "
                + "bell in within zero and 1.");
        System.out.println("N in 10^N, Simpson's rule");
        int N;
        z = 1.0;
        double answer1, answer2;
        for (int i = 3; i < 6; i++) {
            N = (int) Math.pow(10, i);
            answer1 = zHalfBodySimpson(z, N);
            System.out.println(i + "\t  " + answer1);
        }
        System.out.println("Gnumeric " + " 0.34134474606854");
        System.out.println("M  terms, Taylor's polynomials");
```

```
        for (int M = 0; M < 30; M++) {
            answer2 = zHalfBodyTaylor(z, M);
            System.out.println(M + "\t  " + answer2);
        }
    }

    public static void main(String[] args) {
        demo();
    }
}//End of Program E224 ZTaylor
```

225. ***Exercise*** *Play with the code and with Gnumeric and declare whether or not the method of the Taylor's polynomials (plus Java) produces results with professional accurateness in less than the twinkling of an eye.* *Answer*

226. ***Challenge.*** *The Gauss bell is very soft all around and so one can expect from the Taylor's polynomial algorithm to function well everywhere. Nevertheless, direct experimentation shows that the algorithm functions well for values in within zero and two. For values that are greater than two, the algorithm does not function. Besides: when the algorithm functions, the degree of the polynomial must be set to 20 or less, otherwise the result will fluctuate and eventually explode. This phenomenon is very usual in numerical approximation and illustrates the fact that no tool is good for everything. What can we do to extend the scope of our algorithm?*

We see at once two solutions: to divide the real axis in various regions and to calculate appropriate approximations for each one of them separately. This possibility is naturally enabled by the formalism of the Taylor's polynomial. Implement this program.

Another solution is to change the Java Type double by another one with more precision, a choice that is supported by the promise that the series for the cumulative distribution of the bell must converge for every point in the real axis. This option introduces a new to us topic:

12.3 BigDecimals

Because of memory constraints, most programming languages have diverse types of numbers. For normal use, Java offers *int*, *long* and *float*. For usual scientific tasks, we have *int*, *Integer*, *long*, *float* and *double*. The type *Integer* is an object, a heavy duty *int*. Type *float* has a precision of 7 figures to the right of the decimal point while *double* has 15. We have seen that the scope of these types is insufficient for tasks that can be considered normal. Nevertheless, we have good news: Java offers the possibility to work with numbers as large as desired and with decimal expansions of arbitrary precision. In the first case, we have the class *BigInteger* (Oracle, [79] 2011c) and in the second, we have the class *BigDecimal*, which provide us with arbitrary-precision signed decimal numbers (Oracle, [80] 2011d).

227. ***BigDecimals***

One can declare and assign a value to a **BigDecimal** as follows:

```
int scale = 10 //number of significant ciphers.
BigDecimal arctan_5 = arctan(5, scale);
```

The arithmetic operators of both BigDecimals and BigIntegers function according to the point suffix formalism, which is proper of the OOP:

```
//Arithmetic operators:
//To add x: .add(x);
//To multiply by x: .multiply(x)
//To subtract x: .subtract(x)
//To divide by x: .divide(x, scale, roundingMode)
//Operators can be chained and are executed
//from left to right.
//Example
BigDecimal D = arctan_5.add(7);
```

We want to use BigDecimals to improve the scope of our Taylor's algorithm but we have a problem: we are working with an approximation of π expanded up to some 15 decimal places. So, the first task is to calculate π to any arbitrary precision. The following code allows that thanks to BigDecimals.

The code below uses BigDecimals to calculate the decimal expansion of π up to any desired accuracy. We use the Taylor expansion of $arctan(x)$, the inverse function of $tangent(x)$. We also use the Machin's formula for π:
$\pi/4 = 4arctang(1/5) - arctang(1/139)$. Let us explain these topics:

228. ***The series of arctan(x)***

We begin with the next identity:
$(1 + x + x^2 + ... + x^n)(1 - x) = 1 - x^{n+1}$
Therefore and for $x \neq 1$ we have
$(1 + x + x^2 + ... + x^n) = (1 - x^{n+1})/(1 - x)$
If we consider $\mid x \mid < 1$ we are allowed to take the limit when $n \to \infty$ to get
$1/(1 - x) = (1 + x + x^2 + x^3 + ... + x^n + ...)$
We can replace x by $-x^2$ and so
$1/(1 + x^2) = (1 - x^2 + x^4 - x^6 + x^8 + ... + x^n + ...)$
Let us integrate this equality
$\int dx/(1 + x^2) = arctan(x) + k = x - x^3/3 + x^5/5 - x^7/7 + ...$
To evaluate k, we take $x = 0$ and so:
$arctan(0) + k = 0$
since $arctan(0) = 0$, we deduce that $k = 0$. In conclusion:
$arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + ...$
a development that is valid for $\mid x \mid < 1$.

229. ***Machin's formula for*** π ***(Jovanovic, [53] 2005):***

$$\pi/4 = 4arctan(1/5) - arctan(1/139)$$

To prove this equation, our point of departure is the identity

$tan(\alpha + \beta) = \frac{tan\alpha + tan\beta}{1 - tan\alpha tan\beta}$

If we put $\alpha = \beta$, we get

$tan(2\alpha) = \frac{2tan\alpha}{1 - tan^2\alpha}$

Henceforth

$tan(4\alpha) = 4tan\alpha \frac{1 - tan^2\alpha}{1 - 6tan^2\alpha + tan^4\alpha}$

Because

$tan(\alpha - \beta) = \frac{tan\alpha - tan\beta}{1 + tan\alpha tan\beta}$

then

$tan(4\alpha - \beta) = \frac{4tan\alpha(1 - tan^2\alpha) - tan\beta(1 - 6tan^2\alpha + tan^4\alpha)}{(1 - 6tan^2\alpha + tan^4\alpha) + 4tan\alpha tan\beta(1 - tan^2\alpha)}$

Since $tan(\pi/4) = 1$, we can pose $\alpha = arctan(1/5)$, i.e., $tan\alpha = 1/5$, to get $\pi/4 = 4\alpha - \beta = 4/5 - \beta$ and

$tan(\pi/4) = 1 = tan(4/5 - \beta) = \frac{4(1/5)(1 - 1/25) - tan\beta(1 - 6/25 + 1/625)}{(1 - 6/25 + 1/625) + (4/5)tan\beta(1 - 1/25)}$

$1 = tan(4/5 - \beta) = \frac{(4/5)(24/25) - tan\beta(476/625)}{(476/625) + (4/5)tan\beta(24/25)}$

Clearing for $tan\beta$, we get

$(4/5)(24/25) - tan\beta(476/625) = (476/625) + (4/5)tan\beta(24/25)$
$tan\beta = \frac{(4/5)(24/25) - (476/625)}{(4/5)(24/25) + (476/625)} = \frac{(480 - 476)/625}{(480 + 476)/625} = \frac{4}{956} = \frac{1}{239}$
$\beta = arctan(1/239)$

Summary:
$tan(\pi/4) = 1$
$\alpha = arctan(1/5)$,
$\beta = arctan(1/239)$
$\pi/4 = 4\alpha - \beta = 4arctan(1/5) - arctan(1/239)$.
$\pi = 4(4arctan(1/5) - arctan(1/239))$.

We see that to calculate π, it is enough to calculate $arctan(1/5)$ and $arctan(1/239)$, a task that we can solve with the help of the expansion of $arctan(x)$ which is

valid when x is less than 1 (in absolute value). The implementation needs a new structure that we enter to explain.

230. ***The while structure***

We know how to make repetitive tasks using a for structure. When we need to execute a repetitive task but only while certain condition holds, one might use a **while structure** that is defined by the following pseudocode:

```
while (this condition holds)
{
  do this and that;
}
```

231. ***Example.*** *We have the series 1, 1/2, 1/3, 1/4,... We need to know how many terms must be added as a minimum to accumulate a sum of 3.4567. This problem is solved by the next code.*

```
//Program E231 WhileStruct
//Program to learn the while structure

package ejvol5v2p;

public class WhileStruct {

    public static void main(String[] args) {
        double sum = 0;
        long i = 0;
        double one = 1;
        double target = 3.456789;
        while (sum < target) {
            i = i + 1;
            sum = sum + one / i;
            System.out.println(i + "  sum = " + sum);
        }
        System.out.println("To cumulate " + target
                + ", we need the first " + i + " Terms");
    }
}//End of Program E231 WhileStruct
```

232. ***The code to calculate the decimal expansion of π up to any desired accuracy follows. The code is a slight and commented modification of that of Chakraborty ([13] 2007). Pay attention to the definition of arithmetic operators:***

```
/*Program E232 Pi
Pi is calculated to any desired accuracy
thanks to the use of BigDecimals.

Slight and commented modification of the code
presented by Chakraborty (2007)

http://blog.taragana.com/index.php/archive/
calculate-Pi-to-arbitrary-precision-sample-java-code/

Cited 21 Marz 2011.

Original and present codes are free.

BigDecimals are decimal numbers for arbitrary precision
arithmetic.
*/


package ejvol5v2p;


import java.math.BigDecimal;

public class Pi {

    public static void main(String args[])
            throws NumberFormatException {
        //This is a loop that shows successive
        //approximations of Pi.
        //j is the number of decimals
        for (int j = 1; j < 50; j++) {
            Integer digits = j;
            BigDecimal pi = computePi(digits);
            System.out.println(j + "  " + pi);
        }
    }

    /**
     * constants used in Pi computation
     */
    private static final BigDecimal FOUR = BigDecimal.valueOf(4);

    /**
     * rounding mode to use during Pi computation 3.142 is rounded to 3.14 while
```

```
3.14159 is to 3.1416.
     */
    private static final int ROUNDINGMODE = BigDecimal.ROUND_HALF_EVEN;

    /**
     * Compute the value of Pi to the specified number of digits after the
decimal point. The value is computed using Machin's formula:

Pi/4 = 4*arctan(1/5) - arctan(1/239) Pi = (4*arctan(1/5) -
arctan(1/239))*4 and a power series expansion of arctan(x) to sufficient
precision.
     * @param digits
     * @return
     */
    public static BigDecimal computePi(int digits) {
        //Precision of computations is greater than
        //demanded precision. With morePrecision = 10
        //more precision only corrects the last digit.
        int morePrecision = 10;
        int scale = digits + morePrecision;
        //arctan(1/5)
        BigDecimal arctan1_5 = arctan(5, scale);
        //arctan(1/239)
        BigDecimal arctan1_239 = arctan(239, scale);

        //Arithmetic operators:
        //To add x: .add(x);
        //To multiply by x: .multiply(x)
        //To subtract x: .subtract(x)
        //To divide by x: .divide(x)
        //Operators can be chained and are executed
        //from left to right.
        // Pi   = (4*arctan(1/5) - arctan(1/239))*4
        BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
                arctan1_239).multiply(FOUR);
        //only demanded precision is reported
        BigDecimal piAprox = pi.setScale(digits,
                BigDecimal.ROUND_HALF_UP);
        return piAprox;
    }

    /**
     * Compute the value, in radians, of the arctangent of the inverse of th
     * supplied integer to the specified number of digits after the decimal
     * point. The value is computed using the power series expansion for the
     * tangent:
```

```
     *
     * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 + (x^9)/9 ...
     * @param inverseX
     * @param scale
     * @return
     */

    //All results must have a scale of accurateness
    public static BigDecimal arctan(int inverseX,
            int scale) {
        BigDecimal result, numer, term;
        BigDecimal invX = BigDecimal.valueOf(inverseX);
        BigDecimal invX2
                = BigDecimal.valueOf(inverseX * inverseX);

        numer = BigDecimal.ONE.divide(invX,
                scale, ROUNDINGMODE);

        result = numer;
        int i = 1;
        //Loop to compute the series.
        //The loop ends when a term can be rounded to zero
        //given the scale of accurateness.
        do {
            numer
                    = numer.divide(invX2, scale, ROUNDINGMODE);
            int denom = 2 * i + 1;
            term
                    = numer.divide(BigDecimal.valueOf(denom),
                            scale, ROUNDINGMODE);
            //The first term is term number 1.
            //Terms with even i are  subtracted
            //terms with odd i are added
            if ((i % 2) != 0) {
                result = result.subtract(term);
            } else {
                result = result.add(term);
            }
            i++;
        } //Can the term be rounded to zero?
        while (term.compareTo(BigDecimal.ZERO) != 0);
        return result;
    }
}//End of Program E232 Pi
```

233. ***Exercise*** *Play with the code and try to understand it.*

We have a problem: to improve our Taylor's polynomials algorithm to calculate the cumulative function of the bell, we need the square root of π, but the *BigDecimal* class lacks that function. So, let us implement it.

12.4 Big square roots

Our method to find square roots is based on the following fact: if $\sqrt{t} = x$ then $x^2 = t$, i.e., $x^2 - t = 0$. Therefore, to find the root of t we must find a zero of the error function $f(x) = x^2 - t$ which is defined for every real number x. So, we will done if we get a method to find a zero of a differentiable function.

One can use the Newton's method, whose main idea can be understood as follows:

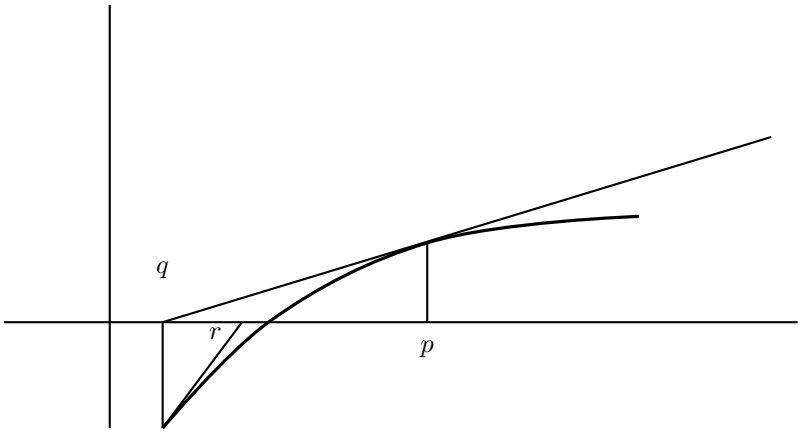


Figure 12.4. To find a zero of a differentiable function, one begins with a preferentially good approximation, p, next one traces the tangent to the point in the curve $(p, f(p))$. The intersection of that tangent with the horizontal axis provides us with q a better approximation. If we repeat the procedure with q, we get r and so on until the desired accuracy is achieved. See Burden et al ([12] l.c., 1985).

If we calculate the slop of the tangent line at $(p, f(p))$, we get
$f'(p) = \frac{f(p)-0}{p-q}$
When p is given and q is needed, we have
$f(p) = pf'(p) - f'(p)q$
$q = \frac{f'(p)p-f(p)}{f'(p)} = p - \frac{f(p)}{f'(p)}$.

From here we get our algorithm for finding a zero of a differentiable function $f(x)$, when a close initial approximation g is given:

$g_o = g$
$g_{i+1} = g_i - \frac{f(g_i)}{f'(g_i)}$

To apply this method to find square roots, we pose $f(x) = x^2 - t$. Since $f'(x) = 2x$, our procedure reads:

Input = n, a real number.
Output: a good approximation to $\sqrt{n}$.
Algorithm:
g_o = initial seed, given by hand.
$g_{i+1} = g_i - \frac{f(g_i)}{f'(g_i)} = g_i - \frac{(g_i)^2 - n}{2g_i} = g_i - \frac{g_i}{2} + \frac{n}{2g_i} = \frac{g_i}{2} + \frac{n}{2g_i} = \frac{g_i + n/g_i}{2}$.

To implement the algorithm with *BigDecimals*, we adapt the algorithm and code by Gilleland (2011) as follows:

1. Get input n as type *double*. Check it must be positive.
2. To find the first approximation g, compute the square root of n using the Java procedure $Math.sqrt(n)$ which is valid for type *double*.
3. Convert n and g to *BigDecimal*.
4. Compute result = $((n/g) + g)/2$. Let g be the result just computed.
5. Repeat step 4 until the last two results obtained are the same (the difference can be approximated to zero given the chosen scale).

In the code below, we introduce two new Java types, StringBuffer and BigInteger.

234. *StringBuffers*

To manage tests, we have used Strings, whose use is simple and direct. Nevertheless, for heavy duty, Java offers us **StringBuffers**, which in particular can be as long as desired. The next piece of code illustrates how they function:

```
//Declaration and instantiation
StringBuffer sb = new StringBuffer ();//sb has no chars
//Random generator
java.util.Random r = new java.util.Random ();
//Ten random digits are appended to the end of
//StringBuffer sb
for (int i = 0; i < nDigits; i++)
   sb.append (r.nextInt (10));
//To know more, type
sb.
//and wait for a document to appear.
```

235. ***BigIntegers***

The initialization of a **BigInteger** can be done as follows:

```
//Initialization from a String
String s = "1234567891234567890";
BigInteger n = new BigInteger(s);
//Initialization from a StringBuffer
StringBuffer sb = new StringBuffer ();
    java.util.Random r = new java.util.Random ();
    for (int i = 0; i < nDigits; i++)
      sb.append (r.nextInt (10));
BigInteger n = new  BigInteger (sb.toString ());

//To know more: type
n.
//and wait.
```

In the next program, we use a new trick:

236. ***Boolean flags.*** *A boolean variable can take on either of two values, false else true. A* ***boolean flag*** *is a boolean variable that is used as indicator along a program.*

237. ***Example.*** *Let us see how a boolean flag is used in a program:*

```
//Accumulating until 10
int i = 3;
int sum = 0;
boolean more = true;
while (more)
 {
  sum = sum + i;
  if (sum > 10) more = false;
  else i = i+1;
 }
System.out.println("i = " + i );
```

That piece of code is equivalent to the next one:

```
//Accumulating until 10
int i = 3;
int sum = 0;
while (sum <= 10)
 {
   sum = sum + i;
   i = i+1;
```

```
 }
int j = i-1;
System.out.println("i = " + j );
```

238. ***Huge variability.*** *We see that any slightly complex task can be fulfilled by various different programs. Since a complex program is, in general, composed of small pieces of code, any complex task can be implemented by a huge number of possible programs. This is important for us because we keep an eye on biology: does natural evolution exhibit the expected huge variability in programs?*

239. ***The code that finds the square root of a number follows. Arbitrary large or small inputs plus arbitrary precision are enabled thanks to BigIntegers and BigDecimals.***

```
/*Program 239 BigSquareRoot
 * The square root of a number is found.
 * Arbitrary large or small inputs plus arbitrary precision
 * are enabled by using BigInteger and BigDecimals.
 *
 * The code is a slight adaptation of that presented by
 * Michael Gilleland
 * http://www.merriampark.com/bigsqrt.htm
 * Cited{21 Marz 2011}.
 * Original and present codes are free.
 * */

package ejvol5v2p;

import java.math.*;

public class BigSquareRoot {

    private static final BigDecimal ZERO = new BigDecimal("0");
    private static final BigDecimal ONE = new BigDecimal("1");
    private static final BigDecimal TWO = new BigDecimal("2");
    public static final int DEFAULT_MAX_ITERATIONS = 50;
    //Precision scale: number of digits after the decimal point
    public static final int DEFAULT_SCALE = 30;

    private static BigDecimal error;
    private static int iterations;
    private static boolean traceFlag;
    private static int scale = DEFAULT_SCALE;
    private static int maxIterations = DEFAULT_MAX_ITERATIONS;

    //----------------------------------------
```

```
    // The error is the original number minus
    // (sqrt * sqrt). If the original number
    // was a perfect square, the error is 0.
    //---------------------------------------
    public BigDecimal getError() {
        return error;
    }

    //-----------------------
    // Number of iterations
    //-----------------------
    public int getIterations() {
        return iterations;
    }

    //-----------
    // Trace flag
    //-----------
    public boolean getTraceFlag() {
        return traceFlag;
    }

    public void setTraceFlag(boolean flag) {
        traceFlag = flag;
    }

    //------
    // Scale
    //------
    public int getScale() {
        return scale;
    }

    public void setScale(int scale) {
        BigSquareRoot.scale = scale;
    }

    //-------------------
    // Maximum iterations
    //-------------------
    public int getMaxIterations() {
        return maxIterations;
    }

    public void setMaxIterations(int maxIterations) {
        BigSquareRoot.maxIterations = maxIterations;
```

```
    }

    //--------------------------
    // Get initial approximation:
    // the square root as double
    //--------------------------
    private static BigDecimal
            getInitialApproximation(double n) {
        double s = Math.sqrt(n);
        BigDecimal g = BigDecimal.valueOf(s);
        return g;
    }

    //----------------
    // Get square root
    //----------------
    private static BigDecimal sqroot(BigDecimal n) {

        // Make sure n is a positive number
        if (n.compareTo(ZERO) <= 0) {
            throw new IllegalArgumentException();
        }
        //n is transformed to type double
        double s = n.doubleValue();
        BigDecimal initialGuess;
        if (s > 0) {
            initialGuess = getInitialApproximation(s);
        } else {
            initialGuess = ONE;
        }
        trace("Initial guess " + initialGuess.toString());
        BigDecimal lastGuess;
        BigDecimal guess = new BigDecimal(
                initialGuess.toString());

        // This is the algorithm:
        iterations = 0;
        boolean more = true;
        while (more) {
            lastGuess = guess;
            /* n/g */
            guess = n.divide(guess, scale,
                    BigDecimal.ROUND_HALF_UP);
            /* n/g + g*/
            guess = guess.add(lastGuess);
            /* (n/g + g)/2 */
```

```
            guess = guess.divide(TWO, scale,
                    BigDecimal.ROUND_HALF_UP);
            trace("Next guess " + guess.toString());
            error = n.subtract(guess.multiply(guess));
            if (++iterations >= maxIterations) {
                more = false;
            } else if (lastGuess.equals(guess)) {
                more = error.abs().compareTo(ONE) >= 0;
            }
        }
        return guess;
    }

    //------
    // Trace
    //------
    private static void trace(String s) {
        if (traceFlag) {
            System.out.println(s);
        }
    }

    //----------------------
    // Get random BigInteger
    //----------------------
    public static BigInteger getRandomBigInteger(int nDigits) {
        StringBuilder sb = new StringBuilder();
        java.util.Random r = new java.util.Random();
        for (int i = 0; i < nDigits; i++) {
            sb.append(r.nextInt(10));
        }
        return new BigInteger(sb.toString());
    }

    //-----
    // Demo
    //-----
    public static void demo() {
        BigInteger n;
        BigDecimal sqrt;
        BigSquareRoot app = new BigSquareRoot();
        app.setTraceFlag(true);

        // Generate a random big integer with a hundred digits
        n = BigSquareRoot.getRandomBigInteger(100);
```

```
        // Build an array of test numbers
        String testNums[]
                = {"9", "30", "720", "1024", n.toString()};

        for (int i = 0; i < testNums.length; i++) {
            n = new BigInteger(testNums[i]);
            if (i > 0) {
                System.out.println("----------------------------");
            }
            System.out.println("Computing the square root of");
            System.out.println(n.toString());
            int length = n.toString().length();
            if (length > 20) {
                app.setScale(length / 2);
            }
            BigDecimal m = new BigDecimal(n);
            sqrt = sqroot(m);
            System.out.println("Iterations = "
                    + app.getIterations());
            System.out.println("Sqrt = " + sqrt.toString());
            System.out.println("Sqrt*Sqrt = "
                    + sqrt.multiply(sqrt).toString());
            System.out.println(n.toString());
            System.out.println("Error = "
                    + app.getError().toString());
        }

    }

    public static void main(String[] args) {
        demo();
    }
}//End of Program 239 BigSquareRoot
```

240. ***Exercise*** *Play with the code and try to understand it.*

Let us use now this know-how to see whether or not we can get rid of the found problems with the algorithm that is based on the Taylor's polynomial for the calculation of the cumulative function of the z-distribution.

12.5 BigDecimals and the z-distribution

Let us use *BigDecimals* and *BigIntegers* to rewrite a previous code that was based on Taylor's theorem to calculate the cumulative function of the standardized normal distribution.

241. ***This is a code to calculate the integral in within zero and z under the standardized normal distribution. BigDecimals are used.***

```
//Program E241 ZTaylorBD
//Computes the  integral under the standard Gauss bell
//in within 0 and a positive z.
//Method one: Simpson's rule.
//Method two: Taylor's polynomials using BigBecimals.
//Pi is calculated to any desired accuracy:
//Slight and commented modification of the code
//presented by Chakraborty (2007)

/*
http://blog.taragana.com/index.php/archive/
calculate-pi-to-arbitrary-precision-sample-java-code/
*/

//Cited 15 Ap 2010.
//Original and present codes are free.

package ejvol5v2p;


import java.math.BigDecimal;

public class ZTaylorBD {

    /**
     * constants
     */
    private static final BigDecimal ZERO = BigDecimal.valueOf(0);
    private static final BigDecimal ONE = BigDecimal.valueOf(1);
    private static final BigDecimal TWO = BigDecimal.valueOf(2);
    private static final BigDecimal FOUR = BigDecimal.valueOf(4);

    //Precision scale: number of digits after
    //the decimal point
    private static final int SCALE = 30;

    private static BigDecimal error;
    private static int iterations;
    private static boolean traceFlag;

    private static final int MAXNITERAIONS = 50;
    private static final double PI = 3.14159265358979323846;
```

```
    /**
     * rounding mode to use during pi computation 3.142 is rounded to 3.14 while
     * 3.14159 is to 3.1416.
     */
    private static final int ROUNDINGMODE = BigDecimal.ROUND_HALF_EVEN;

    /**
     * ************************************
     * Pi is calculated
 * ************************************
     */
    /**
     * Compute the value of pi to the specified number of digits after the
     * decimal point. The value is computed using Machin's formula:
     *
     * pi/4 = 4*arctan(1/5) - arctan(1/239) pi = (4*arctan(1/5) -
     * arctan(1/239))*4 and a power series expansion of arctan(x) to sufficient
     * precision.
     * @param digits
     * @return
     */
    public static BigDecimal computePi(int digits) {
        //Precision of computations is greater than
        //demanded precision. With morePrecision = 10
        //corrections are done to the last digit only.

        //arctan(1/5)
        BigDecimal arctan1_5 = arctan(5, SCALE);
        //arctan(1/239)
        BigDecimal arctan1_239 = arctan(239, SCALE);

        //Arithmetic operators:
        //To add x: .add(x);
        //To multiply by x: .multiply(x)
        //To substract x: .substract(x)
        //To divide by x: .divide(x)
        //Operator can be chained and are executed
        //from left to right.
        // pi   = (4*arctan(1/5) - arctan(1/239))*4
        BigDecimal piAt = arctan1_5.multiply(FOUR).subtract(
                arctan1_239).multiply(FOUR);
        //only demanded precision is reported
        return piAt.setScale(digits,
                BigDecimal.ROUND_HALF_UP);
    }
```

```
/**
 * Compute the value, in radians, of the arctangent of the inverse of th
 * supplied integer to the specified number of digits after the decimal
 * point. The value is computed using the power series expansion for the
 * tangent:
 *
 * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 + (x^9)/9 ...
 * @param inverseX
 * @param scale
 * @return
 */

//All results must have a scale of accurateness
public static BigDecimal arctan(int inverseX,
        int scale) {
    BigDecimal result, numer, term;
    BigDecimal invX = BigDecimal.valueOf(inverseX);
    BigDecimal invX2
            = BigDecimal.valueOf(inverseX * inverseX);

    numer = BigDecimal.ONE.divide(invX,
            SCALE, ROUNDINGMODE);

    result = numer;
    int i = 1;
    //Loop to compute the series.
    //The loop ends when a term can be rounded to zero
    //given the scale of accurateness.
    do {
        numer
                = numer.divide(invX2, SCALE, ROUNDINGMODE);
        int denom = 2 * i + 1;
        term
                = numer.divide(BigDecimal.valueOf(denom),
                        SCALE, ROUNDINGMODE);
        //The first term is term number 1.
        //Terms with even i are  subtracted
        //terms with odd i are added
        if ((i % 2) != 0) {
            result = result.subtract(term);
        } else {
            result = result.add(term);
        }
        i++;
    } //Can the term be rounded to zero?
    while (term.compareTo(BigDecimal.ZERO) != 0);
```

```
        return result;
    }

//********Square root of a BigDecimal number************
    //--------------------------
    // Get initial approximation:
    // the square root as double
    //--------------------------
    private static BigDecimal
            getInitialApproximation(double n) {
        double s = Math.sqrt(n);
        BigDecimal g = BigDecimal.valueOf(s);
        return g;
    }

    //------
    // Trace
    //------
    private static void trace(String s) {
        if (traceFlag) {
            System.out.println(s);
        }
    }
    //----------------
    // Get square root
    //----------------

    private static BigDecimal sqroot(BigDecimal n) {

        // Make sure n is a positive number
        if (n.compareTo(ZERO) <= 0) {
            throw new IllegalArgumentException();
        }
        //n is transformed to type double
        double s = n.doubleValue();
        BigDecimal initialGuess;
        if (s > 0) {
            initialGuess = getInitialApproximation(s);
        } else {
            initialGuess = ONE;
        }
        trace("Initial guess " + initialGuess.toString());
        BigDecimal lastGuess;
        BigDecimal guess = new BigDecimal(initialGuess.toString());

        // This is the algorithm:
```

```
        iterations = 0;
        boolean more = true;
        while (more) {
            lastGuess = guess;
            /* n/g */
            guess = n.divide(guess, SCALE,
                    BigDecimal.ROUND_HALF_UP);
            /* n/g + g*/
            guess = guess.add(lastGuess);
            /* (n/g + g)/2 */
            guess = guess.divide(TWO, SCALE,
                    BigDecimal.ROUND_HALF_UP);
            trace("Next guess " + guess.toString());
            error = n.subtract(guess.multiply(guess));
            if (++iterations >= MAXNITERAIONS) {
                more = false;
            } else if (lastGuess.equals(guess)) {
                more = error.abs().compareTo(ONE) >= 0;
            }
        }
        return guess;

    }

//*********The cumulative function of the***************
//*********standardized normal distribution**************
//********is calculated.********************************
    //****************Method: Simpson's rule*************

    //Returns the density function of the standardized
    //normal distribution.

    private static double zFunction(double z) {
        double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2);
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double z) {
        double f = zFunction(z);
        return f;
    }

    //Returns the integral under the z-density function
```

```
    //in within 0 and z. Precision=1/N.
    //Simpson's rule adapted for reuse.
    private static double zHalfBodySimpson(double z, long N) {

        double h = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * h);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * h);
        }
        sum = sum + f((2 * m) * h);
        sum = (h / 3) * sum;
        return sum;
    }

    //*****************Method 2: Taylor's polynomials********
    //*****************BigDecimals are used*****************
    //Returns numb!
    private static BigDecimal factorial(BigDecimal numb) {
        //output of compare:
        //+1 if numb is greater than ONE.
        //0 if they are equal
        //-1 if numb is less than ONE
        if (numb.compareTo(ONE) < 1) {
            return ONE;
        } else {
            return factorial(numb.subtract(ONE)).
                    multiply(numb);
        }
    }

    //Returns x to power n
    private static BigDecimal Power(double x, int n) {
        BigDecimal xBD = BigDecimal.valueOf(x);
        if (n == 0) {
            return ONE;
        } else {
            return xBD.multiply(Power(x, n - 1));
        }
    }

    //Calculates the j-th term of the Taylor's
    //expansion around zero of the cumulative function of
```

```
    //the standardized normal distribution
    //evaluated at z
    private static BigDecimal zTerm(double z, int j) {

        BigDecimal num = Power(z, 2 * j + 1).multiply(Power(-1, j));

        BigDecimal jBD = BigDecimal.valueOf(j);

        // den = (2*j+1)*Math.pow(2,j);
        BigDecimal den = Power(2, j).
                multiply((jBD.multiply(TWO)).add(ONE));
        BigDecimal denoBD = factorial(jBD).multiply(den);
        BigDecimal zt = num.divide(denoBD, SCALE, ROUNDINGMODE);
        /*System.out.println("j = " + j + " den = "
                  + denoBD + "zt = " + zt);*/

        return zt.setScale(SCALE,
                BigDecimal.ROUND_HALF_UP);
    }

    //To use this method for another distribution,
    //instead of  zterm(z,j), write and make a call for your
    //new term.
    private static BigDecimal jTerm(double z, int j) {
        BigDecimal t = zTerm(z, j);
        return t;
    }

//Returns the integral under the z-density function
//in within 0 and z.
//Taylor's polynomials. M terms included.
    private static BigDecimal zHalfBodyTaylor(double z, int M) {
        /**
         * With M =12, and for z=1 the algorithm achieves 13 correct ciphers
         */

        BigDecimal pi2 = computePi(SCALE);
        //System.out.println("pi = " + pi);
        BigDecimal x = sqroot(pi2.multiply(TWO));
        //System.out.println("sqrt(2*pi) = " + x);
        BigDecimal k = ONE.divide(x, SCALE, ROUNDINGMODE);
        //System.out.println("k = " + k);
        BigDecimal sum = ZERO;
        for (int j = 0; j < M; j++) {
            sum = sum.add(jTerm(z, j));
        }
```

```
        sum = sum.multiply(k);
        //Reported precision
        int print_scale = 15;
        return sum.setScale(print_scale,
                BigDecimal.ROUND_HALF_UP);
    }

    //Here we use the trapezoidal rule to
    //estimate the number of terms that must be included
    //in the Taylor's expansion to achieve professional
    //accurateness.
    private static void demo() {
        double z = 1;
        System.out.println("Area under the standardized "
                + "bell in within zero and z = " + z);
        System.out.println("N in 10^N, Simpson's rule");
        int N;

        double answer1;
        for (int i = 2; i < 6; i++) {
            N = (int) Math.pow(10, i);
            answer1 = zHalfBodySimpson(z, N);
            System.out.println(i + "\t  " + answer1);
        }
        System.out.println("For z= 1, Gnumeric gives "
                + " 0.34134474606854");

        //Number of terms to be included in the Taylor's poly
        int Nterms = 20;
        System.out.println("Taylor's polynomial with "
                + Nterms + " terms");
        System.out.println("Number of digits after the decimal"
                + " point = " + SCALE);
        BigDecimal answer2 = ZERO;
        for (int M = 0; M < Nterms; M++) {
            answer2 = zHalfBodyTaylor(z, M);
            // System.out.println("M = " + M+ "\t" + answer2);
        }
        System.out.println("HalfBody = " + answer2);
    }

    public static void main(String[] args) {
        demo();
    }
}//End of Program E241 ZTaylorBD
```

242. ***Exercise.*** *Play with the code and try to understand it. Work enough to accept else refute the following conclusion drawn by the Author: one can adjust the precision in the scale of BigDecimal together with the number of terms in the Taylor expansion to achieve good results for z in within zero and 9. For $z > 9$, it would be better to return 0.5. While results coincide or even improve those of Gnumeric, we can say that we got professional accurateness but a price in delaying was paid and so we were unable to produce results in less than a twinkling of an eye, as Gnumeric or the Simpson's rule do.*

12.6 Deformations

Were our purpose to make a commercial project, we already would be done: we have explored various methods to solve a problem and we can say that the Simpson's rule plus a change of scale is the best and is enough for the trade in regard with the modern standards of both velocity and accurateness. Nevertheless, our path leads further: we are interesting in the relation between evolution and complexity, where ***evolution*** *stands for tinkering about the form as diverse elements can be combined by a person or a process to achieve a predefined function in an optimized way.* We see that a complex problem can be solved in really many ways, and that improvements are rather difficult. To enrich our experience, let us try out the possibilities of the following idea:

A cumulative function is an expression of the form

$$\Phi(z) = \int_{-\infty}^{z} f(x)dx$$

where $f(x)$ is a probability density function.

All cumulative functions have the same form, beginning from 0 at $-\infty$, advancing in not decreasing form and ending in 1 at ∞. There are too many functions that are easily to calculate and that resemble a cumulative function. So, let us explore the following idea to calculate a cumulative function: we choose an easy to calculate function and deform it to get a good approximation to the cumulative one.

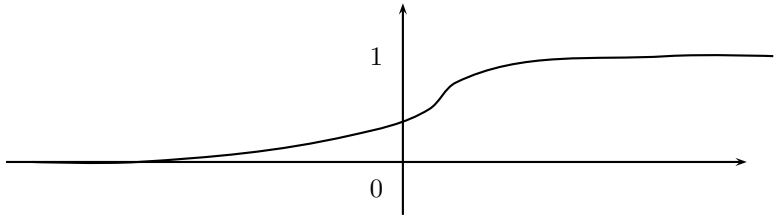


Figure 12.5. The general form of a cumulative function.

In operational terms, we realize this idea in two steps, which will characterize our vision of evolution: firstly, we design an evolutionary environment and, secondly, we run evolution, blind evolution, in the devised environment. The adjective blind arises from the sage "evolution is like a blind watchmaker'... " (Dawkins, [18] 1986). First step in this case: a family of easy to calculate functions and that resemble a cumulative function is chosen. Second: from the selected family we pick up that function that best approximate our target or maybe a subset of those functions. In this regard, a simple concept will enhance clarity and generality:

243. ***Combinatorial bases***

We can understand that concept if we pay attention to the general form of a Taylor's polynomial up to certain degree:

$$p(x) = a_0 + a_1x + a_2x^2 + ...a_nx^n$$

So, a Taylor's polynomial is no more than a special combination of monomials x^n. This combination is dressed in the form of linear combinations, multiplication by real numbers and addition of those terms. Taylor's polynomials have allowed us to use certain rules to combine a certain set, the combinatorial basis, to achieve a given goal. So, a ***combinatorial basis*** *is a set or family of things to tinker with to fill in a predefined function as optimal as possible.* Let us return now to our problem.

244. ***Engineering a combinatorial basis***

Our purpose is to calculate

$$\int_0^z \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx$$

We had the hope that BigDecimals plus Taylor's polynomials would had helped us to produce accurate results in picoseconds by giving us an easy to calculate function that works over the entire range of real numbers. But we have failed. In other words, the basis composed of monomials is not adequate for our purpose. Why? Which could be the reason of our failure? The most obvious explanation is that a Taylor's expansion uses a combinatorial basis, whose members are of the form x^n, which tend to infinite for large values of x, while cumulative functions approach 1 as z grows. Of course, the fitting of our target by polynomials must be very expensive for large values of z, where $z = 5$ is already a very large number. Let us try to engineer a better combinatorial basis.

Our cumulative distribution function is officially

$$\Phi(z) = \int_{-\infty}^z \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx$$

Let us consider a slight modification:
$d(z) = \Phi(z) - 1/2$.
This new function has the following properties:

1. When z diverges to $-\infty$, the function approaches -1/2.
2. When z diverges to ∞, the function approaches $+1/2$.
3. $d(0) = 0$.
4. w is an odd function, that is, it is symmetric with respect to the origin: $d(-z) = -d(z)$.

The function $d(z)$ is called d-cumulative function (d comes from down).

We can approximate every d-cumulative function by either one of the next three functions, which also share those aforementioned characteristics:

$s_(z) = \frac{e^z}{1+e^z} - \frac{1}{2}$

$a(z) = \frac{1}{\pi}(arctan(z))$

$h(x) = \frac{x}{2\sqrt[4]{x^4+1}}$

These functions are indeed representatives of the next families:

$s_k(z) = \frac{e^{kz}}{1+e^{kz}} - \frac{1}{2}$

$a_k(z) = \frac{1}{4\pi}(arctan(kz) + \pi/2)$

$h_k(x) = \frac{kx}{2\sqrt[4]{(kx)^4+1}}$

245. ***Exercise*** *Use your preferred spreadsheet to become acquainted with these families. Take* $e = 2.71828182845905$. *For Gnumeric, Excel or LibreOffice, you must tabulate each function and then call a drawing procedure in scattering or XY mode. To find a fourth root use the function power to the 0.25. Verify else refute the next generalizations made by the Author:*

a) When k grows, the graphic of s_k , a_k, *and* $h_{m,k}$ *become steeper.*

b) All functions are very sensible to a change in k in within zero and one. So, these families enjoy a high degree of plasticity.

246. ***Exercise*** *Use an informal method to find a suitable value for k in each family to approximate the d-cumulative function of the standardized normal distribution. For* s_k *and* a_k *try a value around* $k = 1.5$. *For* h_k *explore around* $k = 0.7$ *and* $m = 4$. *Pick up the more promising family. The Author chose h.*

247. ***Comment***

Aforementioned families can be generalized using an arbitrary function $\tau(z)$ that must be odd, non-decreasing, growing from $(-\infty, -\infty)$ to (∞, ∞) , say, $\tau(z) = kz^3$ with $k > 0$, or one can also take a sum of odd monomials. They take on the following form:

$s_\tau(z) = \frac{e^{\tau(z)}}{1+e^{\tau(z)}} - \frac{1}{2}$

$a_\tau(z) = \frac{1}{4\pi}(arctan(\tau(z)) + \pi/2)$

$h_\tau(z) = \frac{\tau(z)}{2\sqrt[4]{\tau^4(z)+1}}$

The h family also is generalized in

$h_\tau(z) = \frac{\tau(z)}{2\sqrt[4]{((\tau(z))^4+1)}}$

248. ***Our metric.*** *To quantify the error of approximation of f by g we measure their distance as follows:*

$d(f, g) =|| f - g ||= Sup \mid f(x) - g(x) \mid$

where Sup, roughly, takes the maximum value of $\mid f(x)-g(x) \mid$ when x ranges over the points where the functions have been defined. Our measure captures the common sense appreciation of error.

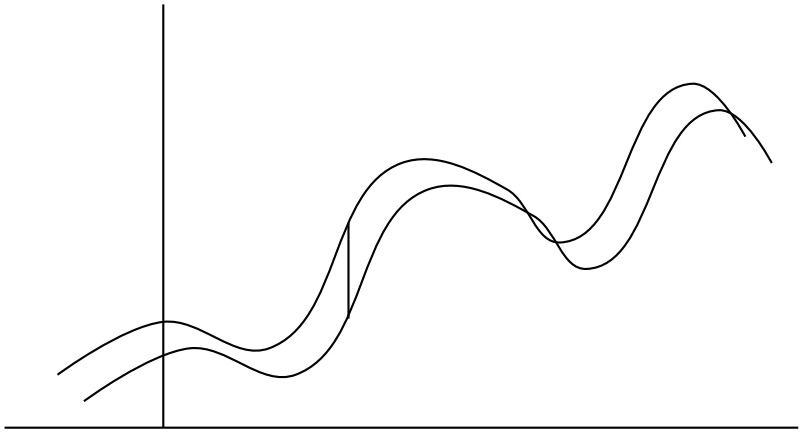


Figure 12.6. We measure the distance between two functions by taking the maximal vertical distance between the two functions. We will use that metric to quantify the error of approximation.

249. ***Exercise*** *In your preferred spreadsheet, use the metric defined in the previous graphic to find the approximation error of the d-cumulative function by*

the best exponent of each family. The values found by the Author are: for a the error was around 0.25, for s around 0.025 and for h less than 0.01.

Let us compose now the Java code to check that our informal work is really fair.

250. ***The next is a code that computes the cumulative function of the standardized normal distribution and that compares it with an ad hoc algebraic approximation given by***

$$h(x) = \frac{0.7x}{2\sqrt[4]{(0.7x)^4+1}}$$

```
//Program E250 HFamily
//Computes the  integral under the standard Gauss bell
//in within 0 and a positive z.
//Method 1: trapezoidal rule.
//Method 2: algebraic approximation given by
//h(z) = 0.5 + (07z)/(2 fourthRoot((0.7z)^4 + 1));

package ejvol5v2p;

public class HFamily {

    private static final double PI = 3.14159265358979323846;
    //Values of the true cumulative function
    private static final double ZVECT[] = new double[10000000];

    // Mandatory initialization
    private static void initialize(long N) {
        for (int i = 0; i <= N; i++) {
            ZVECT[i] = 0.5;
        }
    }

    //*********Method 1: Trapezoidal rule*****************

    //Returns the density function of the standardized
    //normal distribution.
    private static double zFunction(double z) {
        double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2);
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
```

```
    //new distribution.
    private static double f(double z) {
        double f = zFunction(z);
        return f;
    }

    //Returns 0.5 + the integral under the z-density function
    //in within 0 and z. Precision=1/N.
    //Trapezoidal rule adapted for reuse.
    private static void zHalfBodyTrapReuse(double zMax,
            long N) {
        /**
         * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciphers.
         */
        double Nr = N;
        double h = zMax / Nr;
        //System.out.println("h="+h);
        double sum = 0;
        for (int j = 1; j <= N; j++) {
            sum = sum + (h / 2) * (f(h * (j + 1)) + f(h * j));
            ZVECT[j] = sum + 0.5;
        }
    }

    //************** Method 2: algebraic fitting**********
    //Returns an ad hoc algebraic approximation
    //to the cumulative function of the
    //standardized normal distribution.

    private static double hFunction(double z) {
        double hz = 0.5 + (0.7 * z)
                / (2 * Math.pow(Math.pow((0.7 * z), 4) + 1, 0.25));
        return hz;
    }

    //To use this method for another approximation,
    //instead of  hFunction(z), write and make a call for your
    //new proposal.
    private static double fitting(double z) {
        double f = hFunction(z);
        return f;
    }

    //**************Comparison of two methods**************
    private static void title(int N) {
        System.out.println("cum = 0.5 + Area under "
```

```
                + " the standardized bell in within zero and z.");
        System.out.println("That area is calculated by "
                + "the Trapezoidal rule");
        System.out.println("with " + N + " divisions "
                + "in within 0 and 10.");
        System.out.println("hz is our "
                + "algebraic approximation + 0.5.");
        System.out.println("Error = cum - hz.");
    }

    private static void comparison() {
        //Calculations are done for z in (0, zMax)
        double zMax = 10;
        //N= Number of subdivisions of the interval (0,zMax).
        int k = 3;
        int N = (int) Math.pow(10, k);

        initialize(N);
        title(N);
        //Trapezoidal rule
        zHalfBodyTrapReuse(zMax, N);
        double hz = 0;
        System.out.println("z from 0 to 1");
        for (int i = 0; i < 11; i++) {
            int j = i * N / 100;
            double z = i;
            z = z / 10;
            System.out.print("z=" + z + "  cum = " + ZVECT[j]);
            //Our algebraic fitting
            hz = fitting(z);
            System.out.print(" hz= " + hz);
            double error = ZVECT[j] - hz;
            System.out.println(" Error = " + error);
        }
        System.out.println("z from 1 to 10");
        for (int i = 1; i < 11; i++) {
            int j = i * N / 10;
            double z = i;
            System.out.print("z = " + i + "  cum = " + ZVECT[j]);
            hz = fitting(z);
            System.out.print(" hz= " + hz);
            double error = ZVECT[j] - hz;
            System.out.println(" Error = " + error);
        }
    }
```

```
    public static void main(String[] args) {
        comparison();
    }
}//End of Program E250 HFamily
```

251. ***Exercise*** *Run the program and play with the code. Sketch a graphic of the error function and make an estimation of the maximal error.* *Answer*

252. ***Exercise*** *Modify the previous program to exactly compute the maximum error of approximation of our $h(z)$ function with respect to the true value given by the trapezoidal rule. Make sure that results agree with those of your preferred spreadsheet.* *Answer*

253. ***Challenge.*** *Our programs uses a vector to keep the values of the cumulative function once these have been calculated by the trapezoidal rule. Rewrite the program in such a way that no memory is necessary and so anyone can augment accurateness without worrying about memory restrictions.*

We have seen that the h-family provides a not bad approximation to the cumulative function of the z-distribution. Let us find now the best fitting that is furnished by that family. To solve that task, we will resort to evolution.

12.7 Conclusion

We have defined evolution as a process of tinkering about the form as diverse elements can be combined by a person or a process to achieve a predefined function in an optimized way. In general, tinkering happens over a family of objects or things, which conform a combinatorial basis. This abstraction has been achieved by considering Taylor's polynomials, functions that allowed us to approximate the cumulative distribution function of the normal distribution. With this abstraction, most optimizing activities by human beings or nature result to be framed at evolution. We have witnessed that with a bit of inspiration and a lot of patience one can get interesting results. Biological evolution is an automatic mechanism that implements that abstract concept.

Chapter 13

Evolution on letter-zymes

```
Letter-zymes:  strings that are at the same time phenotypes
and genotypes.
```

254. ***Purpose: In this chapter we simulate evolution in the simplest of all worlds, which is composed of letter-zymes: strings that are at the same time phenotypes and genotypes. Moreover, the enzymatic activity to make mutations is giving to oblivion. In that way, we can pay attention to the evolutionary process in abstracto.***

13.1 Evolution

With the developed tools, we can compose our first simulation of evolution. At this moment in time, this simulation is just a game to test our programming skills: the apparent purpose is to devise an evolutionary process that renders a stable population. The considered population consists of strings in which its members need to feed to reproduce and grow but have parasites that cut part of their tails. There is also a hand-given selection against excessive growing. We also simulate an observer that wants to know what happens to the minimum and maximum lengths of the strings.

We will simulate **artificial selection**, as that made by a farmer that ex-profeso wants caves with intermediate weight and kills or sells all the others. By contrast, **natural selection** appears as an indirect or **emergent result** of operations that are not directly tied to reproduction, say, from the ability to distinguish mature fruits from immature ones. Our next simulation is of artificial selection.

255. ***Initialization***

The complete code of our simulation could look like a nightmare, so let us introduce it in two or three steps. The first one is just the initialization

```
//Program E255 Letterzymes
//Clone of Program A135 Letterzymes
//Evolution with letterzymes step 1:
//initialization of  a population.

package ejvol5v2p;


import java.util.Random;

public class Letterzymes
{
    // Declaration of global variables.
    // They are used all throughout the whole class.

    // Individual  is an array of strings.
    static String Individual[ ];//Declaration
    static int NIndiv ; //declaration
    static String PatternParasite, PatternFood;
    // Order  is an array of integers.
    static int Order[];
    static String b;
    static int Gen;
    static int ReportMin[], ReportMax[];


    // Turn on of the random generator
    static Random r  = new Random();

    // This method generates random upper chars
    private  static  Character  randomChar()
    {
        return  ( char ) ( r.nextInt( 26 ) + 65  );
    }

    /* We generate NIndiv individuals (strings)
 ten characters long.
 Sequences are completely random */
    private static void Initialization( )
    {
        //Formal declaration of our array.
        Individual = new String[100];
        for(int i = 0; i< NIndiv; i++)
        {
            //Initialization  in blank.
```

```
            Individual[i]="";

            //An individual is assembled char by char
            for(int j = 0; j< 10; j++)
            {
                //We generate a random char
                Character c =  randomChar();
                //System.out.println( "char = " + c);
                //Char c is converted into a string s
                String s = ""+c;
                //s is concatenated to Individual[i]
                Individual[i] = Individual[i]+s;
            }
            System.out.println("Individual " +  i);
            System.out.println( Individual[i]);
        }

        //We generate a parasite pattern
        PatternParasite = "PARASITE";
        //We generate a food pattern
        PatternFood = "ALGAE";
    }


    public static void main(String[] args)
    {
        //The population has NIndiv members, NIndiv >10
        NIndiv = 15;
        Individual= new String[NIndiv];
        Initialization( );
    }
}//End of Program E255 Letterzymes
```

256. ***Exercise*** *Run the program and play with the code.*

257. ***The heart***

The heart of the simulation is contained in the next method. Let us observe how natural it looks like.

```
//This is the heart of the process
private static void Dynamics()
{
    //All individuals feed
    for(int i = 0; i< NIndiv; i++)
    {
```

```
        //The string Algae is digested and inserted
        //into the individual[i] char by char
        /*System.out.println( "The hungry individual "
                    + i + " is " + Individual[i]); */
        feeding(i);
        /*System.out.println(  "The full individual "
             + i + " is " + Individual(i)); */
        //All individuals are tested by the parasite,
        //which corrodes any  large substring
        //similar to it located at any end.
        parasitation(i);
        /*System.out.println(  "Pos-parasitation individual "
             + i + " is " + Individual(i));*/
        //Very long chains are attacked recurrently,
        //the same as very large fishes in the sea.
        Purge(i);
        /*System.out.println(  "The purged  individual "
             + i + " is " + Individual(i)); */
    }
    //The top ten are allowed to  reproduce:
    //a child occupies the place of the shortest individual
    Reproduction();
    //The new population is subjected to mutation
    Mutation();
}
```

258. ***The complete code follows:***

```
//Program E258 Letterzymes2
//Clone of Program A138 Letterzymes2
//Evolution of a population
//of strings.

//Strings are at the same time
//genotype and phenotype.
//They are like ribozymes.


package ejvol5v2p;



import java.util.Random;

public class Letterzymes2
```

```
{

//Pre-declaration of global variables.
//They are used all throughout the whole class.

//Individuals are kept in the array
//Individual[]. It is an array of strings.
//The number of individuals must be
//less than limit

private static final int LIMIT = 1000;
//How many generation to run
private static final int  NGEN = 2;
//Define here when a string is long enough
//to be purged
private static final int LARGESIZE = 1;
static private final String INDIVIDUAL[ ] = new String[100];
static private final int NINDIV = 15;
//Rank  keeps the rank of each individual.
static private final int RANK[] = new int[LIMIT];
static private String b;

static private int ReportMin[], ReportMax[];


//Turn on of the random generator
private static final Random RANDOM  = new Random();

//This method generates random chars
private  static  Character  randomChar()
{
    return  ( char ) ( RANDOM.nextInt( 26 ) + 65  );
}

/* We generate NINDIV individuals (strings)
 ten characters long.
 Sequences are completely random */
private static void Initialization( )
{
    System.out.println("ORIGINAL POPULATION");
    for(int i = 0; i< NINDIV; i++)
    {
        //Initialization  in blank.
        INDIVIDUAL[i]="";

        //An individual is assembled char by char
```

```
        for(int j = 0; j< 10; j++)
        {
            //We generate a random char
            Character c =  randomChar();
            //System.out.println( "char = " + c);
            //Char c is converted into a string s
            String s = ""+c;
            //s is concatenated to INDIVIDUAL[i]
            INDIVIDUAL[i] = INDIVIDUAL[i]+s;
        }
        System.out.println("Individual " +  i);
        System.out.println( INDIVIDUAL[i]);
    }



    for(int i = 0; i< NINDIV; i++)
        RANK[i]=0;

    ReportMin= new int[NGEN];
    ReportMax= new int[NGEN];
    for(int gen = 0; gen < NGEN; gen++)
    {
        ReportMax[gen]=0;
        ReportMin[gen]=0;
    }



}


//This method takes substring a
//and posits it at place Start inside c
private  static  String
      Insert(String a, int Start, String c)
{

    String  s1, s2;
    // We copy  the left part of c
    s1 = c.substring(0, Start);
    // We copy  the right part of c
    s2 = c.substring(Start);
    // We concatenate the left part of c
    //to a to the right part of c
    String z = s1 + a + s2;
```

```
    /*System.out.println("Original c " + c);
    System.out.println("Original a " + a);
    System.out.println("Start " + Start);
    System.out.println("PosInsertion c = " + z);
    */
    return z;
}

//The letters of "ALGAE" are inserted
//into the string of individual i
private static  void feeding(int i)
{
    String a;
    // We measure the length of Individual[i]
    int l = INDIVIDUAL[i].length();
    //Null strings cannot feed anyhow
    if (l == 0) ;
    else //Non null strings  feed on ALGAE
    {
        a = "A";
        int Start = RANDOM.nextInt(l);
        INDIVIDUAL[i] = Insert(a, Start, INDIVIDUAL[i]);
        a = "L";
        l = INDIVIDUAL[i].length();
        Start = RANDOM.nextInt(l);
        INDIVIDUAL[i] = Insert(a, Start, INDIVIDUAL[i]);
        a = "G";
        l = INDIVIDUAL[i].length();
        Start = RANDOM.nextInt(l);
        INDIVIDUAL[i] = Insert(a, Start, INDIVIDUAL[i]);
        a = "A";
        l = INDIVIDUAL[i].length();
        Start = RANDOM.nextInt(l);
        INDIVIDUAL[i] = Insert(a, Start, INDIVIDUAL[i]);
        a = "E";
        l = INDIVIDUAL[i].length();
        Start = RANDOM.nextInt(l);
        INDIVIDUAL[i] = Insert(a, Start, INDIVIDUAL[i]);
    }
}


//This Function replaces all occurrences of
//substring a by b inside c
private static String
Substitution(String a, String b, String c)
```

```
{
    String z = c.replace(a,b);
    /*System.out.println("original a " + a);
    System.out.println("Original b " + b);
    System.out.println("Original c " + c);
    System.out.println("replaced c " + z);
    */
    return z;
}


//This function or method deletes all occurrences
//of a from c.
static private String Delete(String a, String c)
{
    b = "";
    return Substitution(a, b, c);
}


//Here we see what a PARA-SITE  does to individuals.
private  static void parasitation(int i)
{
    int l  =  INDIVIDUAL[i].length();
    //The parasite deletes from the
    //beginning of the individual
    //any substring   matching "PARA".
    //We copy to Beginning the first
    //4 chars of INDIVIDUAL[i]
    if ( l > 4)
    {
        String Beginning =  INDIVIDUAL[i].substring(0,4);
        String s = "PARA";
        // We compare Beginning and s
        if (Beginning.equals(s))
            INDIVIDUAL[i] = INDIVIDUAL[i].substring(4);

        //The parasite deletes from the tail
        //of the individual
        //any substring   matching "SITE".
        String Ending =  INDIVIDUAL[i].substring(4);

        String t = "SITE";
        // We compare Ending and t
        if (Ending.equals(t))  INDIVIDUAL[i] =
                    INDIVIDUAL[i].substring( 0,l - 4);
```

```
    }
 }


//In this method individuals are mutilated
//as a measure to hinder overgrowth.
//We delete from the original string
//a substring of a random length beginning with
//a random Start position.
private static void Purge(int i)
{
    String c, s1, s2;
    int lc, Start, n;
    c = INDIVIDUAL[i];
    lc = c.length();
    if (lc > LARGESIZE)
    {
        Start = RANDOM.nextInt(lc);
        //We copy  the first part of c
        s1 = c.substring(0, Start);
        // We throw a random number
        n = RANDOM.nextInt(lc - Start+1);
        //We copy  the right part of c
        // after Start + n places
        s2 = c.substring( Start+ n);
        //We concatenate the left part of c
        //to  the right part of c
        INDIVIDUAL[i] = s1 + s2;
    }
}


//Individuals are sorted by fitness.
//Fitness = length, as an indicator of complexity.
private static void Sorting()
{
    int   Champ;
    //Necessary default initialization
    int Fitness[] = new int[100];
    for(int i = 0; i< NINDIV; i++)
        Fitness[i] =0;
    //We define a fitness function equal to the length
    //of the string
    for(int i = 0; i< NINDIV; i++)
    {
        String s = INDIVIDUAL[i];
```

```
        Fitness[i] = s.length();
        //System.out.println(INDIVIDUAL[i] + Fitness[i]);
    }
    //We sort individuals by fitness
    for(int i = 0; i< NINDIV;i++)
    {
        Champ = 0;
        for(int j = 0; j< NINDIV;j++)
            if (Fitness[j] > Fitness[Champ])   Champ = j;

        //The array Rank keeps a record of fitness
        //by decreasing order.
        RANK[i] = Champ;
        //The champ leaves the game
        Fitness[Champ] = 0;

    }
    /*
    //Report in decreasing order
    System.out.println("\nSorting :");
    for(int i = 0; i< NINDIV;i++)
        System.out.println(INDIVIDUAL[RANK[i]]);
    */
}


//The top ten fill the world
    private  static void Copying()
    {
        int   k;
        for(int i = 0; i< 10; i++)
        {
            int  l = 10*(10-i);
            for(int j = 0; j< 10; j++)
            {
                k = l -j-1;
                INDIVIDUAL[k] = INDIVIDUAL[RANK[i]];
                /*System.out.println("Ind " + k
                        + " replaced by top" + i);*/

            }
        }

        //System.out.println(  "THE NEW POPULATION IS");

        // for(int i = 0; i <  NINDIV;i++)
```

```
        //System.out.println(  INDIVIDUAL[i]);
    }

private static void Reproduction()
{
    //Individuals are sorted by length
    Sorting();
    //for(int i = 0; i< 10;i++)
    //System.out.println(  INDIVIDUAL(Rank(i))
    //The top ten produce a copy that
    //substitutes the bottom ten.
    Copying();
}

// From individual i
// the fist n  chars are deleted
private static   void Deletion(int i)
{
    int l, n;
    l = INDIVIDUAL[i].length();
    n = 0;
    if (l > 0)  n = RANDOM.nextInt(l);
    INDIVIDUAL[i] = INDIVIDUAL[i].substring(n);
 }

//This method inverts a
static private   String Inversion(String a)
{
    int  la = a.length();
    //Output
    String  d  = "";

    //We copy the first character of a to the last place
    //and so on.
    for(int  i = 0; i < la; i++)
     {
        char charI = a.charAt(i);
        d = charI + d;
     }
    return d;
}

//Local inversion of a substring inside individual i.
private static void innerInversion(int i)
{
    String sLeft, sRight;
```

```
    int start;
    int n;
    String c = INDIVIDUAL[i];
    int lc = c.length();
    if (lc > 1)
    {
        start = RANDOM.nextInt(lc);
        //We copy  the first part of c
        sLeft = c.substring(0, start);
        // We throw a random number
        n = RANDOM.nextInt(lc - start+1);
        //Inner substring
        String a = c.substring(start, start + n);
        String d = Inversion(a);
        //We copy  the right part of c
        // after Start + n places
        sRight = c.substring( start+ n);
        //We concatenate the left part of c
        //to  the right part of c
        INDIVIDUAL[i] = sLeft + d + sRight;
    }
    /*System.out.println("Input  =   " + c + "  " + n);
    System.out.println("Output  =  " + INDIVIDUAL[i]);
    */
}

//We consider only two types of mutation:
//deletion of the beginning
//and inversion of an inner part.
private static void Mutation()
{
    //Which mutation affects which individual
    //is a matter of gambling.
    //We throw a 0 else a 1
    for(int i = 0; i< NINDIV; i++)
    {
        int Luck = RANDOM.nextInt(2);
        if ( Luck == 1) Deletion(i);
        else  innerInversion(i);
    }
}

//This is the heart of the process
private static void Dynamics()
{
    //All individuals feed
```

```
    for(int i = 0; i< NINDIV; i++)
    {
        //The string Algae is digested and inserted
        //into the individual[i] char by char
        /*System.out.println( "The hungry individual "
                    + i + " is " + INDIVIDUAL[i]); */
        feeding(i);
        /*System.out.println(  "The full individual "
             + i + " is " + INDIVIDUAL(i)); */
        //All individuals are tested by the parasite,
        //which corrodes any  large substring
        //similar to it located at any end.
        parasitation(i);
        /*System.out.println(  "Pos-parasitation individual "
             + i + " is " + INDIVIDUAL(i));*/
        //Very long chains are attacked recurrently,
        //the same as very large fishes in the sea.
        Purge(i);
        /*System.out.println(  "The purged  individual "
             + i + " is " + INDIVIDUAL(i)); */
    }
    //The top ten are allowed to  reproduce:
    //a child occupies the place of the shortest individual
    Reproduction();
    //The new population is subjected to mutation
    Mutation();
}


private static void report(int gen )
{
    System.out.println("\nPopulation at gen = " + gen );
    for(int j = 0; j< NINDIV; j++)
    {
        System.out.println("Gen " + gen + " Ind "
                    + j +" " +  INDIVIDUAL[j]);
    }
}

//Min and Max lengths are detected
private static void Observation(int gen)
{
    int m, Min, Max;
    Max = 0;
    Min = 32000000;
    for(int j = 0; j< NINDIV; j++)
```

```
    {
        m =  INDIVIDUAL[j].length();
        /*System.out.println(  "The Individual  "
           + INDIVIDUAL(j) + " has length " + m);*/
        if (m > Max)  Max = m;
        if (m < Min)   Min = m;
    }

    /*System.out.println(  "Generation "  + Gen
    + " MinLength = " + Min + " MaxLength = " + Max);*/
    ReportMin[gen] = Min;
    ReportMax[gen] = Max;
}

public static void main(String[] args)
{
    Initialization( );

    for(int gen = 0; gen < NGEN; gen++)
    {
        //System.out.println( "Beginning3");
        Dynamics();
        report(gen);
        //The aim of observation is to record
        //the minimum and maximum lengths
        //of the individuals in each generation.
        Observation(gen);
    }
    //Report of the dynamics of min and max indicators.
    System.out.println( "GENERATION MIN AND MAX VALUES");
    for(int Gen = 0; Gen < NGEN; Gen++)
        System.out.println( Gen + "\t" + ReportMin[Gen]
        + "\t" + ReportMax[Gen]);

}

}//End of Program E258 Letterzymes2
```

259. ***Exercise*** *Run the program and play with the code. Once and forever get used to test methods one by one in order to verify that they do what is promised: to test a method, devise a suitable input, process it with the method and compare the output with what is expected according to your intention. Repeat the procedure many times, a task that sometimes can be done automatically by enlarging the number of generations or of individuals of the population. The appropriate code can be inserted just at a method and be called as soon as it*

is possible. Many tests are already at hand by activating the corresponding debugger lines.

260. ***Exercise*** *In the previous program we have implemented selection and mutation. To simulate evolution completely we need to include recombination. Add it to the previous program. Why it is necessary? Selection has as function to favor the reproduction of the fittest while that of mutation is to create variability. On the other hand, recombination joins together partial solutions that were discovered by different individuals and so recombination is the evolutionary way of expressing the power of the whole population that cooperates to improve performance. Recombination of two strings produces a third one composed of parts of the former two. Example: If we have the string* `abcdefg` *and* `1234567` *then a recombinant might be* `abcd567` *and another* `12cdefg`*. In nature recombination takes two strings as input and produces two strings as output but we will produce only one. The Author forgot to include recombination in this program as in the original as in the second versions of the first volume. This blatant error was dragged to other programs because many programs were created by modification of extant ones, i.e., by evolution. (Historically, this same fault was committed by the precursor of this field that is known as Genetic Algorithms.)* *Answer*

13.2 Review

We have simulated evolution using a world composed of letterzymes: strings of letters, which play the roles of genotype and phenotype, and that need no catalysts. Evolution is simple but its simulation may look a bit complex. Or, there are many families of mutation, recombination and of selection procedures. And for all of them Java is sufficient. No biology at the horizon, just a test of our Java skills.

Chapter 14

Taming evolution

`Evolution is a universal tool for problem solving`

261. ***Motivation and purpose.*** *Evolution can be tamed for the welfare of man. It already has been done in artificial selection and it is done now by means of simulations. The simulation of evolution consists of two steps. First: the design of an evolutionary environment defined by the chosen combinatorial basis what to tinker with, their rules for combinations and the form as reproduction is tied to surviving and function. Second: the actual unfolding of evolution. In present times, evolution is a universal tool for optimization problem solving. We prove this by showing that a problem picked at random, actually the first problem we find in our way, can be solved by evolution. Next, we assess the performance of our evolutionary environment.*

14.1 Taming evolution

You can tame an elephant if only you have a lot of time, wisdom and money. The same happens with evolution. But beware: some elephants have killed their masters! By the same token, evolution is a danger for you because it is very attractive but very difficult to subjugate and at the end you can appear with results that are a shame.

262. ***Genetic algorithms***

The official name given in science to a simulation of evolution that pretends to find a very good solution to an optimization problem is **genetic algorithm**. It builds a digital biological world where individuals encode possible solutions to a given optimization problem and reproduction is granted in agreement with the degree of fitness of each potential solution to solve the problem. A genetic algorithm not necessarily produces the optimal solution but if it is well designed, a good approximation can eventually be found.

263. ***Let us fix ideas***

Let us think of our optimization problem, which is to find close algebraic expressions that could be very good approximations to the cumulative function of the standardized normal distribution. We have made some informal work that allowed us to choose the h family as a suitable candidate for our approximations. Our next task is to find the best representative of that family: we need to find the element that is generated by a given family and that most closely resembles a target function. We consider that this problem is not trivial and so we see here a good occasion to explore the possibilities of evolution. Notice that we are implementing our evolutionary program with combinatorial bases: we have explored various families of functions, we picked up the most promising from these and now we prepare to find the best exponent of that family.

264. ***Detailed formulation***

Let us be very explicit in the formulation of our problem that must be solved by a genetic algorithm.

`Our problem reads`: find an easy to calculate function, not piecewise defined, to approximate the cumulative function of the standardized normal distribution

$$\Phi(z) = \int_{-\infty}^{z} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx$$

Once we have a problem, the next step is `to devise a combinatorial basis`. Our previous informal work allowed us to single out the h family as a promising combinatorial family:

$h_{m,k,b}(z) = \frac{kz}{2\sqrt[m]{(kz)^m + b}}$

where parameters m, k, b are real numbers. This family defines our evolutionary environment in which our genetic evolution will proceed. We might say that our combinatorial basis is composed of all those values that these parameters can take on: m (even integer) , k (positive real), b (positive real).

If we have a problem and a combinatorial basis, we need to specify the `optimization problem and fitness`, i.e., the rules for deciding who lives and who dies, who reproduces and in which proportion. In our case, we must `minimize the error function` given by

$d(\Phi, h) = \| f - g \| = Max \mid \Phi(x) - h(x) \mid$

Thus, the elements of our digital biological world will be composed of functions whose fitness is determined by the degree of resemblance with the target function, the cumulative distribution function of the standardized normal distribution.

`Our rule for living and let dying:` every extant individual can live during one generation, after which it dies. It will be replaced by its offspring during the next generation. And so on.

`Our rule for reproduction:` the top ten will be allowed to reproduce with the condition of keeping a constant population size.

`Our rule for mating:` at random. Actually, mating takes two individuals, recombine them and produces a new one.

`Our rule for mutation:` an individual of the new generation is subjected to mutation to create variability. This is the procedure that allows the exploration of the evolutionary environment.

`Our rule for recombination:` string `12345` and string `abcde` recombine into, say, `123de` or `ab345`.

265. ***Encoding.***

We have an optimization problem and we have planned our evolutionary environment. The process of connection of these two elements is called **encoding**. To encode a problem in evolutionary form, we need a rule to represent a plausible solution to our problem by strings, which play the role of DNA. We also need the inverse rule of decoding that associates to any string an approximate solution to the formulated problem. Let us take some pains to understand how this is done.

Suppose that we need to encode a family of numbers of the form 1234 as string. Solution: convert each number to a string. So, number 1234 is encoded as "1234".

Suppose now that we need to encode numbers of the form ± 1234. We encode them as a string with the convention that its first char must indicate the sign. So 1234 is encoded as "+1234", while -1234 is encoded as "-1234".

Let us consider now a number of the form -12.3456789123456789. In this case we deal with numbers with, say, 18 ciphers and a point that can be in any place from zero to 18. We can encode this family of numbers as a string, whose first char denotes the sign, next 18 char represent the ciphers of the numbers and the last two denote the place where the point sits. So, we can encode -12.3456789123456789 as

"-123456789123456789P02"

which reads: we have a negative number whose ciphers are. 123456789123456789 and that have the decimal point in the second place, counting from zero.

If we have to encode various numbers, we form an ordered set an encode them one by one and next we concatenate the resulting strings in the corresponding order. Say, the string "-123456789123456789P02+9876543219876543211P11" represents two numbers, -12.3456789123456789 and 98765432198.7654321.

266. ***The code follows. We use type double and ordinary arithmetic. The cumulative function of the standardized normal distribution,*** $F(z)$ ***is found by using the trapezoidal rule and then an evolutionary process is created to estimate the parameters*** m, k, b ***to find the member of the*** h***-family that best fits*** $F(z)$***.***

```
//Program E266 HFamily3
//Computes the  integral under the standard Gauss bell
//in within 0 and a positive z.
//Method 1: trapezoidal rule.
//Method 2: algebraic approximation given by
//h(z) = 0.5 + (kz)/(2 mth-Root((kz)^m + b)).
//This is the hFamily.
//Optimal parameters m,k,b are found by a genetic algorithm.
//The maximal approximation error is reported.


package ejvol5v3p;

import java.util.Random;

public class HFamily3 {

    private static double pi = 3.14159265358979323846;
    //Values of the true cumulative function
    private static double zVect[] = new double[10000001];
    //Calculations are done for z in (0, zMax)
    private static double zMax = 10;
    //Number of sample points
    private static int N = 10000;

    //***************Genetic part***********
    // Individuals are kept in the array
    // Individual[]. It is an array of strings.
    //Each individual encodes three numbers
    //corresponding to the 3 parameters of the
    //hFamily.
    // The number of individuals must be
    // less than limit.
    static int limit = 50000;
    static double Fitness[];
    static String Individual[], Individualc[];
    //Actual number of individuals
    static int nIndiv;
    //Number of chars per parameter
    static int nChars = 15;
```

```
static int Order[];
static String b;
static int generation;
static int nGen;
static int ReportMin[], ReportMax[];
static double mutationRate;
static double oldError, newError;
static double deltaError;
static boolean test, testEncoding;

// Mandatory initialization
private static void initialize(long N) {
    for (int i = 0; i <= N; i++) {
        zVect[i] = 0.5;
    }
}

//*********Method 1: Trapezoidal rule*****************
//Returns the density function of the standardized
//normal distribution.
private static double zFunction(double z) {
    double f = 1 / Math.pow(2 * pi, 0.5) * Math.exp(-z * z / 2);
    return f;
}

//To use this method for another distribution,
//instead of  zFunction(z), write and make a call for your
//new distribution.
private static double f(double z) {
    double f = zFunction(z);
    return f;
}

private static void trapRule() {
    initialize(N);
    title(N);
    //Trapezoidal rule for the cumulative function
    zHalfBodyTrapReuse(zMax, N);
}

//Returns 0.5 + the integral under the z-density function
//in within 0 and z. Precision=1/N.
//Trapezoidal rule adapted for reuse.
private static void zHalfBodyTrapReuse(double zMax,
        long N) {
    /**
```

```
         * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciph
         */
        double Nr = N;
        double h = zMax / Nr;
        //System.out.println("h="+h);
        double sum = 0;
        for (int j = 1; j <= N; j++) {
            sum = sum + (h / 2) * (f(h * (j + 1)) + f(h * j));
            zVect[j] = sum + 0.5;
        }
    }

    //************** Method 2: algebraic fitting**********
    //Returns an ad hoc algebraic approximation
    //to the cumulative function of the
    //standardized normal distribution.
    private static double hFunction(double z, double k,
            double m, double b) {
        double p = 1;
        p = p / m;
        double hz = 0.5 + (k * z)
                / (2 * Math.pow(Math.pow((k * z), m) + b, p));
        return hz;
    }

    //To use this method for another approximation,
    //instead of  hFunction(z), write and make a call for your
    //new proposal.
    private static double fitting(double z, h hf) {
        double m = hf.m;
        double k = hf.k;
        double bb = hf.b;
        double f = hFunction(z, k, m, bb);
        return f;
    }

    //**************Comparison of two methods**************
    private static void title(long N) {
        System.out.println("cum = 0.5 + Area under "
                + " the standardized bell in within zero and z.");
        System.out.println("That area is calculated by "
                + "the Trapezoidal rule");
        System.out.println("with " + N + " divisions "
                + "in within 0 and 10.");
        System.out.println("A genetic algorithm calculates ");
        System.out.println("the optimal parameters of the h family");
```

```
        System.out.println("to best fit cum.");
    }

    //The maximal error of approximation is found
    private static double findMaxError(double zMax, int N, h hf) {
        double maxError = 0;
        double error;
        double z;
        double Nr = N;
        double h = zMax / Nr;
        double hz;
        //System.out.println("h = " + h);
        for (int i = 0; i < N; i++) {
            z = i * h;
            hz = fitting(z, hf);
            error = zVect[i] - hz;
            error = Math.abs(error);
            /*System.out.print("z="+z + "  cum = " +  zVect[i]);
  System.out.print(" hz= "  +  hz);
  System.out.println(" Error = " + error);*/
            if (error > maxError) {
                maxError = error;
            }
        }
        return maxError;
    }

    //*********************************
    //Genetic algorithm to optimize the
    //parameters of the hFamily.
    //We improve program B78, volume II, that finds the root
    //of a fifth degree polynomial.
    //*********************************
//*****************************************
//****************h CLASS ******************
//*****************************************
//This auxiliary class unites  fundamental
//parameters into an object.
//This encoding eases reuse of old code
    private static class h {

        int m;
        double k;
        double b;

        //Mandatory initialization
```

```
        h(int i, double kk, double bb) {
            m = i;
            k = kk;
            b = bb;
        }

        //Mandatory initialization
        h(h hf) {
            m = hf.m;
            k = hf.k;
            b = hf.b;
        }

        //Printing method.
        static void print(h hf) {
            System.out.print(" m  = " + hf.m);
            System.out.print(" k  = " + hf.k);
            System.out.print(" b  = " + hf.b);
        }

    }//end of class h

//  Turn on of the random generator
    static Random r = new Random();

    //This method generates random chars
    // that represent digits
    private static char randomDigit() {
        return (char) (r.nextInt(10) + 48);
    }

    //m is positive even integer less than 9
    private static char generateM() {
        int m = r.nextInt(4) + 1;
        m = (2 * m + 48);
        char mChar = (char) m;
        return mChar;
    }

    //Returns a string of digital numbers, a basic string
    private static String generateSubString(int nChars) {
        String w = "";
        //An individual is assembled char by char
        for (int j = 1; j <= nChars; j++) {
            // We generate a random numeric char
            char c = randomDigit();
```

```
            String s = "" + c;
            //System.out.println( "char = " + c);
            // Char c is added to w
            w = w + s;
            //System.out.println( "w = " + w);
        }
        return w;
    }

    //A power  is generated
    //Range runs form 0 to nChars
    private static String addPower(int nChars) {
        Integer i = r.nextInt(nChars);
        String s = i.toString();
        if (s.length() == 1) {
            s = '0' + s;
        }
        return s;
    }

    //A string encoding for a sign is generated
    private static String generateSign() {
        int spk = r.nextInt(2);
        char s = '+';
        if (spk == 0) {
            s = '-';
        }
        String t = "" + s;
        return t;
    }

    //A string encoding for a number is generated
    private static String generateString(int nChars) {
        String k = generateSubString(nChars);
        if (test) {
            System.out.println("  String = " + k);
        }
        String pk = addPower(nChars);
        //Powers run contrary wise
        String s = "" + generateSign();

        k = k + 'P' + s + pk;
        if (test) {
            System.out.println("Power  = " + pk);
            System.out.println("SignPower  = " + s);
            System.out.println("As String   = " + k);
```

```
        }
        return k;
    }

    //Declarations and default initializations
    // of other arrays.
    private static void otherInit() {
        Order = new int[limit + 1];
        ReportMin = new int[limit + 1];
        ReportMax = new int[limit + 1];
        Fitness = new double[limit + 1];
        for (int i = 0; i < nIndiv; i++) {
            Order[i] = 0;
            ReportMin[i] = 0;
            ReportMax[i] = 0;
            Fitness = new double[limit + 1];
        }
    }

    private static String generateIndividual() {
        //m is generated
        char m = generateM();
        if (test) {
            System.out.println("m = " + m);
        }
        //k is generated
        if (test) {
            System.out.println("k is generated");
        }
        String k = generateString(nChars);
        //b is generated
        if (test) {
            System.out.println("b is generated");
        }
        String bbb = generateString(nChars);
        String ind = "" + m + k + bbb;
        return ind;
    }

    /* We generate nIndiv individuals (strings)
encoding for three parameters.
All parameters are positive.
Sequences are  random */
    private static void Initialization() {
        //Formal declaration of our array.
        Individual = new String[limit];
```

```
        if (test) {
            System.out.println("ORIGINAL POPULATION");
        }
        for (int i = 0; i < nIndiv; i++) {
            if (test) {
                System.out.println("i = " + i);
            }
            Individual[i] = "";
            Individual[i] = generateIndividual();

            if (test) {
                System.out.println("bString = " + b);
                System.out.println("All = " + Individual[i]);
                System.out.println();
            }
        }
        otherInit();
    }

    //String s is decoded into a number
    private static double recoverNumber(String s) {
        // We get the sign of power of k
        // which could be + else -.
        char sp = s.charAt(nChars + 1);
        if (test) {
            System.out.println("Sign of power = " + sp);
        }
        // We get the   power of k
        // which ranges from 00 to 99.
        char s1 = s.charAt(nChars + 2);
        int p1 = Character.getNumericValue(s1);
        s1 = s.charAt(nChars + 3);
        int p2 = Character.getNumericValue(s1);
        int power = p1 * 10 + p2;
        if (sp == '-') {
            power = -power;
        }
        if (test) {
            System.out.println("Power = " + power);
        }

        //*********** We get k
        String k = s.substring(0, nChars);
        if (test) {
            System.out.println("as substring = " + k);
        }
```

```
        double number = 0;
        for (int j = 0; j < nChars; j++) {
            char s3 = k.charAt(j);
            if (s3 != '0') {
                int l = Character.getNumericValue(s3);
                int q = j + 1;
                double number1 = (int) l * Math.pow(10, power - q);
                number = number + number1;
            }
        }
        if (test) {
            System.out.println("as number = " + number);
        }
        return number;
    }

    // This method transforms a string in a number
    private static h decoder(String s) {
        if (test) {
            System.out.println(" Individual =   = " + s);
            System.out.println("length of s  = " + s.length());
        }
        /*System.out.println( "place nChars + 2 = "
                + s.charAt(nChars + 2));
System.out.println( "place nChars + 6 = "
                + s.charAt(2*nChars + 6));
         */
        //We recover m:
        char ms = s.charAt(0);
        int m = Character.getNumericValue(ms);
        if (test) {
            System.out.println("m = " + m);
        }
        //We get the string that encodes for k
        String kString = s.substring(1, nChars + 5);
        if (test) {
            System.out.println("kString = " + kString);
        }
        if (test) {
            System.out.println("Recovering k");
        }
        double k = recoverNumber(kString);
        if (test) {
            System.out.println("k as number = " + k);
        }
        //We get the string that encodes for b
```

```
        if (test) {
            System.out.println("Recovering b");
        }
        String bString = s.substring(nChars + 5);
        if (test) {
            System.out.println("bString = " + bString);
        }
        double bb = recoverNumber(bString);
        if (test) {
            System.out.println("b as number = " + bb);
        }
        h hf = new h(m, k, bb);
        return hf;
    }

    private static void report() {
        h hf = decoder(Individual[Order[0]]);
        double error = findMaxError(zMax, N, hf);
        System.out.print("Error = " + error);
        h.print(hf);
        System.out.println();
    }

    //Individuals are sorted by fitting
    private static void Sorting(int gen) {
        int Champ;
        for (int i = 0; i < nIndiv; i++) {
            if (test) {
                System.out.println("i = " + i);
            }
            // Individual[i] is a string,
            // it is a genotype that encodes for hf;
            // hf is a set of parameters: it is the phenotype
            h hf = decoder(Individual[i]);
            double error = findMaxError(zMax, N, hf);
            if (test) {
                System.out.println("i = "
                        + i + " maxError  = " + error);
            }
            if (Math.abs(error) < 0.000000000001) {
                System.out.println("Gen = " + gen
                        + " Solution found");
                System.out.println("Error = " + error);
            }
            Fitness[i] = 1 / (1 + error * error);
            if (test) {
```

```
                System.out.println();
            }
        }
        //Sorting
        for (int i = 0; i < nIndiv; i++) {
            Champ = 0;
            for (int j = 0; j < nIndiv; j++) {
                if (Fitness[j] >= Fitness[Champ]) {
                    Champ = j;
                }
            }

            //The array Order classifies individuals by  fitness
            // by equal or decreasing order.
            Order[i] = Champ;
            Fitness[Champ] = 0;
            if (test) {
                System.out.println(i + "th ind. is  No "
                        + Champ);
            }
        }
    }

    //Each individual of the top ten
    // produces 10  copies.
    private static void Reproduction() {
        if (test) {
            System.out.println("Reproduction");
        }
        if (test) {
            System.out.println("The best = " + Order[0]);
        }
        Individualc = new String[limit];
        int counter = 0;
        for (int top = 0; top < 10; top++) {
            for (int j = 0; j < 10; j++) {
                Individualc[counter] = Individual[Order[top]];
                counter = counter + 1;
            }
        }
        /*
     for(int j = 0; j< counter; j++)
     Individual[j] = Individualc[j];
         */
        System.arraycopy(Individualc, 0, Individual, 0, counter);
```

```
    }

    //A mutation of ind j is done at place placeMut
    //Corrected
    private static String mutateIndj(int j, int placeMut) {
        //Test:
        //placeMut = 19;
        if (test) {
            System.out.println("mutatedIndj placeMut = " + placeMut);
        }

        char c;
        String sign;
        String power;
        String s = Individual[j];
        if (test) {
            System.out.println(" indj         = " + s);
        }
        //There are four cases: m, power, sign of power,
        //or basic string.
        //Case m:
        if (placeMut == 0) {
            c = generateM();
            s = c + s.substring(1);
            Individual[j] = s;
            if (test) {
                System.out.println(" m mutated = " + s);
            }
        }
        //Case sign
        if (placeMut == nChars + 2) {
            sign = generateSign();
            s = s.substring(0, nChars + 2)
                    + sign + s.substring(nChars + 3);
            if (test) {
                System.out.println(" Sign mutated = " + s);
            }
        }
        if (placeMut == 2 * nChars + 6) {
            sign = generateSign();
            s = s.substring(0, 2 * nChars + 6)
                    + sign + s.substring(2 * nChars + 7);
            if (test) {
                System.out.println(" Sign mutated = " + s);
            }
        }
```

```
        //Case power:
        if ((placeMut == nChars + 3) | (placeMut == nChars + 4)) {
            power = addPower(nChars);
            s = s.substring(0, nChars + 3)
                    + power + s.substring(nChars + 5);
            if (test) {
                System.out.println("Power mutated = " + s);
            }
        }
        if ((placeMut == 2 * nChars + 7)
                | (placeMut == 2 * nChars + 8)) {
            power = addPower(nChars);
            s = s.substring(0, 2 * nChars + 7)
                    + power + s.substring(2 * nChars + 9);
            if (test) {
                System.out.println("Power mutated = " + s);
            }
        }

        //Case basic string
        if (((0 < placeMut) & (placeMut < nChars + 1))
                | ((nChars + 4 < placeMut)
                & (placeMut < 2 * nChars + 5))) {
            c = randomDigit();
            s = s.substring(0, placeMut)
                    + c + s.substring(placeMut + 1);
            if (test) {
                System.out.println("Basic mutated = " + s);
            }
        }
        return s;
    }

    //We consider only one type of mutation:
    //substitution of one char by another one.
    //The first copy of the winner does not mutate.
    //Corrected
    private static void Mutation() {
        Individualc[0] = Individual[0];
        for (int j = 1; j < nIndiv; j++) {
            //Define place of mutation
            int n = Individual[j].length();
            int placeMut = r.nextInt(n);
            //Will that place mutate?
            double p = r.nextDouble();
```

```
            if (p < mutationRate) {
                Individualc[j] = mutateIndj(j, placeMut);
            }
        }
        //Some individuals are generated ab initio.
        //We hope that they will help to escape
        //from local optimal states.
        int initial = nIndiv - 20;
        if (initial > 0) {
            for (int i = initial; i < nIndiv; i++) {
                if (test) {
                    System.out.println("i = " + i);
                }
                Individualc[i] = "";
                Individualc[i] = generateIndividual();
            }
        }
        /*
            for(int j = 0; j< nIndiv; j++)
            Individual[j] = Individualc[j];
         */
        System.arraycopy(Individualc, 0, Individual, 0, nIndiv);
    }

    //Two strings recombine and produce two offspring.
    //Corrected
    private static void Recombination() {
        /*
            for(int j = 0; j< nIndiv; j++)
            Individualc[j] = Individual[j];
        */
        System.arraycopy(Individual, 0, Individualc, 0, nIndiv);
        for (int j = 1; j < nIndiv; j++) {
            //Define place of mutation
            int m = r.nextInt(nIndiv);
            int n = r.nextInt(nIndiv);
            String a = Individual[m];
            String bb = Individual[n];
            int placeRec = r.nextInt(2 * nChars + 9);
            Individualc[m] = a.substring(0, placeRec)
                    + bb.substring(placeRec);
            Individualc[n] = bb.substring(0, placeRec)
                    + a.substring(placeRec);
        }
        /*
            for(int j = 0; j< nIndiv; j++)
```

```
            Individual[j] = Individualc[j];
        */
        System.arraycopy(Individualc, 0, Individual, 0, nIndiv);

    }

    //Mutation rate can increase
    private static void mutationRedef() {
        String c = Individual[0]; //champ reported
        h hf = decoder(c);
        double error = findMaxError(zMax, N, hf);
        /* System.out.println("gen " + gen +
  " Error " + error);*/
 /* System.out.println("mutation rate = " +
    mutationRate);*/
        newError = error;
        deltaError = Math.abs(newError - oldError);
        /*System.out.println("Error= " + error +
 " deltaError= " + deltaError); */
        oldError = newError;
        if (deltaError < 0.001) {
            mutationRate = 1.1 * mutationRate;
            if (mutationRate > 1) {
                mutationRate = 1;
            }

            /*  System.out.println("gen " + gen + "
    mutRedef");*/
        } else;
    }

    //Overall method
    private static void dynamics(int gen) {
        //Individuals are sorted by fitness
        Sorting(gen);
        //for(int i = 1; i<= 10;i++)
        //System.out.println(  Individual(Order(i))
        report();
        //The top ten are preferentially reproduced
        Reproduction();
        //The new population is subjected to mutation
        Mutation();
        Recombination();
        //every 100 generations, stagnation is revised
        if (gen % 100 == 0) {
            mutationRedef();
```

```
        }
    }

    //Decodes a string into a number
    private static void decode() {
        System.out.println();
        System.out.println("Numeric values at decoding");
        for (int i = 0; i < nIndiv; i++) {
            System.out.println("i = " + i);
            System.out.println("All = " + Individual[i]);
            decoder(Individual[i]);
        }
    }

    //test of encoding and decoding
    private static void test() {
        nIndiv = 1;
        System.out.println("Strings at Initialization");
        Initialization();
        decode();
        trapRule();
        System.out.println("\r Test ");
        h hf = new h(4, 0.4107102082519, 0.096088271997664);
        double error = findMaxError(zMax, N, hf);
        System.out.println("Max error at test= " + error);
    }

    public static void main(String[] args) {
        testEncoding = false; //else true
        if (testEncoding) {
            test();
        } else {
            test = false;
            //The whole cumulative function is estimated
            //by the trapezoidal rule
            trapRule();

            //The parameters in the hFamily are optimized
            //by a genetic algorithm.
            System.out.println("Running ");
            nIndiv = 250;
            Initialization();
            nGen = 300000;
            oldError = 1000;
            mutationRate = 1;
            for (int gen = 1; gen <= nGen; gen++) {
```

```
                System.out.println("Gen = " + gen);
                dynamics(gen);
            }
        }
    }
}//End of Program E266 HFamily3
```

267. ***Exercise.*** *Run the program and play with the code. It is better to launch various replicas at the same time. Verify our previous informal work claiming that the approximation error could be made less than 0.01.*

268. ***Challenge.*** *Change in the previous program the trapezoidal rule by the Simpson rule. Compare results with those of the previous program.*

269. ***The price of perfection***

We made an informal work that allowed us to find parameters with the following values: $m = 4$, $k = 0.7$ and $b = 1$. With these values, the error was approximately 0.017. After running our genetic algorithm, we found cuasi-optimal values that originate an error of 0.0083, roughly the half of that found by informal methods. We see that the price for doubling precision was in this case some 700 lines of code.

In hindsight, a problem that looked very simple has revealed itself as a very difficult one lending itself to a lengthy process of optimization, which generates a clear track of its optimization record. This is recurrently found when someone engages in a battle against complexity.

14.2 Improving our approximation

Our h-family has allowed us to find a good approximation to the cumulative function of the standardized normal distribution. To try to improve our approximation, let us investigate the behavior of the error.

270. ***Error functions for two members of the h-family.*** *The next is an sketch of the error as a function of z, for positive values, when the parameters are optimized:*

$m = 4$; $k = 0.693227030131669$; $b = 0.7790134732911791$.

and for the parameters found by our informal method:

$m = k$; $k = 07$; $b = 1$.

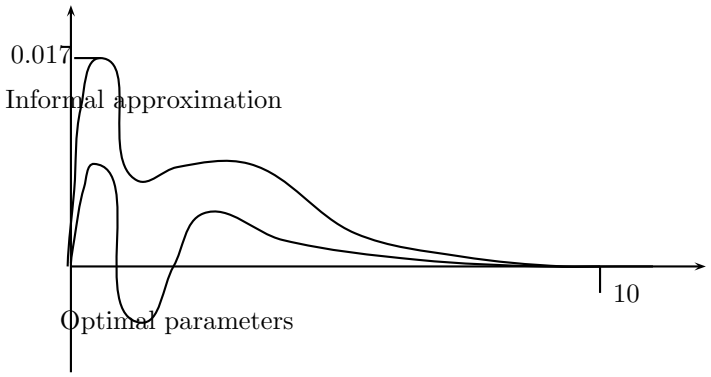


Figure 14.1. Two error functions for two members of the h-family.

In general terms both functions have the same architecture, although the maximal error caused by the optimal parameters is smaller but sometimes negative. Our next move is to try an approximation of the form

$f(z) = \frac{z}{1+z^2}$

whose graphic is the next

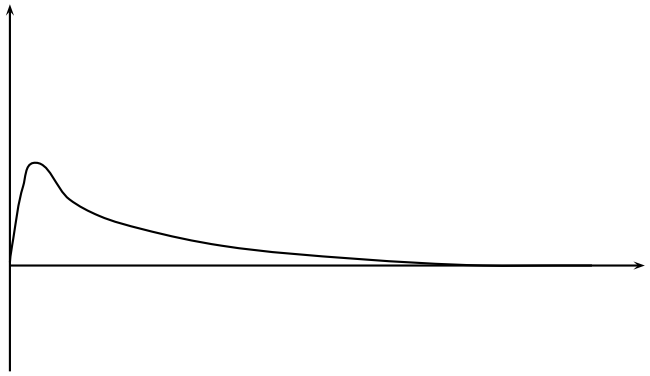


Figure 14.2. If we smooth the error functions of the previous graphic, we get something like this. So, we need to complement the approximation found with the genetic algorithm with a function of this form.

This function is generalized into the next family:

$j_{a,e,i,l}(z) = \frac{az^i}{e+z^l}$

So, we will try an approximation with a family of the next form:

$hj_{m,k,b,a,e,i,l}(z) = \frac{kz}{2\sqrt[m]{(kz)^m+b}} + \frac{az^i}{e+z^l}$

New coefficients a and e must be positive real, while i and l are positive integers, l even and i odd.

271. ***The next code explores how good is the*** $hj-$ ***family to approximate the cumulative function of the*** z ***distribution.***

```
//Program E271 HjFamily
//Computes the  integral under the standard Gauss bell
//in within 0 and a positive z.
//Method 1: trapezoidal rule.
//Method 2: algebraic approximation given by
//hj(z) = 0.5 + (kz)/(2 mth-Root((kz)^m + b)) +
//+ az^i / (e + z^l).
//This is the HjFamily.
//Optimal parameters m,k,b , a, e, i, l are found by
//a genetic algorithm.
//The maximal approximation error is reported.

package ejvol5v3p;



import java.util.Random;

public class HjFamily {

    private static double pi = 3.14159265358979323846;
    //Values of the true cumulative function
    private static double zVect[] = new double[10000001];
    //Calculations are done for z in (0, zMax)
    private static double zMax = 10;
    //Number of sample points
    private static int N = 100000;

    //***************Genetic part***********
    // Individuals are kept in the array
    // Individual[]. It is an array of strings.
    //Each individual encodes three numbers
    //corresponding to the 7 parameters of the
    //jFamily.
    // The number of individuals must be
    // less than limit.
    static int limit = 50000;
```

```
static double Fitness[];
static String Individual[], Individualc[];
//Actual number of individuals
static int nIndiv;
//Number of chars per parameter
static int nChars = 15;
static int Order[];
static String b;
static int generation;
static int nGen;
static int ReportMin[], ReportMax[];
static double mutationRate;
static double oldError, newError;
static double deltaError;
static boolean test;

// Mandatory initialization
private static void initialize(long N) {
    for (int i = 1; i <= N; i++) {
        zVect[i] = 0.5;
    }
}

//*********Method 1: Trapezoidal rule*****************
//Returns the density function of the standardized
//normal distribution.
private static double zFunction(double z) {
    double f = 1 / Math.pow(2 * pi, 0.5) * Math.exp(-z * z / 2);
    return f;
}

//To use this method for another distribution,
//instead of  zFunction(z), write and make a call for your
//new distribution.
private static double f(double z) {
    double f = zFunction(z);
    return f;
}

private static void trapRule() {
    initialize(N);
    title(N);
    //Trapezoidal rule for the cumulative function
    zHalfBodyTrapReuse(zMax, N);
}
```

```
//Returns 0.5 + the integral under the z-density function
//in within 0 and z. Precision=1/N.
//Trapezoidal rule adapted for reuse.
private static void zHalfBodyTrapReuse(double zMax,
        long N) {
    /**
     * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciph
     */
    double Nr = N;
    double h = zMax / Nr;
    //System.out.println("h="+h);
    double sum = 0;
    for (int j = 0; j <= N; j++) {
        sum = sum + (h / 2) * (f(h * (j + 1)) + f(h * j));
        zVect[j] = sum + 0.5;
    }
}

//************** Method 2: algebraic fitting**********
//Returns an   algebraic approximation
//to the cumulative function of the
//standardized normal distribution.
private static double hFunction(double z, double k,
        double m, double b,
        double a, double e,
        int i, int l) {
    double p = 1;
    p = p / m;
    double hz = 0.5 + (k * z)
            / (2 * Math.pow(Math.pow((k * z), m) + b, p))
            + a * Math.pow(z, i) / (e + Math.pow(z, l));
    return hz;
}

//To use this method for another approximation,
//instead of  hFunction(z), write and make a call for your
//new proposal.
private static double fitting(double z, h hf) {
    double m = hf.m;
    double k = hf.k;
    double bb = hf.b;
    double a = hf.a;
    double e = hf.e;
    int i = hf.i;
    int l = hf.l;
    double f = hFunction(z, k, m, bb, a, e, i, l);
```

```
        return f;
    }

    //**************Comparison of two methods**************
    private static void title(long N) {
        System.out.println("cum = 0.5 + Area under "
                + " the standardized bell in within zero and z.");
        System.out.println("That area is calculated by "
                + "the Trapezoidal rule");
        System.out.println("with " + N + " divisions "
                + "in within 0 and 10.");
        System.out.println("A genetic algorithm calculates ");
        System.out.println("the optimal parameters of the hj family");
        System.out.println("to best fit cum.");
    }

    //The maximal error of approximation is found
    private static double findMaxError(double zMax, int N, h hf) {
        double maxError = 0;
        double error;
        double z;
        double Nr = N;
        double h = zMax / Nr;
        double hz;
        //System.out.println("h = " + h);
        for (int i = 0; i < N; i++) {
            z = i * h;
            hz = fitting(z, hf);
            error = zVect[i] - hz;
            error = Math.abs(error);
            if (error > maxError) {
                maxError = error;
            }
        }
        return maxError;
    }

    //*********************************
    //Genetic algorithm to optimize the
    //parameters of the hFamily.
    //We improve program B78, volume II, that finds the root
    //of a fifth degree polynomial.
    //*********************************
//*****************************************
//****************h CLASS ******************
//*****************************************
```

```
//This auxiliary class unites  fundamental
//parameters into an object.
//This encoding eases reuse of old code
    private static class h {

        int m;
        double k;
        double b;
        double a;
        double e;
        int i;
        int l;

        //Mandatory initialization
        h(int mm, double kk,
                double bb, double aa, double ee, int ii, int ll) {
            m = mm;
            k = kk;
            b = bb;
            a = aa;
            e = ee;
            i = ii;
            l = ll;
        }

        //Mandatory initialization
        h(h hf) {
            m = hf.m;
            k = hf.k;
            b = hf.b;
            a = hf.a;
            e = hf.e;
            i = hf.i;
            l = hf.l;
        }

        //Printing method.
        static void print(h hf) {
            System.out.print(" m  = " + hf.m);
            System.out.print(" k  = " + hf.k);
            System.out.print(" b  = " + hf.b);
            System.out.print(" a  = " + hf.a);
            System.out.print(" e  = " + hf.e);
            System.out.print(" i  = " + hf.i);
            System.out.print(" l  = " + hf.l);
        }
```

```
    }//end of class h

//  Turn on of the random generator
    static Random r = new Random();

    //This method generates random chars
    // that represent digits
    private static char randomDigit() {
        return (char) (r.nextInt(10) + 48);
    }

    //Generate a positive odd integer less than 9
    private static char generateOddM() {
        int m = r.nextInt(5);
        m = (2 * m + 49);
        char mChar = (char) m;
        if (test) {
            System.out.println("i = " + mChar);
        }
        return mChar;
    }

    //Generate positive even integer less than 9
    private static char generateEvenM() {
        int m = r.nextInt(4) + 1;
        m = (2 * m + 48);
        char mChar = (char) m;
        if (test) {
            System.out.println("Even power = " + mChar);
        }
        return mChar;
    }

    //Returns a string of digital numbers, a basic string
    private static String generateSubString(int nChars) {
        String w = "";
        //An individual is assembled char by char
        for (int j = 1; j <= nChars; j++) {
            // We generate a random numeric char
            char c = randomDigit();
            String s = "" + c;
            //System.out.println( "char = " + c);
            // Char c is added to w
            w = w + s;
            //System.out.println( "w = " + w);
```

```
        }
        return w;
    }

    //A power  is generated
    //Range runs form 0 to nChars
    private static String addPower(int nChars) {
        Integer i = r.nextInt(nChars);
        String s = i.toString();
        if (s.length() == 1) {
            s = '0' + s;
        }
        return s;
    }

    //A string encoding for a sign is generated
    private static String generateSign() {
        int spk = r.nextInt(2);
        char s = '+';
        if (spk == 0) {
            s = '-';
        }
        String t = "" + s;
        return t;
    }

    //A string encoding for a number is generated
    private static String generateString(int nChars) {
        String k = generateSubString(nChars);
        if (test) {
            System.out.println("  String = " + k);
        }
        String pk = addPower(nChars);
        //Powers run contrary wise
        String s = "" + generateSign();

        k = k + 'P' + s + pk;
        if (test) {
            System.out.println("Power  = " + pk);
            System.out.println("SignPower  = " + s);
            System.out.println("As String   = " + k);
        }
        return k;
    }

    //Declarations and default initializations
```

```
    // of other arrays.
    private static void otherInit() {
        Order = new int[limit + 1];
        ReportMin = new int[limit + 1];
        ReportMax = new int[limit + 1];
        Fitness = new double[limit + 1];
        for (int i = 0; i < nIndiv; i++) {
            Order[i] = 0;
            ReportMin[i] = 0;
            ReportMax[i] = 0;
            Fitness = new double[limit + 1];
        }
    }

    private static String generateIndividualh() {
        //m is generated
        if (test) {
            System.out.println("m is generated ");
        }
        char m = generateEvenM();

        //k is generated
        if (test) {
            System.out.println("k is generated");
        }
        String k = generateString(nChars);
        //b is generated
        if (test) {
            System.out.println("b is generated");
        }
        String bb = generateString(nChars);
        String ind = "" + m + k + bb;
        return ind;
    }

    private static String generateIndividualj() {
        //a is generated
        if (test) {
            System.out.println("a is generated");
        }
        String a = generateString(nChars);
        if (test) {
            System.out.println("e is generated");
        }
        String e = generateString(nChars);
        if (test) {
```

```
            System.out.println("i is generated");
        }
        char i = generateOddM();
        if (test) {
            System.out.println("l is generated");
        }
        char l = generateEvenM();
        String ind = "" + a + e + i + l;
        return ind;
    }

    /* We generate nIndiv individuals (strings)
encoding for three parameters.
All parameters are positive.
Sequences are  random */
    private static void Initialization() {
        //Formal declaration of our array.
        Individual = new String[limit];
        if (test) {
            System.out.println("ORIGINAL POPULATION");
        }
        for (int i = 0; i < nIndiv; i++) {
            if (test) {
                System.out.println("i = " + i);
            }
            Individual[i] = "";
            Individual[i] = generateIndividualh();
            Individual[i] = Individual[i] + generateIndividualj();

            if (test) {
                System.out.println("All = " + Individual[i]);
                System.out.println("length  = "
                        + Individual[i].length());
                System.out.println();
            }
        }
        otherInit();
    }

    //String s is decoded into a number
    private static double recoverNumber(String s) {
        // We get the sign of power of k
        // which could be + else -.
        char sp = s.charAt(nChars + 1);
        if (test) {
            System.out.println("Sign of power = " + sp);
```

```
        }
        // We get the   power of k
        // which ranges from 00 to 99.
        char s1 = s.charAt(nChars + 2);
        int p1 = Character.getNumericValue(s1);
        s1 = s.charAt(nChars + 3);
        int p2 = Character.getNumericValue(s1);
        int power = p1 * 10 + p2;
        if (sp == '-') {
            power = -power;
        }
        if (test) {
            System.out.println("Power = " + power);
        }

        //*********** We get the main string
        String k = s.substring(0, nChars);
        if (test) {
            System.out.println("as substring = " + k);
        }
        double number = 0;
        for (int j = 0; j < nChars; j++) {
            char s3 = k.charAt(j);
            if (s3 != '0') {
                int l = Character.getNumericValue(s3);
                int q = j + 1;
                double number1 = (int) l * Math.pow(10, power - q);
                number = number + number1;
            }
        }
        if (test) {
            System.out.println("as number = " + number);
        }
        return number;
    }

    // This method transforms a large string in  numbers,
    //which are all packed in hf
    private static h decoder(String s) {
        if (test) {
            System.out.println(" Individual =   = " + s);
            System.out.println("length of s  = " + s.length());
        }
        //We recover m:
        char ms = s.charAt(0);
        int m = Character.getNumericValue(ms);
```

```
if (test) {
    System.out.println("m = " + m);
}

//We get the string that encodes for k
String kString = s.substring(1, nChars + 5);
if (test) {
    System.out.println("kString = " + kString);
}
if (test) {
    System.out.println("Recovering k");
}
double k = recoverNumber(kString);
if (test) {
    System.out.println("k as number = " + k);
}

//We get the string that encodes for b
if (test) {
    System.out.println("Recovering b");
}
String bString = s.substring(nChars + 5, 2 * nChars + 9);
if (test) {
    System.out.println("bString = " + bString);
}
double bb = recoverNumber(bString);
if (test) {
    System.out.println("b as number = " + bb);
}

//We get the string that encodes for a
String aString = s.substring(2 * nChars + 9, 3 * nChars + 13);
if (test) {
    System.out.println("aString = " + aString);
}
if (test) {
    System.out.println("Recovering a");
}
double a = recoverNumber(aString);
if (test) {
    System.out.println("a as number = " + a);
}

//We get the string that encodes for e
if (test) {
    System.out.println("Recovering e");
```

```
        }
        String eString = s.substring(3 * nChars + 13, 4 * nChars + 17);
        if (test) {
            System.out.println("eString = " + eString);
        }
        double e = recoverNumber(eString);
        if (test) {
            System.out.println("e as number = " + e);
        }

        //We recover i:
        char mi = s.charAt(4 * nChars + 17);
        int i = Character.getNumericValue(mi);
        if (test) {
            System.out.println("i = " + i);
        }

        //We recover l:
        char ml = s.charAt(4 * nChars + 18);
        int l = Character.getNumericValue(ml);
        if (test) {
            System.out.println("l = " + l);
        }

        h hf = new h(m, k, bb, a, e, i, l);

        return hf;
    }

    private static void report() {
        h hf = decoder(Individual[Order[0]]);
        double error = findMaxError(zMax, N, hf);
        System.out.print("Error = " + error);
        h.print(hf);
        System.out.println();
    }

    //Individuals are sorted by fitting
    private static void Sorting(int gen) {
        int Champ;
        for (int i = 0; i < nIndiv; i++) {
            if (test) {
                System.out.println("i = " + i);
            }
            // Individual[i] is a string,
            // it is a genotype that encodes for hf;
```

```
            // hf is a set of parameters: it is the phenotype
            h hf = decoder(Individual[i]);
            double error = findMaxError(zMax, N, hf);
            if (test) {
                System.out.println("i = " + i
                        + " maxError  = " + error);
            }
            if (Math.abs(error) < 0.000000000001) {
                System.out.println("Gen = " + gen
                        + " Solution found");
                System.out.println("Error = " + error);
            }
            Fitness[i] = 1 / (1 + error * error);
            if (test) {
                System.out.println();
            }
        }
        //Sorting
        for (int i = 0; i < nIndiv; i++) {
            Champ = 0;
            for (int j = 0; j < nIndiv; j++) {
                if (Fitness[j] >= Fitness[Champ]) {
                    Champ = j;
                }
            }

            //The array Order classifies individuals by  fitness
            // by equal or decreasing order.
            Order[i] = Champ;
            Fitness[Champ] = 0;
            if (test) {
                System.out.println(i + "th ind. is  No "
                        + Champ);
            }
        }
    }

    //Each individual of the top ten
    // produces 10  copies.
    private static void Reproduction(int gen) {
        if (test) {
            System.out.println("Reproduction");
        }
        if (test) {
            System.out.println("The best = " + Order[0]);
        }
```

```
        Individualc = new String[limit];
        int counter = 0;
        for (int top = 0; top < 10; top++) {
            for (int j = 0; j < 10; j++) {
                Individualc[counter] = Individual[Order[top]];
                counter = counter + 1;
            }
        }/*
        for (int j = 0; j < counter; j++) {
            Individual[j] = Individualc[j];
        }
        */
        System.arraycopy(Individualc, 0, Individual, 0, counter);
    }

    //A mutation of ind j is done at place placeMut
    private static String mutateIndj(int j, int placeMut) {
        //test: placeMut = 4*nChars +18;
        if (test) {
            System.out.println("mutatedIndj placeMut = " + placeMut);
        }

        char c;
        String sign;
        String power;
        String s = Individual[j];
        //System.out.println(" indj         = " + s);
        if (test) {
            System.out.println(" indj         = " + s);
        }
        //***************Mutation in h******************
        //There are four cases: m,  sign of power, power,
        //or basic string.
        //Case m:
        if (placeMut == 0) {
            c = generateEvenM();
            s = c + s.substring(1);
            Individual[j] = s;
            if (test) {
                System.out.println(" m mutated    = " + s);
            }
        }
        //Case sign of power
        if (placeMut == nChars + 2) {
            sign = generateSign();
            s = s.substring(0, nChars + 2)
```

```
                + sign + s.substring(nChars + 3);
        if (test) {
            System.out.println(" Sign mutated = " + s);
        }
    }
    if (placeMut == 2 * nChars + 6) {
        sign = generateSign();
        s = s.substring(0, 2 * nChars + 6)
                + sign + s.substring(2 * nChars + 7);
        if (test) {
            System.out.println(" Sign mutated = " + s);
        }
    }

    //Case power:
    if ((placeMut == nChars + 3) | (placeMut == nChars + 4)) {
        power = addPower(nChars);
        s = s.substring(0, nChars + 3)
                + power + s.substring(nChars + 5);
        if (test) {
            System.out.println("Power mutated = " + s);
        }
    }
    if ((placeMut == 2 * nChars + 7)
            | (placeMut == 2 * nChars + 8)) {
        power = addPower(nChars);
        s = s.substring(0, 2 * nChars + 7)
                + power + s.substring(2 * nChars + 9);
        if (test) {
            System.out.println("Power mutated = " + s);
        }
    }

    //Case basic string
    if (((0 < placeMut) & (placeMut < nChars + 1))
            | ((nChars + 4 < placeMut)
            & (placeMut < 2 * nChars + 5))) {
        c = randomDigit();
        s = s.substring(0, placeMut)
                + c + s.substring(placeMut + 1);
        if (test) {
            System.out.println("Basic mutated = " + s);
        }
    }

    //**************Mutation in j***************
```

```
        //There are three cases, a or e, i, l.
        //Case a or e
//Case sign of power
        if (placeMut == 3 * nChars + 10) {
            sign = generateSign();
            s = s.substring(0, 3 * nChars + 10)
                    + sign + s.substring(3 * nChars + 11);
            if (test) {
                System.out.println(" a or e Sign mutated = " + s);
            }
        }
        if (placeMut == 4 * nChars + 14) {
            sign = generateSign();
            s = s.substring(0, 4 * nChars + 14)
                    + sign + s.substring(4 * nChars + 15);
            if (test) {
                System.out.println(" a or e Sign mutated = " + s);
            }
        }

        //Case power:
        if ((placeMut == 3 * nChars + 11)
                | (placeMut == 3 * nChars + 12)) {
            power = addPower(nChars);
            s = s.substring(0, 3 * nChars + 11)
                    + power + s.substring(3 * nChars + 13);
            if (test) {
                System.out.println(" a or e Power mutated = " + s);
            }
        }
        if ((placeMut == 4 * nChars + 15)
                | (placeMut == 4 * nChars + 16)) {
            power = addPower(nChars);
            s = s.substring(0, 4 * nChars + 15)
                    + power + s.substring(4 * nChars + 17);
            if (test) {
                System.out.println("a or e Power mutated = " + s);
            }
        }

        //Case basic string
        if (((2 * nChars + 8 < placeMut)
                & (placeMut < 3 * nChars + 10))
                | ((3 * nChars + 12 < placeMut)
                & (placeMut < 4 * nChars + 13))) {
            c = randomDigit();
```

```
            s = s.substring(0, placeMut)
                    + c + s.substring(placeMut + 1);
            if (test) {
                System.out.println(" a or e Basic mutated = " + s);
            }
        }

        //Case i
        if (placeMut == 4 * nChars + 17) {
            char ic = generateOddM();
            s = s.substring(0, 4 * nChars + 17)
                    + ic + s.substring(4 * nChars + 18);
            if (test) {
                System.out.println(" i mutated =    " + s);
            }
        }

        //Case l
        if (placeMut == 4 * nChars + 18) {
            char lc = generateEvenM();
            s = s.substring(0, 4 * nChars + 18)
                    + lc + s.substring(4 * nChars + 19);
            if (test) {
                System.out.println(" l mutated =    " + s);
            }
        }
        if (test)
            System.out.println(" s            = " + s);
        return s;
    }

    //We consider only one type of mutation:
    //substitution of one char by another one.
    //The first copy of the winner does not mutate.
    private static void Mutation() {
        Individualc[0] = Individual[0];
        for (int j = 1; j < nIndiv; j++) {
            //Define place of mutation
            int n = Individual[j].length();
            int placeMut = r.nextInt(n);
            //Will that place mutate?
            double p = r.nextDouble();
            if (p < mutationRate) {
                Individualc[j] = mutateIndj(j, placeMut);
            }
        }
```

```
        //Some individuals are generated ab initio.
        //We hope that they will help to escape
        //from local optimal states.
        int initial = nIndiv - 20;
        if (initial > 0) {
            for (int i = nIndiv; i < nIndiv; i++) {
                if (test) {
                    System.out.println("i = " + i);
                }
                Individualc[i] = "";
                Individualc[i] = generateIndividualh()
                        +  generateIndividualj();
            }
        }
        /*
            for(int j = 0; j< nIndiv; j++)
            Individual[j] = Individualc[j];
         */
        System.arraycopy(Individualc, 0, Individual, 0, nIndiv);
    }

    //Two strings recombine and produce two offspring.
    private static void RecombinationOld() {
        int l = Individual[0].length();
        for (int j = 30; j < nIndiv; j++) {
            //Define individuals to recombine
            int m = r.nextInt(nIndiv);
            int n = r.nextInt(nIndiv);
            String a = Individual[m];
            String bb = Individual[n];
            System.out.println(a);
            System.out.println(bb);
            //Define place of recombination
            int placeRec = r.nextInt(l);
            if (m != 0) {
                Individual[m] = a.substring(0, placeRec)
                        + bb.substring(placeRec);
            }
            if (n != 0) {
                Individual[n] = bb.substring(0, placeRec)
                        + a.substring(placeRec);
            }
        }
    }

    //Two strings recombine and produce two offspring.
```

```
    //Corrected
    private static void Recombination() {
        int l = Individual[0].length();
        /*
            for(int j = 0; j< nIndiv; j++)
            Individualc[j] = Individual[j];
        */
        System.arraycopy(Individual, 0, Individualc, 0, nIndiv);
        for (int j = 0; j < nIndiv; j++) {
            //Define place of mutation
            int m = r.nextInt(nIndiv);
            int n = r.nextInt(nIndiv);
            String a = Individual[m];
            String bb = Individual[n];

            int placeRec = r.nextInt(l);
            Individualc[m] = a.substring(0, placeRec)
                    + bb.substring(placeRec);
            Individualc[n] = bb.substring(0, placeRec)
                    + a.substring(placeRec);
            if (test)
            {
                System.out.println("a0 = " + a);
                System.out.println("b0 = " + bb);
                System.out.println("a2 = " + Individualc[m]);
                System.out.println("b2 = " + Individualc[n]);
            }
        }
        /*
            for(int j = 0; j< nIndiv; j++)
            Individual[j] = Individualc[j];
        */
        System.arraycopy(Individualc, 0, Individual, 0, nIndiv);

    }

    //Mutation rate can increase
    private static void mutationRedef() {
        String c = Individual[0]; //champ reported
        h hf = decoder(c);
        double error = findMaxError(zMax, N, hf);
        /* System.out.println("gen " + gen +
  " Error " + error);*/
 /* System.out.println("mutation rate = " +
    mutationRate);*/
        newError = error;
```

```
        deltaError = Math.abs(newError - oldError);
        /* System.out.println("Error= " + error +
  " deltaError= " + deltaError); */
        oldError = newError;
        if (deltaError < 0.001) {
            mutationRate = 1.1 * mutationRate;
            if (mutationRate > 1) {
                mutationRate = 1;
            }

            /*  System.out.println("gen " + gen + "
    mutRedef");*/
        } else;
    }

    //Overall method
    private static void dynamics(int gen) {
        //Individuals are sorted by fitness
        Sorting(gen);
        //for(int i = 1; i<= 10;i++)
        //System.out.println(  Individual(Order(i))
        report();
        //The top ten are preferentially reproduced
        Reproduction(gen);
        //The new population is subjected to mutation
        Mutation();
        Recombination();
        //every 100 generations, stagnation is revised
        if (gen % 100 == 0) {
            mutationRedef();
        }
    }

    //Decodes a string into a number
    private static void decode() {
        System.out.println();
        System.out.println("Numeric values at decoding");
        for (int i = 0; i < nIndiv; i++) {
            System.out.println("i = " + i);
            System.out.println("All = " + Individual[i]);
            decoder(Individual[i]);
        }
    }

    //test of encoding and decoding
    private static void test() {
```

```
        nIndiv = 1;
        System.out.println("Strings at Initialization");
        Initialization();
        decode();
    }

    public static void main(String[] args) {
        test = false; //else true
        if (test) {
            test();
        } else {
            test = false;
            //The whole cumulative function is estimated
            //by the trapezoidal rule
            trapRule();

            //The parameters in the hFamily are optimized
            //by a genetic algorithm.
            System.out.println("Running ");
            nIndiv = 150;
            Initialization();
            nGen = 30000;
            oldError = 1000;
            mutationRate = 1;
            for (int gen = 0; gen < nGen; gen++) {
                System.out.println("Gen = " + gen);
                dynamics(gen);
            }
        }
    }
}//End of Program E271 HjFamily
```

272. ***Exercise.*** *Run the previous program various times in parallel. Verify that the new code diminishes the error to roughly half of that found by the paternal code after some 200 generations with 100000 individuals.*

273. ***Alarm.*** *If one improves a code and as result the error diminishes to a half, How many improvements do we need to diminish the error from circa one to* 10^{-9}*? We need around 28 improvements because* $2^{-28} = 3 \times 10^{-9}$.

This is very onerous. Therefore, we must think finer and better.

274. ***Trying another modification***

We have been considering the family

$$hj_{m,k,b,a,e,i,l}(z) = \frac{kz}{2\sqrt[m]{(kz)^m + b}} + \frac{az^i}{e + z^l}$$

The optimal member has $m = 4$, $i = 1$, $l = 4$:

$$hj(z) = \frac{kz}{2\sqrt[4]{(kz)^4+b}} + \frac{az}{e+z^4}$$

We can rewrite this function as follows:

$$hj(z) = \frac{1}{2\sqrt[4]{1+\frac{a}{z^4}}} + \frac{1}{\frac{b}{z}+cz^3}$$

where coefficients a, b, c are real and must be determined. We can generalize this expression into the next family:

$$hc(z) = \frac{1}{2\sqrt[m]{1+\frac{a_{-2}}{z^2}+\frac{a_{-4}}{z^4}+....}} + \frac{1}{....+\frac{c_{-5}}{z^5}+\frac{c_{-3}}{z^3}+\frac{c_{-1}}{z}+c_1z^1+c_3z^3+c_5z^5+....}$$

This new family has the virtue of including higher corrections terms that will modify the shape of our approximation function and that eventually could fit better the error of approximation. Is that true?

275. *The next code test the* hc *family.* *With* $nTermsRoot = 2$ *and* $nTermsSecond = 2$*, this program estimates the best parameters for the next family:*

$$hc(z) = \frac{1}{2\sqrt[m]{1+\frac{a_{-2}}{z^2}+\frac{a_{-4}}{z^4}+....}} + \frac{1}{....+\frac{c_{-5}}{z^5}+\frac{c_{-3}}{z^3}+\frac{c_{-1}}{z}+c_1z^1+c_3z^3+c_5z^5+....}$$

```
//Program E275 HcFamily
//Computes the  integral under the standard Gauss bell
//in within 0 and a positive z.
//Method 1: trapezoidal rule.
//Method 2: algebraic approximation given by
//hc(z) = 1/( 2 sqrt{  1 +  a_{-2}/z^{2} +
//                       a_{-4}/ z^{4}+ .. ..})
//
//+  1 /(.. .. +   c_{-5}/z^5   +  c_{-3}/z^3  +
//c_{-1}/z} + c_1z^{1} + c_3z^{3} + c_5z^{5}+.. ..  ) .
//This is the HcFamily.
//We consider three first level parameters:
//M the order of the root in the first fraction.
//n the number of terms  under the root
//i the number of terms in the second fraction.
//Optimal parameters   are found by
//a genetic algorithm.
//The maximal approximation error is reported.

//The genetic algorithm runs over translapped generations,
```

```
//i.e., the new and old generations mix one with another.

package ejvol5v3p;


import java.util.Random;

public class HcFamily {

    private static double pi = 3.14159265358979323846;
    //Values of the true cumulative function
    private static double zVect[] = new double[10000001];
    //Calculations are done for z in (0, zMax)
    private static double zMax = 10;
    //Number of sample points
    private static int N = 100000;
    //Max power of ten of numbers
    private static int maxPower;

    //****************Genetic part***********
    // Individuals are kept in the array
    // Individual[]. It is an array of strings.
    //Each individual encodes three numbers
    //corresponding to the 7 parameters of the
    //jFamily.
    // The number of individuals must be
    // less than limit.
    static int limit = 50000;
    static double Fitness[];
    static String Individual[], Individualc[];
    //Actual number of individuals
    static int nIndiv;
    //Number of chars per parameter
    static int nChars = 15;
    static int Order[];
    //static String b;
    static int generation;
    static int nGen;
    static int ReportMin[], ReportMax[];
    //Mutation rate per site
    static double mutationRate;
    static double oldError, newError;
    static double deltaError;
    static boolean test;
    //number of terms under the root
    static int nTermsRoot;
```

```
    //number of terms in the second fraction
    static int nTermsSecond;
    static boolean inducedInit;

    // Mandatory initialization
    private static void initialize(long N) {
        for (int i = 0; i <= N; i++) {
            zVect[i] = 0.5;
        }
    }

    //*********Method 1: Trapezoidal rule*****************
    //Returns the density function of the standardized
    //normal distribution.
    private static double zFunction(double z) {
        double f = 1 / Math.pow(2 * pi, 0.5) * Math.exp(-z * z / 2);
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double z) {
        double f = zFunction(z);
        return f;
    }

    private static void trapRule() {
        initialize(N);
        title(N);
        //Trapezoidal rule for the cumulative function
        zHalfBodyTrapReuse(zMax, N);
    }

    //Returns 0.5 + the integral under the z-density function
    //in within 0 and z. Precision=1/N.
    //Trapezoidal rule adapted for reuse.
    private static void zHalfBodyTrapReuse(double zMax,
            long N) {
        /**
         * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciphers.
         */
        double Nr = N;
        double h = zMax / Nr;
        //System.out.println("h="+h);
        double sum = 0;
```

```
        for (int j = 1; j <= N; j++) {
            sum = sum + (h / 2) * (f(h * (j + 1)) + f(h * j));
            zVect[j] = sum + 0.5;
        }
    }

    //************** Method 2: algebraic fitting**********
    //Returns an   algebraic approximation
    //to the cumulative function of the
    //standardized normal distribution.
    private static double hcFunction(double z, h hf) {
        double p1 = 1;
        double p = p1 / hf.m;
        double hc = 0.5;
        double firstTerm = 1;
        for (int j = 0; j < nTermsRoot; j++) {
            firstTerm = firstTerm + hf.a[j] / (Math.pow(z, 2 * (j + 1)));
        }
        if (firstTerm >= 0) {
            hc = hc + 1 / (2 * Math.pow(firstTerm, p));
        } else {
            hc = 100;
        }
        double secondTerm = 0;
        for (int j = 0; j < nTermsSecond; j++) {
            secondTerm = secondTerm + hf.nc[j] / (Math.pow(z, 2 * j + 1))
                    + hf.c[j] * (Math.pow(z, 2 * j + 1));
        }
        secondTerm = p1 / secondTerm;
        hc = hc + secondTerm;
        return hc;
    }

    //To use this method for another approximation,
    //instead of  hFunction(z), write and make a call for your
    //new proposal.
    private static double fitting(double z, h hf) {
        double f = hcFunction(z, hf);
        return f;
    }

    //**************Comparison of two methods**************
    private static void title(long N) {
        System.out.println("cum = 0.5 + Area under "
                + " the standardized bell in within zero and z.");
        System.out.println("That area is calculated by "
```

```
                + "the Trapezoidal rule");
        System.out.println("with " + N + " divisions "
                + "in within 0 and 10.");
        System.out.println("A genetic algorithm calculates ");
        System.out.println("the optimal parameters of the hc family");
        System.out.println("to best fit cum.");
    }

    //The maximal error of approximation is found
    private static double findMaxError(double zMax, int N, h hf) {
        zHalfBodyTrapReuse(zMax, N);
        double maxError = 0;
        double error;
        double z;
        double Nr = N;
        double h = zMax / Nr;
        double hz;
        //System.out.println("h = " + h);
        for (int i = 1; i < N; i++) {
            z = i * h;
            hz = fitting(z, hf);
            error = zVect[i] - hz;
            error = Math.abs(error);
            if (error > maxError) {
                maxError = error;
            }
            /*System.out.print("i = " + i + " zVect = " + zVect[i]);
  System.out.println( " hz= " + hz + " error = " + error);*/
        }
        return maxError;
    }

    //*********************************
    //Genetic algorithm to optimize the
    //parameters of the hFamily.
    //We improve program B78, volume II, that finds the root
    //of a fifth degree polynomial.
    //*********************************
//*****************************************
//****************h CLASS ******************
//*****************************************
//This auxiliary class unites  fundamental
//parameters into an object.
//This encoding eases reuse of old code
    private static class h {
        //root
```

```
int m;
//number of terms under the root
double a[] = new double[50];
//Coefficients of terms with positive power,
//second term
double c[] = new double[50];
//Coefficients of terms with negative power
//second term
double nc[] = new double[50];

//Mandatory initialization
//Mandatory initialization
h() {
    m = 4;
    for (int j = 0; j < nTermsRoot; j++) {
        a[j] = 0;
    }
    for (int j = 0; j < nTermsSecond; j++) {
        c[j] = 0;
        nc[j] = 0;
    }
}

h(int mm, int nn,
        int ii, double aa[], double cc[], double ncc[]) {
    m = mm;
    /*
    for (int j = 0; j < nTermsRoot; j++) {
    a[j] = aa[j];
    }
     */
    System.arraycopy(aa, 0, a, 0, nTermsRoot);
    for (int j = 0; j < nTermsSecond; j++) {
        c[j] = cc[j];
        nc[j] = ncc[j];
    }
}

//Mandatory initialization
h(h hf) {
    m = hf.m;
    /*
    for (int j = 0; j < nTermsRoot; j++) {
    a[j] = hf.a[j];
    }
```

```
             */
            System.arraycopy(hf.a, 0, a, 0, nTermsRoot);
            for (int j = 0; j < nTermsSecond; j++) {
                c[j] = hf.c[j];
                nc[j] = hf.c[j];
            }
        }

        //Printing method.
        static void print(h hf) {
            System.out.print(" m  = " + hf.m + ";");
            System.out.print(" a[] : ");
            for (int j = 0; j < nTermsRoot; j++) {
                System.out.print("  j = " + j + "   " + hf.a[j]);
            }
            System.out.print("; nc[] : ");
            for (int j = 0; j < nTermsSecond; j++) {
                System.out.print("  j = " + j + "   " + hf.nc[j]);
            }
            System.out.print("; c[] : ");
            for (int j = 0; j < nTermsSecond; j++) {
                System.out.print("  j = " + j + "   " + hf.c[j]);
            }
        }

    }//end of class h

//  Turn on of the random generator
    static Random r = new Random();

    //This method generates random chars
    // that represent digits
    private static char randomDigit() {
        return (char) (r.nextInt(10) + 48);
    }

    //Generate a positive odd integer less than 9
    private static char generateOddM() {
        int m = r.nextInt(5);
        m = (2 * m + 49);
        char mChar = (char) m;
        if (test) {
            System.out.println("i = " + mChar);
        }
        return mChar;
    }
```

```
//Generate positive even integer less than 9
private static char generateEvenM() {
    int m = r.nextInt(4) + 1;
    m = (2 * m + 48);
    char mChar = (char) m;
    return mChar;
}

//Returns a string of digital numbers, a basic string
private static String generateSubString(int nChars) {
    String w = "";
    //An individual is assembled char by char
    for (int j = 1; j <= nChars; j++) {
        // We generate a random numeric char
        char c = randomDigit();
        String s = "" + c;
        //System.out.println( "char = " + c);
        // Char c is added to w
        w = w + s;
        //System.out.println( "w = " + w);
    }
    return w;
}

//A power  is generated
//Range runs form 0 to nChars
private static String addPower(int n) {
    Integer i = r.nextInt(n);
    String s = i.toString();
    if (s.length() == 1) {
        s = '0' + s;
    }
    return s;
}

//A string encoding for a sign is generated
private static String generateSign() {
    char s = '+';
    int spk = r.nextInt(2);
    if (spk == 0) {
        s = '-';
    }
    String t = "" + s;
    return t;
}
```

```
//A string encoding for a number is generated.
//a sign, a substring, a power with its sign.
private static String generateString(int nChars) {
    String sign = "" + generateSign();
    String ms = generateSubString(nChars);
    String signOfPower = "" + generateSign();
    String pk = addPower(maxPower);
    //Powers run contrary wise

    String k = sign + ms + 'P' + signOfPower + pk;
    if (test) {
        System.out.println("Sign  = " + sign);
        System.out.println("Main string  = " + ms);
        System.out.println("SignPower  = " + signOfPower);
        System.out.println("Power  = " + pk);
    }
    return k;
}

//Declarations and default initializations
// of other arrays.
private static void otherInit() {
    Order = new int[limit + 1];
    ReportMin = new int[limit + 1];
    ReportMax = new int[limit + 1];
    Fitness = new double[limit + 1];
    for (int i = 0; i < nIndiv; i++) {
        Order[i] = 0;
        ReportMin[i] = 0;
        ReportMax[i] = 0;
        Fitness = new double[limit + 1];
    }
}

private static String generateIndividual() {
    //m is generated
    char m = generateEvenM();
    String ind = "" + m;
    if (test) {
        System.out.println("m = " + m);
    }
    //the a's are  generated
    if (test) {
        System.out.println("\r The a's are generated");
    }
```

```
        for (int j = 0; j < nTermsRoot; j++) {
            String a = generateString(nChars);
            ind = ind + a;
            if (test) {
                System.out.println("j = " + j + " a = " + a);
            }
        }
        //the nc's are  generated
        if (test) {
            System.out.println("\r The nc's are generated");
        }
        for (int j = 0; j < nTermsSecond; j++) {
            String nc = generateString(nChars);
            ind = ind + nc;
            if (test) {
                System.out.println("j = " + j + " nc = " + nc);
            }
        }
        //the c's are  generated
        if (test) {
            System.out.println("\r The c's are generated");
        }
        for (int j = 0; j < nTermsSecond; j++) {
            String c = generateString(nChars);
            ind = ind + c;
            if (test) {
                System.out.println("j = " + j + " c = " + c);
            }
        }
        return ind;
    }

    private static String forcedInd() {
        //m is generated
        char m = '4';
        String ind = "" + m;
        //Coefficients are set to 1
        String a = "+100000000000000P+01";
        for (int i = 0; i < nTermsRoot + 2 * nTermsSecond; i++) {
            ind = ind + a;
        }
        return ind;
    }

    /* We generate nIndiv individuals (strings)
encoding for three parameters.
```

```
All parameters are positive.
Sequences are  random */
   private static void Initialization() {
       //Formal declaration of our array.
       Individual = new String[limit];
       if (test) {
           System.out.println("ORIGINAL POPULATION");
       }

       for (int i = 0; i < nIndiv; i++) {
           if (test) {
               System.out.println(" \r individual  = " + i);
           }
           Individual[i] = "";

           //Some individuals are given by the developer
           if (inducedInit & (i < 10)) {
               Individual[i] = forcedInd();
           } else {
               Individual[i] = generateIndividual();
           }
           if (test) {
               System.out.println("All at initialization = "
                       + Individual[i]);

               System.out.println("length  = "
                       + Individual[i].length());
               System.out.println();
           }
       }

       otherInit();
   }

   //A string encoding for a number is generated.
   //a sign, a substring, a power with its sign.
   private static double recoverNumber(String s) {
       //sign is recovered
       char sign = s.charAt(0);
       //sign is deleted
       s = s.substring(1);

       //main string is recovered
       String ms = s.substring(0, nChars);
       //main string is deleted
       s = s.substring(nChars);
```

```
//'P' is deleted
s = s.substring(1);
//Sign of power is recovered
char signOfPower = s.charAt(0);
//Sign is deleted
s = s.substring(1);
//Power is recovered
char c1 = s.charAt(0);
char c2 = s.charAt(1);
String pk = "" + c1 + c2;
int p1 = Character.getNumericValue(c1);
int p2 = Character.getNumericValue(c2);
int power = p1 * 10 + p2;
if (signOfPower == '-') {
    power = -power;
}

String k = sign + ms + 'P' + signOfPower + pk;

if (test) {
    System.out.println("as substring = " + k);
}
double number = 0;
for (int j = 0; j < nChars; j++) {
    char s3 = ms.charAt(j);
    if (s3 != '0') {
        int l = Character.getNumericValue(s3);
        int q = j + 1;
        double number1 = (int) l * Math.pow(10, power - q);
        number = number + number1;
    }
}
if (sign == '-') {
    number = -number;
}
if (test) {
    System.out.println("as number = " + number);
}

if (test) {
    System.out.println("Sign  = " + sign);
    System.out.println("Main string  = " + ms);
    System.out.println("SignOfPower  = " + signOfPower);
    System.out.println("Power as string = " + pk);
    System.out.println("Power as number = " + power);
    System.out.println("Final string = " + k);
```

```
            System.out.println("Final number = " + number);
        }
        return number;
    }

    // This method transforms a large string in  numbers,
    //which are all packed in hf
    private static h decoder(String s) {
        h hf = new h();
        if (test) {
            System.out.println("All    at   decoder  =  " + s);
            System.out.println(" length of s  = " + s.length());
        }
        //m is recovered
        char ms = s.charAt(0);
        int m = Character.getNumericValue(ms);
        if (test) {
            System.out.println("m = " + m);
        }
        //m is deleted
        s = s.substring(1);

        //the a's are  recovered
        if (test) {
            System.out.println("\r The a's are recovered");
        }
        for (int j = 0; j < nTermsRoot; j++) {
            if (test) {
                System.out.println("j = " + j);
            }
            hf.a[j] = recoverNumber(s);
            if (test) {
                System.out.println(" a = " + hf.a[j]);
            }
            //a is deleted
            s = s.substring(nChars + 5);
        }

        //the nc's are  recovered
        if (test) {
            System.out.println("\r The nc's are recovered");
        }
        for (int j = 0; j < nTermsSecond; j++) {
            if (test) {
                System.out.println("j = " + j);
            }
```

```
            hf.nc[j] = recoverNumber(s);
            if (test) {
                System.out.println(" nc = " + hf.nc[j]);
            }
            //a is deleted
            s = s.substring(nChars + 5);
        }

        //the c's are  recovered
        if (test) {
            System.out.println("\r The c's are recovered");
        }
        for (int j = 0; j < nTermsSecond; j++) {
            if (test) {
                System.out.println("j = " + j);
            }
            hf.c[j] = recoverNumber(s);
            if (test) {
                System.out.println(" c = " + hf.c[j]);
            }
            //a is deleted
            s = s.substring(nChars + 5);
        }
        return hf;
    }

    //Individuals are sorted by fitting
    private static void Sorting(int gen) {
        int Champ;
        for (int i = 0; i < nIndiv; i++) {
            if (test) {
                System.out.println("ind = " + i);
            }
            // Individual[i] is a string,
            // it is a genotype that encodes for hf;
            // hf is a set of parameters: it is the phenotype
            h hf = decoder(Individual[i]);
            double error = findMaxError(zMax, N, hf);
            if (test) {
                System.out.println("i = " + i
                        + " maxError  = " + error);
            }
            if (Math.abs(error) < 0.000000000001) {
                System.out.println(" Gen = " + gen
                        + " Solution found");
                System.out.println(" Error = " + error);
```

```
                h.print(hf);
            }
            Fitness[i] = 1 / (1 + error * error);
            if (test) {
                System.out.println();
            }
        }
        //Sorting
        for (int i = 0; i < nIndiv; i++) {
            Champ = 0;
            for (int j = 0; j < nIndiv; j++) {
                if (Fitness[j] >= Fitness[Champ]) {
                    Champ = j;
                }
            }
            //The array Order classifies individuals by  fitness
            // by equal or decreasing order.
            Order[i] = Champ;
            Fitness[Champ] = 0;
            if (test) {
                System.out.println(i + "th ind. is  No "
                        + Champ);
            }
        }
    }

    //champ reported
    private static void report() {
        String c = Individual[Order[0]];
        h hf = new h(decoder(c));
        double error = findMaxError(zMax, N, hf);
        System.out.println(" Genotype = " + c);
        h.print(hf);
        System.out.println("\n Error = " + error);
    }

    //Each individual of the top ten
    // produces 10  copies.
    private static void Reproduction(int gen) {
        if (test) {
            System.out.println("Reproduction");
        }
        if (test) {
            System.out.println("The best = " + Order[0]);
        }
        Individualc = new String[limit];
```

```
        int counter = 0;
        for (int top = 0; top < 10; top++) {
            for (int j = 0; j < 10; j++) {
                Individualc[counter] = Individual[Order[top]];
                counter = counter + 1;
            }
        }
        /*
        for (int j = 0; j < counter; j++) {
        Individual[j] = Individualc[j];
        }
         */
        System.arraycopy(Individualc, 0, Individual, 0, counter);
    }

    private static String mutate(int k, int place) {
        String s = Individual[k];
        //place = 18;
        if (test) {
            System.out.println(" indj         = " + s);
            System.out.println("placeMut = " + place);
        }
        //We go over the same steps of decoding
        //case m
        if (place == 0) {
            char c = generateEvenM();
            s = c + s.substring(1);
            if (test) {
                System.out.println(" m mutated    = " + s);
            }
        }

        //The a's, nc's and c's all have the same encoding
        //Case sign of number
        if (place % (nChars + 5) == 1) {
            String sign = generateSign();
            s = s.substring(0, place)
                    + sign + s.substring(place + 1);
            if (test) {
                System.out.println(" SSSSSSSSSSSSSSSign = "
                        + s.charAt(place));
                System.out.println(" sign mutated    = " + s);
            }
        }

        //Case main string
```

```
        if (1 < (place % (nChars + 5))
                & (place % (nChars + 5) < nChars + 1)) {
            char c = randomDigit();
            s = s.substring(0, place)
                    + c + s.substring(place + 1);
            if (test) {
                System.out.println("Basic mutated = " + s);
            }
        }

        //'P' is not a case for mutation
        //case sign of power
        if (place % (nChars + 5) == nChars + 3) {
            String sign = generateSign();
            s = s.substring(0, place)
                    + sign + s.substring(place + 1);
            if (test) {
                System.out.println(" Sign of power = "
                        + s.charAt(place));
                //System.out.println (" sign mutated    = " + s);
            }
        }
        //Case power:
        if (place % (nChars + 5) == nChars + 4) {
            String power = addPower(maxPower);
            s = s.substring(0, place)
                    + power + s.substring(place + 2);
            if (test) {
                System.out.println("Power mutated = " + s);
            }
        }
        if (place % (nChars + 5) == nChars + 5) {
            String power = addPower(maxPower);
            s = s.substring(0, place - 1)
                    + power + s.substring(place + 1);
            if (test) {
                System.out.println("Power mutated = " + s);
            }

        }
        return s;
    }

    //Individual j undergoes site by site mutation.
    //New and old generations are mixed
    private static void mutateIndj(int j) {
```

```
        String s = Individual[j];

        int length = s.length();
        for (int place = 0; place < length; place++) {
            double q = r.nextDouble();
            if (q < mutationRate) {
                Individual[j] = mutate(j, place);
            }
        }
    }

    //We consider only one type of mutation:
    //substitution of one char by another one.
    //The first copy of the winner does not mutate.
    private static void Mutation() {
        for (int j = 1; j < nIndiv; j++) {
            mutateIndj(j);
        }
        //Some individuals are generated ab initio.
        //We hope that they will help to a rapid escape
        //from local optims.
        int initial = 15;
        if (initial > 0) {
            for (int i = initial; i < nIndiv; i++) {
                if (test) {
                    System.out.println("i = " + i);
                }
                Individual[i] = "";
                Individual[i] = generateIndividual();
            }
        }
    }

    //Two strings recombine and produce two offspring.
    private static void Recombination() {
        int l = Individual[0].length();
        for (int j = 10; j < nIndiv; j++) {
            int m;
            int n;
            m = r.nextInt(nIndiv);
            n = r.nextInt(nIndiv);
            String a = Individual[m];
            String b = Individual[n];
            //Define place of recombination
            int placeRec = r.nextInt(l);
            //the zeroth individual is kept intact.
```

```
            if (m != 0) {
                Individual[m] = a.substring(0, placeRec)
                        + b.substring(placeRec);
            }
            if (n != 0) {
                Individual[n] = b.substring(0, placeRec)
                        + a.substring(placeRec);
            }
        }
    }

    //Mutation rate can increase
    private static void mutationRedef() {
        String c = Individual[0]; //champ reported
        h hf = decoder(c);
        double error = findMaxError(zMax, N, hf);
        /* System.out.println("gen " + gen +
  " Error " + error);*/
 /* System.out.println("mutation rate = " +
    mutationRate);*/
        newError = error;
        deltaError = Math.abs(newError - oldError);
        /* System.out.println("Error= " + error +
  " deltaError= " + deltaError); */
        oldError = newError;
        if (deltaError < 0.001) {
            mutationRate = 1.1 * mutationRate;
            if (mutationRate > 1) {
                mutationRate = 1;
            }

            /*  System.out.println("gen " + gen + "
    mutRedef");*/
        } else;
    }

    //Overall method
    private static void dynamics(int gen) {
        //Individuals are sorted by fitness
        Sorting(gen);
        //for(int i = 1; i<= 10;i++)
        //System.out.println(  Individual(Order(i))
        report();
        //The top ten are preferentially reproduced
        Reproduction(gen);
        //The new population is subjected to mutation
```

```
        Mutation();
        Recombination();
        //every 100 generations, stagnation is revised
        if (gen % 100 == 0) {
            mutationRedef();
        }
    }

    //Decodes a string into a number
    private static void decode() {
        System.out.println();
        System.out.println("Numeric values at decoding");
        for (int i = 0; i < nIndiv; i++) {
            System.out.println("i = " + i);
            decoder(Individual[i]);
        }
    }

    //test of encoding and decoding
    private static void test() {
        nIndiv = 1;
        System.out.println("Strings at Initialization");
        Initialization();
        decode();
    }

    public static void main(String[] args) {
        nTermsRoot = 2;
        nTermsSecond = 2;
        inducedInit = true;
        nIndiv = 25;
        maxPower = 6;
        Initialization();
        System.out.println("nTermsRoot =  " + nTermsRoot);
        System.out.println("nTermsRoot =  " + nTermsSecond);
        nGen = 100000;
        oldError = 1000;
        //Mutation rate per site
        mutationRate = 0.3;
        test = false; //else true
        if (test) {
            test();
        } else {
            //test = true;
            //The whole cumulative function is estimated
            //by the trapezoidal rule
```

```
            trapRule();
            //The parameters in the hFamily are optimized
            //by a genetic algorithm.
            System.out.println("Running ");
            //Random else induced initialization

            for (int gen = 1; gen <= nGen; gen++) {
                System.out.println("Gen = " + gen);
                dynamics(gen);
            }
        }
    }
}//End of Program E275 HcFamily
```

276. ***Exercise.*** *Run the code and verify else reject what the author found: with the given parameters, the program arrives in some 200 generations to what seems to be its best performance, which by no means is better than that of the previous program.*

277. ***Exercise.*** *Study the effect of adding more corrections terms to the hc family. For instance, with $nTermsRoot = 2$ and $nTermsSecond = 3$, one estimates the best parameters for the next family:*

$$hc(z) = \frac{1}{2\sqrt[4]{1+\frac{a[0]}{z^2}+\frac{a[1]}{z^4}}} + \frac{1}{\frac{nc[0]}{z}+c[0]z^1+\frac{nc[1]}{z^3}+c[1]z^3+\frac{nc[2]}{z^5}+c[2]z^5}$$ *Answer*

278. ***Exercise.*** *Modify the code of the previous program to report the waiting time T that measures the number of generations that the program needs to improve a solution. Verify else reject the thesis that $T(j)$, the waiting time for improvement j, is an exponential function of j. Define progress, g, as the marginal fitness: $f(j+1)-f(j)$, and optimization velocity, v, as g/T, the ratio of progress/waiting time. Verify else reject the thesis that v is a negative exponential function of the order of appearance of improvements.* *Answer*

279. ***Exercise*** *Study the effect of modifying the architecture of the fitness function. For instance, slight modifications of that function would allow one to study the following family:*

$$hcu(z) = \frac{1+\frac{1}{\frac{nc[0]}{z}+c[0]z^1+\frac{nc[1]}{z^3}+c[1]z^3+\frac{nc[2]}{z^5}+c[2]z^5}}{2\sqrt[4]{1+\frac{a[0]}{z^2}+\frac{a[1]}{z^4}}}$$ *Answer*

280. ***Exercise*** *Study the following family:*

$$hcn(z) = \frac{1}{2\sqrt[4]{1+\frac{a[0]}{z^2}+\frac{a[1]}{z^4}}+\frac{1}{\frac{nc[0]}{z}+c[0]z^1+\frac{nc[1]}{z^3}+c[1]z^3+\frac{nc[2]}{z^5}+c[2]z^5}}$$ *Answer*

281. ***Exercise*** *Study the following family:*

$$hcn(z) = \frac{1/2}{\sqrt[4]{1+\frac{a[0]}{z^2}+\frac{a[1]}{z^4}}+\frac{1}{\frac{nc[0]}{z}+c[0]z^1+\frac{nc[1]}{z^3}+c[1]z^3+\frac{nc[2]}{z^5}+c[2]z^5}}$$ *Answer*

282. ***Exercise.*** *The web contains many approximations to the cumulative normal distribution. One very simple was proposed by Bowling et al. ([10] 2009) :*

$$F(z) \approx \frac{1}{1+e^{-(1.5976*z+0.07056*z^3)}}$$

This approximation is promised to match 3 decimal figures (this is the precision of such a respectful package as R). Test that claim and use evolution to research what the type `double` *and a fifth degree polynomial give of themselves.* *Answer*

283. ***Challenge.*** *Notice that our genetic algorithms produce a not conservative trend, in the sense that the best performance of the new generation might eventually be worst than the best performance of the given generation. Discuss the thesis of the Author that claims that this is due to recombination that breaks continuity. This is a bonus of evolution because thanks to this mechanism, it is never trapped in local optima. Modify the code to produce a conservative program. Now, this is true for easy problems but false for hard ones.*

14.3 Conclusion

We have been acquainted with the general scheme of a genetic algorithm, which tames evolution to have it serving a human purpose. We have witnessed that evolution works, yes, it works, but it is not indeed like a donkey that can be conducted by children, it rather looks like an alligator: one needs a lot of patience, training and smart ideas to produce something good. That is another reason because of which we propose evolution as a profession for a life. In other words, complexity can be somehow defeated by evolution (a term that includes genetic algorithms plus our effort to create them) but a lengthy track in the path for perfection can always be traced.

Chapter 15

From densities to p- and critical values

Velocity and accurateness

284. ***Purpose.*** *While our simulations must be restricted to a finite number of trials, in the order of millions for the best cases, mathematicians have produced results with simulations with infinitely many trials. Their results are consigned in the density functions. We use here those functions to infer the p-values of those distributions that are related to Z distribution. We also compute the usual critical values associated to a given significance level.*

15.1 Some recursive definitions

To use density functions in our Java codes, we need some prerequisites, the first of which is well known:

285. ***The factorial function of non negative integers 0,1,2,3,4...***

The next notation is used to define the factorial of integer n:

$n! = n \times (n-1) \times (n-2) \times ... \times 3 \times 2 \times 1$

Examples:

$0! = 1$ (definition)
$1! = 1$ (definition)
$2! = 2 \times 1 = 2$
$3! = 3 \times 2 \times 1 = 6$
$4! = 4 \times 3 \times 2 \times 1 = 24$
$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

286. ***Exercise.*** *Use a for-structure to implement the Java code for the factorial function over int.* *Answer*

Factorials also can be computed according to the next property:

$0! = 1$
$1! = 1 \times 0! = 1 \times 1 = 1$
$2! = 2 \times 1! = 2 \times 1 = 2$
$3! = 3 \times 2! = 3 \times 2 = 6$
$4! = 4 \times 3! = 4 \times 6 = 24$
$5! = 5 \times 4! = 5 \times 24 = 120$

This allows us to present a compact definition of the factorial function:

287. ***Recursive definition of the factorial function***

$$n!= \begin{cases} n \times ((n-1)!) \ if \ x > 1 \ is \ integer. \\ 1 \ if \ x = 0. \end{cases}$$

This type of definition is called recursive, the reason is understood if one uses it to calculate 7!:
$7! = 7 \times 6!$
$7! = 7 \times 6 \times 5!$
$7! = 7 \times 6 \times 5 \times 4!$
$7! = 7 \times 6 \times 5 \times 4 \times 3!$
$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2!$
$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1!$
$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 0!$
$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1 = 5040$

288. ***Java allows a wonderful code for this recursive definition:***

```
//Program E288 Factorial2
//Calculates the factorial function.
//Implements a recursive definition.

package ejvol5v2p;

public class Factorial2 {

//Returns numb!
    private static long factorialRec(long numb) {
```

```
        if (numb <= 1) {
            return 1;
        } else {
            return numb * factorialRec(numb - 1);
        }
    }

    public static void main(String[] args) {
        int n = 20;
        System.out.println("Facorials over type  long");
        for (int i = 1; i < n; i++) {
            System.out.println(i + "  " + factorialRec(i));
        }

        System.out.println("20!/19! = " + factorialRec(20) / factorialRec(19));
        System.out.println("30!/29! = " + factorialRec(30) / factorialRec(29));
    }
}//End of Program E288 Factorial2
```

289. ***Exercise.*** *Run the program, compare its output with that of the code of the previous exercise. Results must be identical otherwise there is a bug.*

290. ***Exercise.*** *Test previous program that calculates the factorial function with inputs from 20 to 30. Verify that it produces nonsense. Explain.* *Answer*

Types int and long are insufficient to deal with factorials. What else do we have?

291. ***The factorial with big numbers:***

```
//Program E291 Factorial3
//Calculates the factorial function.
//Big Numbers are used.
//Implements a recursive definition.

package ejvol5v2p;

import java.math.BigDecimal;

public class Factorial3 {

    private static final BigDecimal ONE = BigDecimal.valueOf(1);

//Returns numb!
    private static BigDecimal factorial(BigDecimal numb) {
```

```
        //output of compare:
        //+1 if numb is greater than ONE.
        //0 if they are equal
        //-1 if numb is less than ONE
        if (numb.compareTo(ONE) < 1) {
            return ONE;
        } else {
            return factorial(numb.subtract(ONE)).
                    multiply(numb);
        }
    }

    public static void main(String[] args) {
        int n = 500;
        for (int i = 1; i < n; i++) {
            System.out.println(i + "  "
                    + factorial(BigDecimal.valueOf(i)));
        }
        //Test for precision + overflow
        System.out.println(" Test: (n)!/(n-1)! = n");
        for (int i = 1; i < n; i++) {
            System.out.println(i + "  "
                    + factorial(BigDecimal.valueOf(i)).divide(
                            factorial(BigDecimal.valueOf(i - 1))));
        }
    }
}//End of Program E291 Factorial3
```

292. ***Exercise.*** *Run the program and declare whether or not your expectations have been fulfilled.*

293. ***The double factorial of an integer number 1,2,3,4, ..., is defined by:***

$$n!!= \begin{cases} (x) \times (x-2) \times ...5 \times 3 \times 1 \ if \ x > 0 \ is \ odd. \\ (x) \times (x-2) \times ...6 \times 4 \times 2 \ if \ x > 0 \ is \ even. \\ 1 \ if \ x \ is \ -1 \ or \ 0. \end{cases}$$

The recursive version of this definition can be written as:

$$n!!= \begin{cases} x \times (x-2)!! \ if \ x > 0. \\ 1 \ if \ x \ is \ -1 \ or \ 0. \end{cases}$$

Let us keep in mind that $n!!$ is quite different than $(n!)!$.
Examples:

$(-1)!! = 1$
$0!! = 1$
$1!! = 1 \times (-1)!! = 1 \times 1 = 1$
$2!! = 2 \times 0!! = 2 \times 1 = 2$
$3!! = 3 \times 1!! = 3 \times 1 = 3$
$4!! = 4 \times 2!! = 4 \times 2 = 8$
$5!! = 5 \times 3!! = 5 \times 3 = 15$
$6!! = 6 \times 4!! = 6 \times 8 = 48$

294. ***Exercise.*** *Use the Java type long to implement the Java code for the double factorial. Make appropriate tests to make sure that your code is right.* *Answer*

295. ***Exercise.*** *Use Java big numbers to implement the Java code for the double factorial.* *Answer*

The double factorial is used in the next definition:

296. ***The Gamma function*** Γ

The **Gamma function**, Γ, is defined for almost all real and complex numbers but we need only its values over positive integers, 1,2,3,4,.... and half-integers, 1/2, 3/2, 5/2, 7/2,...For n, a positive integer, we have the next properties that serve as tests for the Java implementation of this function:

$\Gamma(n) = (n-1)!$

Examples:
$\Gamma(1) = 0! = 1$
$\Gamma(2) = 1! = 1$
$\Gamma(3) = 2! = 2$
$\Gamma(4) = 3! = 6$
$\Gamma(5) = 4! = 24$
$\Gamma(6) = 5! = 120$

And we also have the next property:

$\Gamma(n + (1/2)) = \frac{(2n-1)!!}{2^n}\sqrt{\pi}$

Example:
$\Gamma(0 + 1/2) = \Gamma(1/2) = \sqrt{\pi} \sim 1.77$

Moreover
$\Gamma(x) = (x-1)\Gamma(x-1)$

Examples:
$\Gamma(3/2) = (1/2)\Gamma(1/2) \sim 0.8862$
$\Gamma(5/2) = (3/2)\Gamma(3/2) \sim 1.3293$
$\Gamma(7/2) = (5/2)\Gamma(5/2) \sim 3.3233$
$\Gamma(9/2) = (7/2)\Gamma(7/2) \sim 11.6$

Introduced definitions allow the next compact expression for the Γ function

$$\Gamma(x)= \begin{cases} (x-1)!\ if\ x\ is\ integer\ 0,1,2,3,4,.. \\ \frac{(2n-1)!!}{2^n}\sqrt{\pi}\ if\ x=n+1/2\ is\ halfInteger\ 1/2,3/2,5/2,7/2.... \end{cases}$$

The nonrecursive version of the Gamma function might help to understand it:

$$\Gamma(x/2)= \begin{cases} (x/2-1)(x/2-2)...3\times 2\times 1\ if\ x\ is\ even. \\ (x/2-1)(x/2-2)...3/2\times 1/2\times\sqrt{\pi}\ if\ x\ is\ odd. \end{cases}$$

297. *Exercise.* *Use the Java type long to implement the Java code for the Gamma function. Make appropriate tests to make sure that your code is right. Hint: to take the integer value of a double number, use*

```
int xRounded = (int) x;
```

Answer

298. *Exercise.* *Use Java big numbers to implement the Java code for the Gamma function. Recall that we already have a valuation of π using big numbers in program E232 pag 210. Answer*

299. *The beta function β*

The beta function is defined for degrees of freedom ν_1 in the numerator and ν_2 in the denominator by

$$\beta(\nu_1,\nu_2) = \frac{\Gamma(\nu_1)\Gamma(\nu_2)}{\Gamma(\nu_1+\nu_2)}$$

An alternate definition is:

$$\beta(\nu_1,\nu_2) = \int_0^1 x^{\nu_1-1}(1-x)^{\nu_2-1}dx$$

300. *Exercise.* *Calculate by hand $\beta(i,j)$ for $i,j<5$. Answer*

301. *Exercise.* *Use types long and double to implement the Java code for the β function according to both definitions and check their congruency. Make tests for small degrees of freedom and also for large ones, say 100 and 100. Answer*

302. *Exercise.* *Use big numbers to implement the Java code for the β function. Compare the performance of big numbers with that of the integral method that was implemented in the previous exercise. Answer*

15.2 p-values of the diverse distributions

Let us recall that the **p-value** associated to a specific event is the probability to find a value more extreme than it. A p-value can be defined with one or two tails. Let us learn to calculate the p-value for each one of the distributions seen in our course. The Reader is invited to check our programs against professional ones, say, those used by Gnumeric or Casio ([11] 2017)

303. ***The binomial distribution***

$$p(r) = C_r^n p^r q^{n-r} = C_r^n p^{n-r} q^r$$

where

$$C_r^n = \frac{n!}{r!(n-r)!}$$

304. ***Exercise.*** *Develop a program that for low values of n calculates the p-value of the binomial distribution $X \sim Bi(n, p)$. Distinguish between one and two tails. Use of type* `int` *will suffice for the computation of factorials.* *Answer*

305. ***Exercise.*** *Use and reuse previous programs to develop a professional program to calculate the p-value of the binomial distribution for a given input. Distinguish between one and two tails. Run your own tests and use official tables or Gnumeric to check the correctness of your program. The program receives three inputs: $1 \leq n \leq 500$, the number of coins or events; $0 \leq p \leq 1$, the probability of success or head; and $0 \leq k < n$, the number of successes. The program calculates for one tail $p(X > k)$ when $k > np = mean$ or $p(X < k)$ in contrary case. This result is multiplied by two for two tails.* *Answer*

306. ***Exercise.*** *We have two implementations of the same task, to calculate a binomial distribution. Compare them to establish how are the changes from the simpler version to the complex one. How can you use your observation in the study of the evolution?* *Answer*

307. ***Exercise.*** *We have developed a program with professional accurateness to calculate a binomial distribution. The price we paid was to use a heavy duty machinery. Maybe we have been too ostentatious because it is not excluded that with ordinary tools and a bit of wisdom we would had succeeded. To check this, study the next two options. First: use the type double instead of BigDecimal. Second: for factorials use the approximation provided by Ramanujan (Wikipedia, 2011a):*

$$\log n! \sim n \log n - n + \frac{\log(n(1+4n(1+2n)))}{6} + \frac{\log(\pi)}{2}.$$ *Answer*

308. ***Challenge: discuss the next argumentation:*** *The binomial distribution is indeed simple but because of memory constraints, it has become into a*

complex affair. We have produced many diverse solutions to it and many more exist in the literature. We can take this as a characterization of complexity: the complexity of a problem is related to the number of ways as it can be approximately solved. We have now a prediction: if species arouse by evolution, there must be huge variability in the forms as the same biological problems appear to have been solved in nature. The justification of our prediction is that while a conscious developer is acquainted with some few forms or styles of solving problems, randomness can mimic all possible developers at once. All this looks very sound but, how we will convert this wording into science?

309. *The normal distribution*

The density function of the normal distribution is:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

310. ***Exercise.*** *Develop a program to calculate the p-value of the non standardized normal distribution for a given input. Distinguish between one and two tails. Use official tables or Gnumeric to check the correctness of your program: insert → function → Statistics → normsdist.*

The one-tail solution of the standardized normal distribution might be implemented as a modification of, say, program E218, pag 199. Answer

311. *The t-distribution*

The density function of the t-distribution with $\nu = n-1$ degrees of freedom is given by

$$f_t(x) = \frac{\Gamma[(\nu+1)/2]}{\Gamma(\nu/2)\sqrt{\nu\pi}} (1 + \frac{x^2}{\nu})^{-\frac{\nu+1}{2}}$$

312. ***Exercise.*** *Use type double to develop a program to calculate the p-value of the t distribution for a given input and ν degrees of freedom. Distinguish between one and two tails. Use official tables or Gnumeric to check the correctness of your program: insert → function → Statistics → tdist. Answer*

313. ***Challenge.*** *One can achieve the accurateness of Gnumeric using type double. How does that accurateness look when compared with a program that uses Big Numbers? To decide this question, make the corresponding implementation.*

314. *The χ^2 distribution*

The density function of the χ^2 with $\nu = n-1$ degrees of freedom is given by

$$f_{\chi^2}(x) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} e^{-\frac{x}{2}}\, x^{\frac{\nu-2}{2}} \text{ for } x \geq 0.$$

315. ***Exercise.*** *Develop a program to calculate the p-value of the χ^2 distribution for a given input and ν degrees of freedom. Distinguish between one and two tails. Use official tables or Gnumeric to check the correctness of your program: insert → function → Statistics → chidist.* *Answer*

316. ***Challenge.*** *The reported by the Author answer to the previous exercise is a program whose outputs slightly differ from those of Gnumeric. Decide which is more exact. To that aim, use Big Numbers.*

317. ***The F distribution***

The density function of the F-distribution is given by

$$f_F(x) = \frac{\nu_1^{\frac{\nu_1}{2}}\nu_2^{\frac{\nu_2}{2}}x^{\frac{\nu_1}{2}-1}}{(\nu_2+\nu_1 x)^{\frac{\nu_1+\nu_2}{2}}\beta(\frac{\nu_1}{2},\frac{\nu_2}{2})} = \frac{\nu_1^{\frac{\nu_1}{2}}\nu_2^{\frac{\nu_2}{2}}}{\beta(\frac{\nu_1}{2},\frac{\nu_2}{2})}\frac{x^{\frac{\nu_1}{2}-1}}{(\nu_2+\nu_1 x)^{\frac{\nu_1+\nu_2}{2}}} \text{ for } x \geq 0.$$

Or

$$f_F(x) = k\frac{x^{\frac{\nu_1}{2}-1}}{(\nu_2+\nu_1 x)^{\frac{\nu_1+\nu_2}{2}}} \text{ with } k = \frac{\nu_1^{\frac{\nu_1}{2}}\nu_2^{\frac{\nu_2}{2}}}{\beta(\frac{\nu_1}{2},\frac{\nu_2}{2})}$$

where β is the beta function with degrees of freedom $\nu_1 = n_1 - 1$ in the numerator and $\nu_2 = n_2 - 1$ in the denominator.

318. ***Exercise.*** *Use and reuse previous programs to develop a Java class to calculate the p-value of the F distribution for a given input and ν_1 degrees of freedom in the numerator and ν_2 degrees of freedom in denominator. Distinguish between one and two tails.* *Answer*

319. ***Challenge: prove the next result that leads to a more professional approach to the calculation of p-values for the F-distribution.***

The density of the F- distribution is:

$$f_F(x) = \frac{\nu_1^{\frac{\nu_1}{2}}\nu_2^{\frac{\nu_2}{2}}x^{\frac{\nu_1}{2}-1}}{(\nu_2+\nu_1 x)^{\frac{\nu_1+\nu_2}{2}}\beta(\frac{\nu_1}{2},\frac{\nu_2}{2})} = \frac{\nu_1^{\frac{\nu_1}{2}}\nu_2^{\frac{\nu_2}{2}}}{\beta(\frac{\nu_1}{2},\frac{\nu_2}{2})}\frac{x^{\frac{\nu_1}{2}-1}}{(\nu_2+\nu_1 x)^{\frac{\nu_1+\nu_2}{2}}} \text{ for } x \geq 0.$$

Consider now the following definitions:

$u = \frac{\nu_1 x}{\nu_2+\nu_1 x}$ *(change of variable with inverse* $x = \frac{u\nu_2}{(1-u)\nu_1}$ *)*

$\beta(u;a,b)$ *is the* ***incomplete*** β ***function*** *given by*

$$\beta(u;a,b) = \int_0^u t^{a-1}(1-t)^{b-1}dt$$

I_β *is the* ***regularized incomplete beta function*** *given by*

$I_\beta(u;a,b) = \frac{\beta(u;a,b)}{\beta(1;a,b)}$

Prove that

$F(x) = \int_0^x f_F(x)dx = I_\beta(u;\nu_1/2,\nu_2/2)$

Therefore, the p-value with the upper tail associated to V, a given F-value, is:

$p\text{-}value = 1 - I_\beta(\frac{\nu_1 V}{\nu_2+\nu_1 V};\nu_1/2,\nu_2/2)$

Use the next identity as a test:

$I_\beta(u;a,b) + I_\beta(1-u;b,a) = 1.$

The error against the exact equality gives an estimation of the precision of the algorithm.

320. ***Exercise.*** *Implement into a Java program the aforementioned professional approach to calculate the p-value of a random variable of an F distribution. Compare results with Gnumeric: insert → function → Statistics → fdist. Type the F-value, the degrees of freedom of the numerator and of the denominator. The output must coincide with our p-value for one tail.* Answer

321. ***Challenge.*** *The outputs of the previous program differs from those of Gnumeric beyond the 9th decimal place. Use Big Numbers to decide which is more exact.*

322. ***Challenge.*** *Refine the previous code to duplicate its velocity. To that aim, notice that the code makes twice the same calculation.*

323. ***The Pareto distribution.***

The probability density of a Parteto distribution is

$$f(x) = \begin{cases} 0, & \text{if } x < m, \\ \frac{rm^r}{x^{r+1}}, & \text{if } x \geq m, \end{cases}$$

324. ***Exercise.*** *Develop a program to calculate the p-value of the Pareto distribution for a given input V, power r and a cut value m. You can also use integration techniques to solve the problem.* Answer

325. ***Challenge:*** *make a program to calculate p-values of the Kolmogorov-Smirnov density function.*

15.3 Critical values

Critical values mark the borders that divide normal from extreme events such that the total proportion of extreme events is α.

326. ***Definition.*** *For a random variable X with probability density $f(s)$ and cumulative distribution function $F = \int_{-\infty}^{x} f(s)ds$, and for a significance level α and one tail, the* ***upper critical value*** *x_u is defined by*

$$p(X > x_u) = \alpha.$$

For the lower tail, the lower critical value x_l is defined by

$$p(X < x_l) = \alpha.$$

For two tails, we have two critical values, the upper, x_+, and the lower one, x_-, defined by :

$$p(X > x_+) = \alpha/2$$

and

$$p(X > x_+) = \alpha/2.$$

327. ***Criticism and challenge.*** *In the previous definition of the critical values for two tails, we divided the total significance level by two, because we must accumulate values in two tails. This option is not necessarily the best: one might incorporate in the critical value two concepts: to be extreme and to have low probability. If we follow this criterion, we must begin the accumulation procedure with the term that is both the most extreme and the less probably to continue with the next in the ensuing hierarchy no matter in which tail it might be. Although this criticism will not be followed here, the reader is invited to tailor and implement these concepts.*

328. ***Exercise.*** *Develop a program to calculate the critical values of the binomial distribution for one and two tails for a given significance level α. Check results with Gnumeric: insert → function → Statistics → =binomdist(0,6,0.25,1). The number of successes is 0, the number of trials is 6, the probability of success is 0.25 and the last 1 means that we evaluate the cumulative distribution function. Answer*

329. ***A trick***

To use and reuse previous programs for developing new applications, one would desire to make the less number of changes. Why? Because **every change in a program creates bugs whose corrections generates more bugs**. So, let us learn to force a for-structure to simulate a while-structure. This can be learned by running the next code:

```
//Program E329 WhileFor
//Uses a for-structure to simulate a while-structure.

package ejvol5v2p;


public class WhileFor {

    public static void main(String[] args) {
        //While
        System.out.println("NATIVE WHILE");
        double sum = 0;
        long i = 0;
        double one = 1;
        double target = 3.456789;
        while (sum < target) {
            i = i + 1;
            sum = sum + one / i;
            System.out.println(i + "  sum = " + sum);
        }
        System.out.println("To cumulate " + target
                + ", we need the first " + i + " Terms");
        //Simulating a while with a for loop
        sum = 0;
        System.out.println("\nSIMULATED WHILE");
        for (long k = 1; sum < target; k++) {
            sum = sum + one / k;
            System.out.println(k + "  sum = " + sum);
            i = k;
        }
        System.out.println("To cumulate " + target
                + ", we need the first " + i + " Terms");
    }
}//End of Program E329 WhileFor
```

330. ***Exercise.*** *Run the previous code and compare the native while-structure with the simulated one.*

331. ***Challenge: Discuss the net reasoning on Evolution vs Intelligence***

Our strategy is to use developed code to design new one. Thus, our strategy is evolution. In this regard, to develop programs to calculate critical values, we use those that calculate p-values. Nevertheless, our strategy is not elegant: to

calculate p-values, we calculated a definite integral from zero to certain value, but to adapt this procedure to calculate critical values associated to a given significance level, we need to accumulate area up to that level and then find the corresponding event. Elegance appears as follows: the cumulative function we are considering are monotone, always increasing functions, so they are invertible. Then, find the inverse and use it to predict critical values with one line of code. The problem is that in general there is no close formulas for the inverse of distribution functions, so numerical receipts are generally applied. Anyway, the path of the inverse of cumulative functions is the preferred one by experts in numerical analysis.

These considerations allow us to claim that intelligence of non evolutionary origin can be detected in a family of programs if new strategies that can be explained by mathematical reasoning are preferred to those naturally posited by evolution.

332. ***Example.*** *Let us develop a program to calculate the critical values of a standardized normal distribution for one and two tails and for a given significance level* α*. Results might be checked with Gnumeric: insert* $\rightarrow$ *function* $\rightarrow$ *Statistics* $\rightarrow$ *norminv(0.95,0,1). (0.95 = 1 - significance level; 0 = mean; 1 = deviation).*

We use previous code but with appropriate modifications to produce a program that must be good to accumulate area and not only to calculate a final one. To that aim, we use the Simpson's rule in its crude form (see numeral E216, page 195):

$$\int_a^b p(x)dx = \frac{b-a}{6}[f(a) + 4f(m) + f(b)] = \frac{b-a}{6}[f(a) + 4f(\frac{a+b}{2}) + f(b)].$$

In our case, $a = jh$, $b = (j + 1)h$, $m = (j + 0.5)h$. The code follows:

```
//Program E332 CritNormal
//Reports critical values
//for a normal distribution.
//Combines Simpson's rule
//with a change of scale.


package ejvol5v2p;


public class CritNormal {

    private static final double PI = 3.14159265358979323846;
    private static double z;
    private static double mean;
    private static double deviation;
```

```
//Significance level
private static double alpha;

//****************Style: reuse*************

//Returns the transformed density function of
//the standardized normal distribution.
private static double zFunction(double t) {
    z = t / (1 - t * t);
    double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2)
            * (1 + t * t) / ((1 - t * t) * (1 - t * t));
    return f;
}

//To use this method for another distribution,
//instead of  zFunction(z), write and make a call for your
//new distribution.
private static double f(double x) {
    double f = zFunction(x);
    return f;
}

//Reports the transformed upper u-critical for
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
private static double semiCum(double u,
        long N,
        double alpha) {
    double h = u / N;
    double r = h / 6;
    double sum = 0.5;
    double uCrit = 0;
    double target = 1 - alpha;
    //System.out.println("Target = " + target);
    for (int j = 0; sum < target; j++) {
        sum = sum
                + r * (f(j * h) + 4 * f((j + 0.5) * h) + f((j + 1) * h))
        uCrit = j * h;
        //System.out.println(uCrit + " " + sum);
    }
    return uCrit;
}

//Reports the upper z-critical for
//Significance level = alpha.
```

```
    private static double zCrit(double alpha) {
        double u = 1;
        int N;
        int i = 3;
        N = (int) Math.pow(10, i);
        double uCrit = semiCum(u, N, alpha);
        double zCrit = uCrit / (1 - uCrit * uCrit);
        return zCrit;
    }

    public static void main(String[] args) {
        mean = 5;
        deviation = 0.7;
        alpha = 0.05;
        double zCrit = zCrit(alpha);
        System.out.println("Upper critical value for one tail "
                + "\nof the normal "
                + "distribution with \nmean = "
                + mean + ", deviation = " + deviation
                + ".\nSignificance level = " + alpha + ".");

        System.out.println("zCritic = " + zCrit);
        double criticalvalue = zCrit*deviation + mean;
        System.out.println("Critical value =  mean + zCrit*deviation");
        System.out.println("Critic value of given "
                + "normal distribution = " + criticalvalue);
    }
}//End of Program E332 CritNormal
```

333. ***Exercise.*** *Run the program and play with the code. Verify that to fit the precision of Gnumeric, one must increment the number of subdivisions beyond* 10^3*, but that this implies too many calculations and so the output appears after an extremely and intolerable long delay.*

Let us see two methods to improve precision and velocity of our algorithm.

334. ***Local refinment*** *In our previous program, we divide the universe, the interval (0,1), in 1000 parts. To achieve a better approximation, let us divide the sub-interval where the critical value is enclosed in other thousand parts to see what we achieve. The code follows.*

```
//Program E334 CritNormal2
//Reports critical values
//for a normal distribution.
//Combines Simpson's rule
//with a change of scale.
```

```
//A refinement is checked out.

package ejvol5v2p;

public class CritNormal2 {

    private static final double PI = 3.14159265358979323846;
    private static double z;
    private static double mean;
    private static double deviation;
    //Significance level
    private static double alpha;
    private static double target;
    private static double greatSum;
    private static double uCritic;
    private static int N;

    //****************Style: reuse*************

    //Returns the transformed density function of
    //the standardized normal distribution.
    private static double zFunction(double t) {
        z = t / (1 - t * t);
        double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2)
                * (1 + t * t) / ((1 - t * t) * (1 - t * t));
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double x) {
        double f = zFunction(x);
        return f;
    }

    //Reports the transformed upper u-critical for
    //Significance level = alpha.
    //Precision=1/N.
    //Simpson's rule adapted for reuse.
    private static double semiCum(double u,
            long N,
            double target) {
        double h = u / N;
        double r = h / 6;
        double sum = 0.5;
```

```
        double jCrit = 0;
        //System.out.println("Target = " + target);
        for (int j = 0; sum < target; j++) {
            sum = sum
                    + r * (f(j * h) + 4 * f((j + 0.5) * h) + f((j + 1) * h));
            jCrit = j + 1;
            //System.out.println(jCrit*h + " " + sum);
        }
        jCrit = jCrit - 1;
        //System.out.println("jCrit = " + jCrit);
        greatSum = sum
                - r * (f(jCrit * h) + 4 * f((jCrit + 0.5) * h)
                + f((jCrit + 1) * h));

        double uAprox = (jCrit) * h;
        /*System.out.println("uCritic = " + uAprox
                  + " greatSum = " + greatSum;
         */
        return uAprox;
    }

    //Reports the upper z-critical for
    //Significance level = alpha.
    private static double zCrit(double alpha) {
        double u = 1;
        int i = 3;
        N = (int) Math.pow(10, i);
        uCritic = semiCum(u, N, alpha);
        double zCritic = uCritic / (1 - uCritic * uCritic);
        return zCritic;
    }

    //A refinement is implement: the critical interval
    //is divided in 1000 subintervals.
    private static double second(double target) {

        double h = 1;
        h = h / (N * N);
        double r = h / 6;
        double sum = greatSum;
        double jCrit = 0;
        double b = uCritic;
        /*
  System.out.println("N = " + N);
  System.out.println("h = " + h);
  System.out.println("Target = " + target);
```

```
  System.out.println("b = " + b);
         */
        for (int j = 0; sum < target; j++) {
            sum = sum
                    + r * (f(b + j * h) + 4 * f(b + (j + 0.5) * h)
                    + f(b + (j + 1) * h));
            jCrit = j;
            //System.out.println(b+j*h + "     " + sum);
        }
        greatSum = sum
                - r * (f(b + jCrit * h) + 4 * f(b + (jCrit + 0.5) * h)
                + f(b + (jCrit + 1) * h));
        double uCritic2 = b + (jCrit - 1) * h;
        /*System.out.println("uCritic = "
+ uCritic + " gretSum = " + greatSum);
         */
        double zCritic = uCritic2 / (1 - uCritic2 * uCritic2);
        return zCritic;
    }

    //************Test********
    //The cumulative function evaluated at z
    private static double cum(double z, long N) {

        double h = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * h);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * h);
        }
        sum = sum + f((2 * m) * h);
        sum = (h / 3) * sum;
        return sum + 0.5;
    }

    //The cumulative function evaluated at
    //zCritic must be equal to the target = 1-alpha.
    private static void test(double z) {

        System.out.println("N in 10^N  Simpson's rule,"
                + "\nArea under the bell in within -infty and " + z);
        z = (Math.sqrt(1 + 4 * z * z) - 1) / (2 * z);
        int NN;
```

```
        double answer = 0;
        for (int i = 3; i < 5; i++) {
            NN = (int) Math.pow(10, i);
            answer = cum(z, NN);
            System.out.println(i + "\t" + answer);
        }
        double error = target - answer;
        System.out.println("Error = " + error);
    }

    public static void main(String[] args) {
        mean = 5;
        deviation = 0.7;
        alpha = 0.05;
        target = 1 - alpha;
        double zCrit = zCrit(target);
        System.out.println("Upper critical value for one tail "
                + "\nof the normal "
                + "distribution with \nmean = "
                + mean + ", deviation = " + deviation
                + ".\nSignificance level = " + alpha + ".");

        System.out.println("z-critic, first approximation = "
                + zCrit);
        zCrit = second(target);
        System.out.println("z-critic, second approximation = "
                + zCrit);
        System.out.println("\nTest: it must return 1-alpha = "
                + (1 - alpha));
        //Test to measure the accurateness of the approximation
        test(zCrit);
    }
}//End of Program E334 CritNormal2
```

335. ***Exercise.*** *Run the program and play with the code. Play enough to agree else disagree with the Author: the improvement in precision is insignificant.*

336. ***Analytic hunting.*** *Let use numerical trickery to find a better and swifter approximation to the critical value for the normal distribution.*

We know how to swiftly find an approximation for the critical value for a given significance up to two decimal places. The critical interval containing the critical value also has been specified. With that approximation, we commit an error E that always can be assumed to be positive. To find a better approximation, let us solve the next problem in which we fit the error E

$$\int_a^b p(x)dx = \frac{b-a}{6}[f(a) + 4f(m) + f(b)] = \frac{b-a}{6}[f(a) + 4f(\frac{a+b}{2}) + f(b)] = E.$$

here, a is the left border of the critical interval and we need to specify b as follows:

$h = \frac{b-a}{6}$

$\frac{b-a}{6}[f(a) + 4f(\frac{a+b}{2}) + f(b)] = h[f(a) + 4f(\frac{a+b}{2}) + f(b)] = E$

$h[f(a) + 4f(\frac{a+b}{2}) + f(b)] - E = 0$

So, we need to find a zero of a function

$F(b) = \int_a^b f(x)dx - E$
$F(b) = h[f(a) + 4f(\frac{a+b}{2}) + f(b)] - E$

Let us recall the Newton's algorithm, 12.4, to find a zero of a differentiable function $F(x)$, when a close initial approximation g is given:

$g_o = g$
$g_{i+1} = g_i - \frac{F(g_i)}{F'(g_i)}$

To apply this method, we take $F(b) = h[f(a) + 4f(\frac{a+b}{2}) + f(b)] - E$, and $g = r$, where r is the right border of the critical interval (which contains the critical value).

To calculate $F'(b)$, we use the fundamental theorem of calculus for continuous function f:

$F(b) = \int_a^b f(x)dx - E$
$F'(b) = f(b)$

So, taking b as the left border of the critical interval, the algorithm reads:

$g_o = b$
$g_{i+1} = g_i - \frac{F(g_i)}{F'(g_i)} = g_i - \frac{r[f(g_i)+4f(\frac{a+g_i}{2})+f(g_i)]-E}{f(g_i)}$

337. ***Example*** *Let us previous theory to develop a program to exactly calculate the one tail upper critical value of the standardized normal distribution for a given significance level α. We will check results with Gnumeric: insert $\rightarrow$ function $\rightarrow$ Statistics $\rightarrow$ normsinv. We will also devise a test on our own without the help of Gnumeric.*

```
//Program E337 CritNormal3
//Reports the one tail upper critical value
```

```
//for a normal distribution.
//Combines Simpson's rule
//with a change of scale.
//An initial approximation is improved
//by analytical hunting.

package ejvol5v2p;

public class CritNormal3 {

    private static final double PI = 3.14159265358979323846;
    private static double z;
    private static double mean;
    private static double deviation;
    //Significance level
    private static double alpha;
    private static double target;
    private static double greatSum;
    private static double Error;
    private static double uCritic;
    private static double zCritic;
    private static double jCritic;
    //Right border or the critical interval
    private static double rBorder;
    private static double h;
    private static int N;

    //****************Style: reuse*************
    //Returns the transformed density function of
    //the standardized normal distribution.
    private static double zFunction(double t) {
        z = t / (1 - t * t);
        double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2)
                * (1 + t * t) / ((1 - t * t) * (1 - t * t));
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double x) {
        double f = zFunction(x);
        return f;
    }

    //Reports the transformed upper u-critical for
```

```
    //the given distribution.
    //Significance level = alpha.
    //Precision=1/N.
    //Simpson's rule adapted for reuse.
    private static double upperU(double u,
            long N,
            double target) {
        h = u / N;
        double r = h / 6;
        double sum = 0.5;
        jCritic = 0;
        //System.out.println("Target = " + target);
        for (int j = 0; sum < target; j++) {
            sum = sum
                    + r * (f(j * h) + 4 * f((j + 0.5) * h) + f((j + 1) * h))
            jCritic = j + 1;
            //System.out.println(jCrit*h + " " + sum);
        }
        jCritic = jCritic - 1;
        //System.out.println("jCrit = " + jCrit);
        greatSum = sum
                - r * (f(jCritic * h) + 4 * f((jCritic + 0.5) * h)
                + f((jCritic + 1) * h));

        double uAprox = (jCritic) * h;
        /*System.out.println("uCritic = " + uAprox
                  + " greatSum = " + greatSum;
         */
        return uAprox;
    }

//Returns the transformed value z = u/(1-u*u)
    private static double z(double u) {
        double zz = u / (1 - u * u);
        return zz;
    }

    //Reports the upper z-critical for
    //Significance level = alpha.
    private static double zCrit(double alpha) {
        double u = 1;
        int i = 3;
        N = (int) Math.pow(10, i);
        uCritic = upperU(u, N, alpha);
        Error = target - greatSum;
        double zCritic2 = z(uCritic);
```

```
        return zCritic2;
    }

    //*****Analytical hunting*******
    //Returns the density function of the Z distribution
    private static double ff(double z) {
        double f = Math.exp(-z * z / 2) / Math.pow(2 * PI, 0.5);
        return f;
    }

    // Returns F(b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
    private static double F(double b) {
        double a = zCritic;
        double hh = (b - a) / 6;

        double Fb = hh * (ff(a) + 4 * ff((a + b) / 2)
                + ff(b)) - Error;
        return Fb;
    }

    //Returns the  ratio  F(gi)/ ff(gi)
    private static double ratio(double gi) {
        double ratio = F(gi) / ff(gi);
        System.out.println("ratio = " + ratio);
        return ratio;
    }

    //A better approximation is found by
    //analytical hunting
    private static double hunting(double zCritic) {
        double gSubi = rBorder;
        double gSubiPlus = 0;
        int iterations = 0;
        int maxIterations = 30;
        boolean more = true;
        while (more) {
            gSubiPlus = gSubi - ratio(gSubi);
            iterations = iterations + 1;
            if (iterations >= maxIterations) {
                more = false;
            }
            if (gSubiPlus == gSubi) {
                more = false;
            }
            gSubi = gSubiPlus;
            System.out.println("iteration = " + iterations
```

```
                + " new zcrit = " + gSubiPlus);
        }
        return gSubiPlus;
    }

    //************Test********
    //The cumulative function evaluated at z
    private static double cum(double z, long N) {
        double hh = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * hh);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * hh);
        }
        sum = sum + f((2 * m) * hh);
        sum = (hh / 3) * sum;
        return sum + 0.5;
    }

    //The cumulative function evaluated at
    //zCritic must be equal to the target = 1-alpha.
    private static void test(double z) {
        System.out.println("N in 10^N  Simpson's rule,"
                + "\nArea under the bell in within -infty and " + z);
        z = (Math.sqrt(1 + 4 * z * z) - 1) / (2 * z);
        int NN;
        double answer;
        NN = (int) Math.pow(10, 3);
        answer = cum(z, NN);
        System.out.println(answer);
        double error = target - answer;
        System.out.println("Forth and back Error = " + error);
    }

    public static void main(String[] args) {
        mean = 5;
        deviation = 0.7;
        alpha = 0.05;
        target = 1 - alpha;
        System.out.println("Upper critical value for one tail "
                + "\nof the normal "
                + "distribution with \nmean = "
                + mean + ", deviation = " + deviation
```

```
                + ".\nAlpha = Significance level = "
                + alpha + ".");
        zCritic = zCrit(target);
        System.out.println("The critical interval begins with "
                + zCritic);
        double ur = (jCritic + 1) * h;
        rBorder = z(ur);
        System.out.println("The critical interval ends at "
                + rBorder);

        System.out.println("Cumulated area = " + greatSum);
        System.out.println("Error = " + Error);
        zCritic = hunting(zCritic);
        System.out.println("z-critic, hunting approximation = "
                + zCritic);
        System.out.println("\nTest: it must return 1-alpha = "
                + (1 - alpha));
        //Test to measure the accurateness of the approximation
        test(zCritic);
    }
}//End of Program E337 CritNormal3
```

338. ***Exercise.*** *Run the program and play with the code. Verify that it exactly fulfills the corresponding mathematical procedure.*

339. ***Exercise.*** *Develop a professional program to exactly calculate the critical values of the standardized normal distribution for one and two tails and for a given significance level α.* Answer

340. ***Exercise.*** *Develop a professional program to exactly calculate the critical values of the t-distribution for one and two tails and for a given significance level α. Check results with Gnumeric: insert $\rightarrow$ function $\rightarrow$ Statistics $\rightarrow$ tinv. Also devise a test on your own without the help of Gnumeric.* Answer

341. ***Exercise.*** *Develop a professional program to exactly calculate the critical values of χ^2-distribution for one and two tails and for a given significance level α. Check results with Gnumeric: insert $\rightarrow$ function $\rightarrow$ Statistics $\rightarrow$ chiinv. Also devise a test on your own without the help of Gnumeric.* Answer

342. ***Exercise.*** *Develop a professional program to exactly calculate the critical values of the F-distribution for one and two tails and for a given significance level α. Check results with Gnumeric: insert $\rightarrow$ function $\rightarrow$ Statistics $\rightarrow$ finv. Also devise a test on your own without the help of Gnumeric.* Answer

343. ***Exercise.*** *Develop a program to calculate the critical values of the Pareto distribution for one tail and for a given significance level α. The problem also can be solved mathematically.* Answer

344. *Achieving Perfection*

Our programs produce results that are slightly different than those of Gnumeric. It is straightforward to see that the results of Gnumeric are more exact than ours because our forth and back errors are detectable while those of Gnumeric are zero (up to reported precision). This contrast fires our curiosity: how can we achieve such perfection without going into numerical expertise, just with our nails?

To improve accurateness, we have applied mathematical hunting. So, the next program explores what happens if we use recurrently that method. We test the idea over the last program, which gives critical values for the F-distribution.

```
//Program E344 CritF2
//Reports   critical values
//for the F-distribution
//and a given significance level.
//Combines Simpson's rule
//with a change of scale.
//An initial approximation is improved
//by analytical hunting.
//Exaggerated  perfection.

package ejvol5v2p;


public class CritF2 {


    //Significance level
    private static double alpha;
    private static double target;
    private static double greatSum;
    private static double Error;
    private static double uCritic;
    private static double vCritic;
    private static double jCritic;
    //Right border or the critical interval
    private static double rBorder;
    private static double h;
    private static int N;
    private static double nu1, nu2;
    private static double k;

    //****************Style: reuse*************
    //Returns the beta   function  with   d.f. =
    //a(num) and b(den)
```

```
    private static double g(double x, double a, double b) {
        double g = Math.pow(x, a - 1)
                * (Math.pow(1 - x, b - 1));
        return g;
    }

    //Returns the integral under the function f
    //in within 0 and u. Precision=1/N.
    //Simpson's rule adapted for reuse.
    //a and b are degrees of freedom
    private static double HalfBodySimpsong(double u,
            double a, double b, long N) {
        double hh = u / (2 * N);
        long m = N;
        double sum = g(0, a, b);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * g((2 * j - 1) * hh, a, b);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * g((2 * j) * hh, a, b);
        }
        sum = sum + g((2 * m) * hh, a, b);
        sum = (hh / 3) * sum;
        return sum;
    }

    //Returns the  beta  function in integral form
    //with a and b d.f.
    private static double beta(double a, double b) {
        double answer;
        int i = 3;
        int NN = (int) Math.pow(10, i);
        double u = 1;
        answer = HalfBodySimpsong(u, a, b, NN);
        return answer;
    }

    //****************F density function*************
    //Returns the transformed density function of the
    //F distribution with nu1 and nu2 d.f.
    private static double FFunction(double t) {
        double z = t / (1 - t * t);
        double f = k * Math.pow(z, nu1 / 2 - 1)
                * Math.pow(nu2 + nu1 * z, -(nu1 + nu2) / 2)
                * ((1 + t * t) / ((1 - t * t) * (1 - t * t)));
        //System.out.println(f);
```

```
    return f;
}

//To use this method for another distribution,
//instead of  FFunction(z), write and make a call for your
//new distribution.
private static double f(double z) {
    double f = FFunction(z);
    return f;
}

//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
private static double upperU(double u,
        long N,
        double target) {

    double a = Math.pow(nu1, nu1 / 2);
    double b = Math.pow(nu2, nu2 / 2);
    double c = beta(nu1 / 2, nu2 / 2);
    k = a * b / c;
    h = u / N;
    double r = h / 6;
    double sum = 0;
    jCritic = 0;
    for (int j = 0; sum < target; j++) {
        sum = sum
                + r * (f(j * h) + 4 * f((j + 0.5) * h) + f((j + 1) * h))
        jCritic = j + 1;
    }
    jCritic = jCritic - 1;
    greatSum = sum
            - r * (f(jCritic * h) + 4 * f((jCritic + 0.5) * h)
            + f((jCritic + 1) * h));

    double uAprox = (jCritic) * h;
    return uAprox;
}

//Returns the transformed value z = u/(1-u*u)
private static double z(double u) {
    double z = u / (1 - u * u);
    return z;
```

```
    }

    //Reports the upper z-critical for
    //Significance level = alpha.
    private static double vCrit(double alpha) {
        double u = 1;
        int i = 4;
        N = (int) Math.pow(10, i);
        uCritic = upperU(u, N, target);
        Error = target - greatSum;
        double zCritic = z(uCritic);
        return zCritic;
    }

    //*****Analytical hunting*******
    //Returns the density function of the t-distribution
    private static double ff(double z) {
        double f = k * Math.pow(z, nu1 / 2 - 1)
                * Math.pow(nu2 + nu1 * z, -(nu1 + nu2) / 2);
        return f;
    }

    //F(b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
    private static double F(double b) {
        double a = vCritic;
        double hh = (b - a) / 6;
        double Fb = hh * (ff(a) + 4 * ff((a + b) / 2)
                + ff(b)) - Error;
        return Fb;
    }

    //Returns the  ratio  F(gi)/ ff(gi)
    private static double ratio(double gi) {
        double ratio = F(gi) / ff(gi);
        return ratio;
    }

    //Critical value for the upper tail.
    //A professional  approximation is found by
    //analytical hunting
    private static double hunting() {
        double gSubi = rBorder;
        double gSubiPlus = 0;
        int iterations = 0;
        int maxIterations = 30;
        boolean more = true;
```

```
        while (more) {
            gSubiPlus = gSubi - ratio(gSubi);
            iterations = iterations + 1;
            if (iterations >= maxIterations) {
                more = false;
            }
            if (gSubiPlus == gSubi) {
                more = false;
            }
            gSubi = gSubiPlus;
            /*System.out.println("iteration = " + iterations +
              "New crit-value = " + gSubi);*/
        }
        return gSubiPlus;
    }

    //Critical value for significance level alpha, two tails
    private static double critical(
            double alpha) {
        target = 1 - alpha;
        vCritic = vCrit(target);
        //System.out.println("First approx = " + vCritic);
        double ur = (jCritic + 1) * h;
        rBorder = z(ur);
        vCritic = hunting();
        System.out.println("Hunting approx = " + vCritic);
        return vCritic;
    }

    //************Test********
    //The cumulative function evaluated at z
    private static double cum(double z, long N) {
        double hh = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * hh);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * hh);
        }
        sum = sum + f((2 * m) * hh);
        sum = (hh / 3) * sum;
        return sum;
    }
```

```
    //The cumulative function evaluated at
    //vCritic must be equal to the target = 1-alpha.
    private static void test(double z) {
        System.out.println(
                "\nThe area under the F- density function "
                + "in within 0 and \n" + z + " is ");
        z = (Math.sqrt(1 + 4 * z * z) - 1) / (2 * z);
        int NN;
        double answer;
        NN = (int) Math.pow(10, 4);
        answer = cum(z, NN);
        System.out.println("            " + answer);
        System.out.println("Expected =  " + target);
        Error = target - answer;
        System.out.println("Forth and back Error = " + Error);
    }

    public static void main(String[] args) {
        alpha = 0.05;
        nu1 = 7;
        nu2 = 6;

        System.out.println("Critical values for one tail "
                + "\nof the F- "
                + "distribution with dfs   "
                + nu1 + " and " + nu2 + ", "
                + "\nAlpha = Significance level = "
                + alpha + ".");

        System.out.println("Lower critical value  = "
                + critical(1 - alpha));
        System.out.println("Upper critical value  = "
                + critical(alpha));
        System.out.println("\nCritical values for two tails "
                + "\nof the chi-square "
                + "distribution."
                + "\nAlpha = Significance level = "
                + alpha + ".");
        double alphaHalf = alpha / 2;

        System.out.println("Lower critical value  = "
                + critical(1 - alphaHalf));

        System.out.println("\nTEST");
        //Test to measure the accurateness of the approximation
        test(vCritic);
```

```
        //Second application of mathematical hunting
        vCritic = hunting();
        System.out.println("\nSecond hunting approx = " + vCritic);
        test(vCritic);
    }
}//End of Program E344 CritF2
```

345. ***Exercise.*** *Run the program and detect where analytical hunting was used by a second time. Test what happens to the overall precision if the Simpon's rule is applied using a partition of 100 intervals instead of 10000.* *Answer*

346. ***Challenge.*** *Apply your conclusion arrived at the last exercise to improve perfection and velocity of all your previous programs.*

347. ***Worm holes.*** *When one uses and reuses previous material to develop new programs, one can trace a line of descent. But it usually happens that working in late programs one discovers that something can be done to improve the overall quality of early ones. In this way, the line of descent gets entangled and the future is mixed with the past, as it happens in the worm holes of physics. So, a line of evolution is in general ill defined and may be accepted only as a concession. Biology is also involved in the discussion: thanks to horizontal gene transfer, late development might eventually be incorporated into early products. How can we detect or reject such possibilities?*

348. ***Validation of algorithms.***

Software developers are instinctively guided to check their results. In that regard, we have been greatly favored by Gnumeric although we have devised our own tests.

As a rule, one tests a given algorithm for one or two values and rarely more than that. Very careful people might check up to ten values. One proceeds in that way because one feels that a complex program cannot fit a sophisticated and specific target unless it is correct for all values. This anomalous logic is the real logic of software development but, of course, it is against the logic used by pristine mathematics that go like this:

A program can be understood as a function, a recursive function, that for a given input produces a given output. Since an algorithm is a function, the same output always must be produced by the same input. This remains true even when one uses random numbers because the generator of random numbers is also a program that is deterministic and always produces the same output on the same input or seed. Therefore, the validation of algorithms is just a mathematical task that consists in reproducing the function that the algorithm implements to compare it with the function that was the target of the developer. The two functions must be equal one to another. Now, when two functions are

equal? They are equal when they respond to the same input in the same form. And this must be valid for all possible inputs not just two or three.

Thus, we end with a discrepancy in the number of checked values: anomalous logic takes one or two, while mathematical logic requires all possible values. Now, this discrepancy cannot be solved numerically, so it must be solved analytically. The way as this is done is to run the program in a virtual mathematical computer. This game is known as **lambda calculus**: it captures the function fulfilled by any given method. We can understand what lambda calculus consists in if we look at the next method:

```
private static double  square(double x)
{
 return x*x;
}
```

If one makes a numerical test with $x = 5$, one finds that the method outputs 25. One can also check the method for $x = 7$ to witness that it outputs its square. We make in this way numerical testing. But, we also can look at the method and read directly that for input x, it outputs x^2, for every possible value x. If this is what the developer had in mind, the method has been validated. The automation of this procedure is the **lambda calculus** (Selinger, [91] 2007).

Unfortunately, lambda calculus has two dangerous assumptions. First: that the programs or methods will be run in computers with infinite precision. The problem is that such computers or computing languages do not exist. Instead, we have numerical trickery to try to remedy the troubles created by finite precision and limited memory. Second: that programs are simply and that lambda calculus produces understandable output. This is also false. By these reasons, the validation of algorithms is in general a task that is delayed and delayed for ever. The price that we must pay is that programs collapse and systems fall down. Next, bugs are unveiled and fixed until a new crash happens and so on.

That is why software is always dressed in abundant bugs that companies try to unveil and fix through beta versions and automatic feedback. In that way this **law of abundant bugs** is always true for software solutions that are engineered by human beings and their tools. What can we say about software produced by natural evolution? The answer is obvious: the software that has been evolved by natural evolution, if any, must also be plagued with bugs that generate malformations, malfunctions and whose tracks must be visible as in the fossil record as in modern populations.

15.4 Conclusion

Our tinkering with numerical analysis and trickery to calculate p- and critical values has awaken and strengthened two important claims. First: intelligence of non evolutionary origin can be detected in a family of programs if new strategies that can be explained by mathematical reasoning are preferred to those naturally posited by evolution. Second: the validation of algorithms is unrealistic in

practical terms. Therefore, software is always dressed in abundant bugs that companies try to unveil and fix through beta versions and automatic feedback. This law of abundant bugs is always true for software solutions engineered by human beings. What can we say about software produced by natural evolution? The answer is obvious: the software that has been evolved by natural evolution, if any, must include representatives that are plagued with bugs that generate malformations, malfunctions and whose tracks must be visible as in the fossil record as in modern populations.

Chapter 16

Project

Joining all pieces together

349. ***Challenge.*** *Design and implement a toolbox to help yourself in own scientific or pedagogic work related to the scientific method, say, in a course of statistics. To that aim, join into a single great functional unit all those small pieces that we have developed in this course. You must add other pieces written on your own to get a package that will satisfy you to the full.* *Answer*

16.1 Toolboxes

A **toolbox** is a program whose main method is void so that it can be filled at pleasure by the User by making calls to needed methods. In practice, one fills the main method with a list of all available methods but they are muted with the possible exception of one.

350. ***Example.*** *A toolbox looks like this:*

```
//Program E350 ToolBox1
//This is how a
//toolbox looks like

package ejvol5v3p;

public class ToolBox1 {

    //*******Globally defined variables*******

    private static final int  DATA1[] = {1, 2, 3, 4, 5};

    //*****Methods********
```

```
    //Processes a string
    private static void method1(String s) {
        System.out.println(s);
    }

    //Processes a vector
    private static void method2(int Vect[]) {
        System.out.println(Vect[1]);
    }

    //Processes a string and vector
    private static void method3(String s, int n) {
        System.out.println(s + DATA1[4]);
    }

    //Manager
    public static void main(String[] args) {
        method1("abc"); //inactive
        method2(DATA1); //inactive
        method3("Hello ", 4); //active
    }

}//End of Program E350 ToolBox1
```

16.2 Succeed or perish

The elaboration of the proposed project is indeed a difficult task but to succeed is mandatory. If you accomplish it, you will sufficiently acquainted with the saturating omnipresence of interconnection bugs along any software development process. In consequence, you will be enabled with great authority to discuss the evolutionary implications of horizontal gene transfer (Choi et al, [14] 2007), which corresponds to patching software just as we are committed to do in the present challenge.

If you try to accomplish this project but desist, your failure will unconsciously program your mind as follows: “I am a very intelligent person and was unable to complete a simple task of programming. So, how can natural evolution be responsible of the genome, an example of extremely complex software, when it has no intelligence at all?”. So, failing in this project will blind you to discuss with open mind very interesting questions. Such questions have proved to be very difficult to study. So, if you cannot stand this tiny task, how do you expect to get alive after facing those coming troubles?

That is why to succeed in this project is mandatory. So, keep working until plain victory: begin with a program to calculate the mean and variance of a list of data and then add more functions as needed, say, by reading your preferred text of statistics.

Besides, your honor is at stage: at present, your enemies mock on you because one reason or another, but in five years of continuous work, they will not be able to hide the envy that you will awake in them because of the clarity and elegance of your concepts and authority to discuss them. By the way, envious people eventually might cause great damage, so be prudent, very wise and do not hurt them anyhow.

16.3 Conclusion

Our Java lab has allowed us to formulate questions of singular biological importance, such as this: "A very intelligent person suffers a lot to complete a simple task of programming. So, how can natural evolution be responsible of the genome, an example of extremely complex software, when it has no intelligence at all?". To convert this question into a fertile terrain for science is the aim of Vol II and III.

Chapter 17

Evolution in perspective

There is no perspective without a horizon

351. ***Introduction***

We have made a path through a hard lab in software design whose purpose was to devise a kit of Java programs to implement the scientific method for elementary statistical analysis. In its nude version, this method captures the natural instinct of treating extreme events in a central distribution as indicators of anomaly behavior. So hard work allows us to declare that **science is in our community a craftsmanship. It is ours, the product of our hands and our responsibility.**

While the essence of the scientific method is simple, reality is not and, therefore, we need a code of use for critical moments. Codes of use are person dependent but can be grouped in philosophical schools or philosophies. Thus, we provide here a primer in philosophy of science, the horizon that frames our thought. Next **we express our reasons to consider that evolution and the evolutionary theory are scientific. At the heart of our discussion we find the proposition that the genome is software, that evolution is a software developer and that the evolutionary theory claims that natural evolution is the software developer that is responsible for the genome of living beings**.

Thanks to the authority acquired through our hard lab in software design, use and reuse, we judge the fairness of the evolutionary theory. Next, we consider some ensuing philosophical reactions.

17.1 The skeleton of science

Science is a very complex sociological phenomenon but it has an skeleton which is conformed by the scientific method. A skeleton has no life in itself but needs commands form nerves and force from muscles. In the same way, the scientific method must be regulated by some philosophical principle to declare how it

must be used in critical moments. The scientific method and a given philosophy need altogether the force of a convinced heart to become an enlivened ideology.

352. *The scientific method*

We find a very simple procedure at the core of the **scientific method** (compare, Wudka, [96] 2004): *contrast what you see with what you think.* More operationally, we have the following steps, which have been followed all along the present volume:

1. Clearly specify what you see and also what you think.

2. Tailor a critical way of measuring the discrepancy between what you see and what you think.

3. If you notice a discrepancy that is small enough to be explained by uncontrolled factors, retain your thoughts and pass to checking another measure of discrepancy. Otherwise, reject what you think and look for a better explanation.

We have seen how this program is implemented over very simple but illustrative instances in which noise is included. The procedure is as follows:

1. Choose a given measure of the discrepancy between what you see and what you think. Measure the actual discrepancy to get D_{exp}.

2. Study the distribution of discrepancies between what you see in a virtual world fueled by your belief or null hypothesis and what you expect in that world were noise turned off. The purpose of that study is to determine what is normal and what is outlier in the space of discrepancies.

3. Judge D_{exp}, the discrepancy caused by your experimental result: if it classifies as normal, accept the null hypothesis but invent and test new forms of measuring the discrepancy between what you see and what you think. But if D_{exp} classifies as outlier, as an extreme value in a distribution of central tendency, reject the null hypothesis, invent a new theory and test it over and over.

By running this very same program over many and diverse cases, we have shown in which way making science is in the EvolJava Community a craftsmanship. Science is anymore an abstract item, it is a hand made product: science is ours. Science is our responsibility.

For our own honor we must make sure to devise a science that will be appreciated by coming generations far in the future. We consider that this is possible because by looking at ancient cave art (Visual-arts-cork, [95] 2011), one feels how nice, beautiful and powerful it is. In this regard, we long for simplicity, depth, clarity of concepts, sharply cutting tests and well chosen studies.

17.2 Fundamental problems

While the scientific method is direct and simple, reality is not. Therefore when one tries to apply the scientific method to real situations, one finds, apart from operational troubles, problems that are tied in general to some few conundrums and that eventually might be solved by philosophical world views or religious stands. The first problem is the relation of scientific propositions and truth. Others are related to complexity and consciousness.

353. *The definition of truth*

The scientific method is a method not a truth. It is a method to construct propositions that resist experimental tests. Such tests might be real, in the lab, or be the fruit of observations, such as in astrophysics, or maybe mental ones, in gedanken (German word that means imagined) experiments, or in simulations in a computer. The aspiration of every scientist is that his or her scientific propositions represent essential mechanisms of nature and those mechanisms is what we think that constitute **the truth** (Glanzberg, [41] 2009). The problem is that the relation between refined products of the scientific method and reality is not a transparent one. The very first reason for this failure is that the scientific method is error prone:

One might accept a null hypothesis that is false or reject a null hypothesis that is correct. This may happen because events that look as normal might had been produced by quite different mechanisms than those consider to explain what is considered to be usual. And vice verse, an extreme event might be the result of the very same mechanisms that explain normal values.

354. *Complexity*

The connection of the scientific method with the truth is hampered by complexity in at least two ways. The first is in relation with the emerging of effective truths and the second with the axiomatic deductive method inherited from mathematics.

An **effective truth** is a proposition that is statistically robust against tests of diverse nature. This means that for administrative purposes, one prefers to ignore violations. Robustness of effective truths increases with the development of science and at the same time to built fitter models becomes more difficult. The overall consequence is that people get used to the effective truth that are accepted at the moment and take it as a dogma. Now, dogmas hamper further insights and research.

The relation of truth with mathematics is as follows: to clearly understand what one thinks in and what one sees, one prefers to compress available information, i.e. to summarize it leaving behind redundant information. A preferred way to do this is by formulating a set of axioms and to deduce everything using the rules of logic. The problem is that even slight complex universes cannot be exhaustively modeled by any set of axioms. In plain words, one must get content with partial explanations else with full but self contradictory ones (Ebbinghaus et al, [25] 1984; Myers, [72] 2010). For this reason, a collage of effective truths

of restrained scope of validity tends to dominate science: we do not have fundamental science that encompasses all everything, instead we have quantum mechanics, physics, chemistry,...

355. ***Consciousness***

The scientific method contrasts reality with ideas. While this is the simplest cartoon of the scientific method, it is in strict sense false. Rather, the scientific method contrasts ideas with ideas: what we see is not reality but an idea about reality. The reason is that there exists between consciousness, the element that sees, and reality a very complex machinery responsible for perception. That machinery elaborates what is perceived by the senses and constructs an artificial reality for just one observer: our consciousness (Rodríguez, [86] 2007). For instance, colors perceived by our consciousness do not exist in nature, instead they appear as construction of our visual cells and their neural connections.

Consciousness is for science a mystery: nothing in the brain or in robotic systems helps us to predict that it must exist. All we need is a system for observation, evaluation of data and decision making (Horst, [49] 2011). This does not demand consciousness such as you and me perceive it in our own inner beings.

The net effect of all these problems is that even to this very day there is no agreement on what reality is.

17.3 Fundamental philosophies

The fundamental positions to define reality can be organized if we recognize that the most delicate of all problems to define reality is that of consciousness.

356. ***Idealism and materialism***

A person knows that his or her consciousness exists and that it has sensations. **Idealism** proclaims that this proposition is true and it is at the base of every knowledge, so every knowledge is subjective. **Solipsism** complaints that apart from the certainty that consciousness exists, every thing else is mere speculation made in the consciousness and for the consciousness. **Materialism** replies that with care and wisdom one can single out those sensations that reflect a reality that is not created by the consciousness, that exists outside of it, that is composed of matter, and that is almost fairly registered by the perception machinery. So, **matter** is what exists outside the mind.

357. ***Materialism and science***

Matter is declared as the target of **science**, so it is is built over **objective materialism**, which is defined by the next three items:

1. We have some sensations that correspond to carriers of other consciousness, to persons.

2. As idealism claims, every reality is subjective and is no more than a pattern elaborated by the brain, personality and ideas learned from other persons.

3. One declares that certain knowledge is objective when all persons agree on it and when it can be perceived by material instrumentation. Example: every person agrees that there is sun in the sky, a fact that can be registered by photographic cameras. So, the sun is objective, it is part of reality. The correct description of reality is thus possible and is the aim of science.

In this way, official contemporary science considers that the physics and chemistry of books and journals are all to reality and that science is just an investigation of how their already known constituents interact. Nevertheless, modern science has tenets that are not objective and that fit well in a fairy tale:

- It is taught by the Big Bang cosmology and its corrections that our universe was created by or passed throughout an explosion that gave rise to all that exists (Koelman, [59] 2010). Presumably, everything appeared from the quantum vacuum (Puthoff, [81] 1990?; Rafelski and Müller, [83] 2006). Nevertheless, it never has been witnessed that even an electron has appeared from vacuum. Quite to the contrary, huge amounts of electric energy, sufficient to feed a whole country, are needed to achieve in the lab a very crude simulation of a tiny big bang.

- There are over two million species in the Earth (Wolosz, [57] 2011) and according to modern science all of them appeared by evolution. Nevertheless, even with genetic engineering, we had not been unable to go beyond races and subspecies. This is very shameful.

Now, how is it possible that scientists include fairy tales in their beliefs?

The reason is that materialism, the philosophical basis of science, is a **world view**, i.e. it is a theoretical body to create and organize knowledge in a complete, sufficient, all encompassing, all judging and jealous form. And it happens that every world view, of whatever nature, to be complete needs fairy tales. So, the culprit is not science but logic or maybe something more abstract that causes that mysteries should be killed by still greater mysteries.

358. *Spiritualism and religions*

Materialism must battle for surviving against other philosophies in the open market of ideologies, among which a very popular one is built around the belief that there are spirits apart from matter. In fact, many people consider that there is a spiritual reality that is perceived by children, almost all women, some men, old persons, drunkards, drug-addicts, warlocks, witches. That reality appears in every anthropological study of cultures of all times and places around the Earth

and is captured by the great religions. The following story is surely known to the Reader from a direct witness:

A girl lays down and sleeps in her bed but suddenly she is awaken by the deformation of her bed as if a person would had sit down. But there is nobody. If the girl reacts peacefully, she feels that an invisible being lays down by her side. The ensuing relation of that being with the girl may infringe on her hard psychological troubles as exemplified by the book of Tobit, found in a Bible with Deuterocanonicals. It refers a narration about a girl that is pray of a terrible suffering caused by a paranormal bad being, a demon.

The spiritual reality is composed of gods, demons and ghosts -the spiritual part of every person and that continues to exist after dead. The ideology that pretends that spiritual experiences are real and not hallucinations is **spiritualism**. The philosophical stand of spiritualism might be **subjective materialism** because it considers that consciousness is furnished with more channels for the detection of reality than those that has been discovered or accepted by science. By abuse of language, the word **materialism**, when it goes alone, means objective materialism.

Science is dominated by the present form of materialism because no new form of mater has been unveiled. Say, the human perception of paranormal entities has not derived into a repeatable procedure to register ghosts. This failure has two possible causes: there is in paranormal phenomena nothing apart from hallucinations else we have not been smart enough to register ghosts, which indeed exist. So, the position of science is:

If you have a proof that ghosts can be objectively registered, show the method to me and I will greatly reward you.

Meanwhile, the biological role of the spirit is not considered in science except in medicine, where some doctors use religions as placebo but others seem to know that no placebo is powerful enough to explain the miracles they sometimes see. On the other hand, the faith in God appears as object of study in anthropology and sociology (Easa, [26] 2017). Its permanent importance is derived from the fact that religion and democracy (terrorism included) have many folds of crucial interaction.

Religions are spiritualist currents that teach that persons can be happier both in this world and after life if they organize their lives around a relation with gods. Every religion seems to say: hold tied to God and be happy now and forever! But, behold, many gods demand the sacrifice of human beings (Ashliman, [5] 2009). In reaction to this and other defects, hard scientists are atheistic. This position is strongly reinforced by the belief that evolution is the correct and complete explanation of our existence.

Many religions aspire to be world views. As such they also have fairy tales and contradictions. Let us illustrate this problem with an instance taken from

the Christian religion: one of its tenets is that *God is with us*. For many people, this cannot be more than poetry, a nice fairy tale, a very tender one. But, if someone considers that God is really with us, he or she must listen to a complaint that Job makes onto God: *How long will you not depart from me, nor let me alone till I swallow down my spittle?* (Job 7: 19). So, the dream of a God with us is not necessarily a remedy for our problems but possibly the most terrible one. So, God with us is *a fairy tale* else a contradiction.

The preferred battlefield for religions vs. science is the problem of the origin of life and of species.

359. ***Creation vs Evolution***

Many religions teach that we exists thank to the work of God. They preach **creationism**. By contrast, science says that we appeared by a natural process, by **evolution**.

While Hinduism is akin to evolution, the creationism in Islam is simply sketched but in Judaism it is developed so extensively that it can be taken as fundamental: G-d created us and will judge us and to that aim He has a command for holiness together with a call to a reconciliation with Him by means of an offer for the sin, which is the sacrifice of a bull.

But on the other hand, if the man appeared on the face of the Earth by the effect of randomness and self-organization, what right does somebody have to judge me? None. Therefore, if the evolutionary theory is correct, it dismisses the authority of Judaism, G-d gets redundant, His great glory will vanish away and the historical roots of the nation will fade into a collection of stories targeted to reinforce patriotism. So, for Judaism, derived religions and for all those that expect a judgment by G-d, the evolutionary theory is worth a careful examination to clearly determine whether or not the evolutionary theory explains our existence.

This task will be accomplished when all, scientists, lay people, school teachers, religious people, detractors and future generations agree. We are obviously very far from achieving such a clarity. More to the point, if you want clarity, stop murmuring and make it by yourself. In this regard, our point of departure is a faithful model of evolution that we enter to discuss.

17.4 The evolutionary theory

Our purpose is to contrast what we see with what we think. This can not be done without the help of simplifying models that abstract main ideas and postulate an adequate reference frame. Our proposal to discuss the evolutionary theory is a natural and fair lecture of the genome and its evolution.

360. ***The genome is software***

The **genome** is the genetic material that conveys inheritable information. The proof that **the genome is software** is straightforward:

Computers are machines that obey specific verbal orders, the **software**, thanks to a code that translate verbal orders into commands for their processing units. In its turn, the genome contains verbal instructions to synthesize nucleotide chains, some of which are edited into regulatory units while others go to the ribosomes to be translated into polipeptide chains. Translation is made using a **genetic code** that associates codons to amino acids (Zien, [97] 2000?). So, the ribosome functions like a processing unit of a very special type of computer.

361. ***Evolution as a software developer***

Mutation is the first ingredient of evolution, it is simply the modification of software. Those modifications can be made in artificial software by humans or can be automatically programmed. In nature, the genome is mutation prone: nucleic bases can adopt different conformations, some of which can induce errors in pairing, the elementary event at the root of molecular pattern recognition, and so information might change over duplication or transcription.

The products of mutation might eventually improve function in artificial software and surviving in nature. It is inspiring to imagine what many mutations in tandem might achieve. The overall phenomenon is **evolution**. Because modified software eventually might enable a new individual, evolution actually classifies as **software developer**.

362. ***The two stages of evolution***

Evolution can be understood as a process using tinkering over a given elements to iteratively improve a function. With this abstraction, most optimizing activities by human beings or nature result to be framed at evolution. In general, tinkering happens over a family of objects or things, which conform the **combinatorial basis**. So, evolution is composed of two stages. The first is the design of the combinatorial basis and the second is the actual evolutionary process along which the elements of the extant combinatorial basis are combined to produce better and better outputs.

Biological evolution is an automatic mechanism that implements that abstract concept. This has been done at least twice. We have in first place a combinatorial basis composed of the more than 20 amino acids that appear in the diverse genetic codes (Elzanowski and Ostell, [30] 2008). In second place, we have a combinatorial basis of modules, of pieces of DNA that play the role of methods, procedures of subroutines in our computer programs. This hypercombinatorial basis appears in the bacterial genome, the library of genetic modules. For instance, some genes of the human brain are found in bacteria but not in the chimp (Salzberg, [89] 2001).

17.5 Evolution and science

Every one likes to test what one beliefs in, but in science this is done systematically. To test the evolutionary theory, we have clear concepts, questions and predictions.

363. ***Is evolution scientific?***

We look at evolution with simplicity and determinacy: **Evolution is a process in which software changes.** It is an undeniable phenomenon in nature and the very essence of the trade of software development.

We declare the study of evolution as a scientific enterprise and in doing so we mean three things. First, that evolution, natural, artificial, simulated or in software development, is subject to the laws of nature no less than rain and no more than a sunrise. Second, that this phenomenon is so important for modern culture that it is worth a very serious investigation. Third, that we with pleasure commit ourselves to an extensive and rigorous study of that phenomenon.

364. ***The evolutionary theory***

Given that the genome is software, that evolution is a software developer and that evolution exists in nature, a question naturally springs: Is natural evolution the software developer that is responsible for the genome of living beings? We reformulate with this question our own understanding of the **evolutionary theory** of the origin of species. We claim that this vision will remain valid and actual until eternity.

Is the evolutionary theory scientific?

Yes: we promise that we will make of this theory something deliciously scientific, with the type of science that must be obeyed by philosophy and religion. We have three reasons to assure this:

- We have so huge and exasperating volumes of data about living beings and their genomes that the evolutionary theory cannot be a part of philosophy. It must be decided in within science with the nude scientific method.
- We have a very generous laboratory on evolution everywhere a person is inventing and testing a receipt or developing a computer program. But, beware, software contains instructions to synthesize a recursive function, so computer engineering is just a chapter of engineering whose purpose is to produce receipts for the synthesis of the most varied products, be they goods, services, processes or computer programs. Now, any man, being homo faber, is a full time experimentalist in the difficult art of producing receipts. In this regard, women must be looked at with respect because as a rule they enjoy experimenting with cooking receipts. Actually, it is difficult to single out a human activity that is not related to a receipt of some sort. That is also why every person is a sufficiently authorized judge of any appropriately formulated study about evolution. So, make sure that your products resist their criticisms.
- We have in computing science a very refined theoretical frame for this type of discussions.

So, the evolutionary theory is scientific. The price that we must pay in making such declaration is that the evolutionary theory might be false. In this regard, we consider that the vision that *the genome is software and that evolution is a software developer* is both necessary and sufficient to decide the fundamental questions of evolution and very specially to decide whether or not the evolutionary theory is false.

If we admit the possibility that the evolutionary theory might be false, we need a reason to present it as worth of study and once we have done that, we must ask ourselves: if the evolutionary theory is scientific, where are those predictions whose testing will be accepted by both believers and detractors, by you and me, as fundamental and determining ones?

365. *The worth of the evolutionary theory*

One can formulate an evolutionary theory following the next procedure that gives to the ideas of Linnaeus (Fossil Museum, [36] 2011) an evolutionary flavor:

1. Species are described according to their characteristics. Actually, a well chosen sample of descriptors, say, of DNA subsequences, may suffice.
2. Species are organized in a tree taking care of similarities and divergences and of the cues giving by the fossil record and biogeography.
3. The resultant tree is postulated to reflect a story of descent from simpler forms to more complex ones and the tree is called phylogenetic.

We see that this formulation makes of the puzzle of the diversity and grandiosity of the species a nice and recognizable pattern (Biologicalphysics, [8] 2013). This summarizing work fits in the very essence of hard science. Now, the number of species might be over two million and when one sees them in a phylogenetic tree, one gets the feeling that the evolutionary theory has two million proofs. Nevertheless, that sensation is illusory: **a tree by itself is no proof at all**. In fact, similarity trees can be constructed for any set of objects, say, machines, computers, cloths, species... A tree is converted into a theoretical weapon by the postulate of relation by descent among its nodes. And here is where science begins properly:

The task of science struggles between to options. First, to convert the postulate of relation by descent into a proposition that must be robust against the most varied tests. Second, to reject it. In this regard, the most important and immediate goal must be the formulation of a mechanism that might eventually explain or, in certain sense, recreate the phylogenetic tree. The glory for this hallmark of science rests upon the shoulders of Charles Robert Darwin and of the society that educated and supported him. With minor corrections (research), the presentation of the mechanism of the evolutionary theory made by Darwin himself (Darwin, [17] 1859) is quite modern:

It is interesting to contemplate an entangled bank, clothed with many plants of many kinds, with birds singing on the bushes, with various insects flitting about, and with worms crawling through the damp earth, and to reflect that these elaborately constructed forms, so different from each other, and dependent on each other in so complex a manner, have all been produced by laws acting around us. These laws, taken in the largest sense, being Growth with Reproduction; inheritance which is almost implied by reproduction; Variability from the indirect and direct action of the external conditions of life, and from use and disuse; a Ratio of Increase so high as to lead to a Struggle for Life, and as a consequence to Natural Selection, entailing Divergence of Character and the Extinction of less-improved forms. Thus, from the war of nature, from famine and death, the most exalted object which we are capable of conceiving, namely, the production of the higher animals, directly follows. There is grandeur in this view of life, with its several powers, having been originally breathed into a few forms or into one; and that, whilst this planet has gone cycling on according to the fixed law of gravity, from so simple a beginning endless forms most beautiful and most wonderful have been, and are being, evolved.

We clearly see that the evolutionary theory divides its essence into poetry and science, so it is indeed the type of science that one expects to live for long.

Let us pass now to predictions and their testing.

366. *First mandatory prediction: right!*

The number of species surpasses two million: they are too many! Therefore, if we want the theme of the origin of species to be included in science, we must predict that it must be too easy both to modify a species and to make a new one.

Genetic engineering shows that one can modify extant genomes to produce variants that are viable in spite of rather great modifications, be they deletions, insertions or correcting patches (Epstein, [31] 2004). Thus, the (incipient as yet) success of genetic engineering is a clear demonstration that the theme of the origin of species is part of a technology in progress and therefore of science.

367. *Second mandatory prediction: opps!*

An evolutionary reading of biogeography (Fitch, 2011 [34] ; FAO, [32] 2011) shows that the arising of new species in nature is also too easy, too common. Therefore, we must predict that it must be very easy to experimentally caught evolution on the fly:

- If one seeds a species in a new habitat, the arising of new species with new bauplan must be immediate.
- If a new habitat arises, it must be colonized by extant species that rapidly shall evolve into new ones with abundant morphological changes.

- If one makes an experiment in the laboratory, one must produce deep speciation quite easily.

These questions are so important that they have spanned the whole field of **experimental speciation**. Since the whole world believes in the evolutionary theory, one must expect from this field of science to be among the more developed and expanding. But this is false: if you want it to become into existence, create it by yourself. You can possibly find some cues in experimental evolution (Dettnan, [21] 2007; Reznick, [84] 2011).

Our definitive failure in the experimental creation of species compels us to think that the evolutionary theory must be polished as a fine diamond and that we must get prepared for a very long work. The right place to begin with is by giving a look to the proposed mechanism behind the postulated evolution of species.

368. ***Third mandatory prediction; ooh!***

The fundamental mechanism of the evolutionary theory reads as follows:

- Fathers have offspring with genetic, inheritable variation that must struggle for surviving and reproduction.
- Since some offspring will do better than others, differential surviving and reproduction appears as an emerging feature of the interaction of the genomes of individuals with their environment. That emerging feature is **natural selection**.
- Thanks to natural selection, the population gets more and more adapted to its environment as generations pass by.

We see that natural selection appears almost as an abstract feature, actually an emerging one. In consequence, a mandatory prediction is that evolution must run wherever we change natural selection by artificial one, which is plainly under our control.

Our prediction is correct as it is well known since ancient times: the invention and practice of agriculture is tied to an artificial selection for productivity, homogeneity of time of harvest and good cooking qualities, among others. By the same token, domestic animals were in general spontaneously selected for manageability and productivity.

369. ***Fourth mandatory prediction: a bull's eye***

The evolutionary theory is plainly mechanistic and its sufficient ingredients are: reproduction (with mutation and recombination), variability in the offspring and a mean to select the fit from the unfit. Therefore, another mandatory prediction is that it must be possible to device **evolutionary environments**, i.e. artificial and well defined worlds in which evolution can be implemented

and run as a tool to solve optimization problems in which artificial selection is defined by the function one wants to optimize.

This prediction has been fulfilled with flying colors: evolution is used in our modern world as an ordinary tool to solve every kind of problems if only these can be reformulated as optimization problems that admit approximate solutions that converge to an acceptable solution (Holland, [47] 2011). For this reason, **evolution is anymore a matter for speculation: think whatever you want about evolution, but, please, test your thoughts at least in a simulation**.

370. ***Fifth mandatory prediction: another hit!***

If the genome was developed by natural evolution, and given that the genome is software, we must conclude immediately that evolution is capable of developing software. So, we predict that we can use simulated evolution to develop software. This prediction is terrific because developing software is one of the most difficult enterprises of modern world. Actually, there are some few gurus that have made of this challenge a profession for their lives, the very first of them was John Koza ([61] 1996). This profession is called **genetic programming**. He works in his monastery with a computer facility with 1000 processors and has gotten some patents (Koza, [62] 2007). The molecular equivalent of genetic programming is the new discipline of artificial DNA-computers, the first of which was made by Adlemann ([1] 1994). Natural DNA-computers also exist in nature, say, in the immune system: from a combinatorial basis of genes, immunoglobulins are evolved for specificity and affinity to foreign molecules (Darnell, [16] 1986).

We made in Vols II and III of this series an attempt to popularize genetic programming. Nevertheless, our success was incipient. More is wanted. Anyway, we consider and accept as a challenge that the mandatory prediction that simulated evolution can be used to develop software is fulfilled. In case of doubt, a good option might be to experiment with the free material provided by Meffert and Rotstan ([68] 2011).

371. ***Sixth mandatory prediction: horror!***

The experience of every one that has engaged in developing software is that committing bugs is the closest reality to the dealt. So, we formulate a mandatory prediction: there is no software developing without bugs. More operationally, if someone claims that he or she developed a piece of software, he or she must be able to refer his or her personal experience with bugs, narrating a vivid evolutionary process, in which a succession of modifications of goals and of correction of bugs that generate more bugs was followed. Example: the Mozilla project, registered its bug number 640000 thanks to Igor Velkov on 2011-03-08 14:31:56 PST (Mozilla, [71] 2011). Actually, the omnipresence of bugs is something that hurt the eyes directly (Lanza, [64] 2008).

The situation is similar to a race of trotters across a desert. If one stands at the final line, it is very easy to distinguish true competitors from false ones.

Why? The reason is that we have a mandatory prediction: long trotting produces abundant sweat. No sweat, no trotting. Let us notice that our prediction is mandatory no matter whether we have a race with a finish line or just we are dealing with purposeless wandering across the desert.

By the same token, we have a **mandatory prediction** to test the evolutionary theory of the origin of the species: **there is no software developing without a generous evolution towards perfection, which must be evidenced by a rich history of bugs and ensuing corrections that generate more bugs**.

372. *Falsification*

A **theory is falsified** when one of its important predictions is found to be false. In regard with the evolutionary theory we have:

The number of extant species is over 2 million. So, if the evolutionary theory of the origin of species is false, it must be immediate to unveil its lies for we have so high level of raw material as no judge has ever witnessed. Now, we extend an official invitation to consider the next argumentation as a sharp and sufficient evidence of the blatant falsity of the evolutionary theory:

Extant life is an epitome of both perfection and complexity, so if it appeared as a result of natural evolution, a very rich evidence of evolution towards excellence must exists both in the record fossil and in extant populations, which must reveal itself in every sort of malformations and malfunctions. Since such mandatory evidence is lacked, we conclude that present life is obviously not the result of natural evolution and that the modern evolutionary theory of the origin of species is scientifically witless.

We find this accusation so simple, sound, clear cutting and mortal that we contend that every other attack against the evolutionary theory is less universal.

373. *Seventh mandatory prediction: Yes!.. or no?*

Experience in software design clearly shows that there is a huge variability in the solutions to any given problem. For instance, if by whatever reason one loses a program and one is forced to compose a new solution, one always ends with another program that might be strongly different than the first. So, we predict that the same must happen in nature: every function must be fulfilled in many diverse forms.

The celebrated example that illustrates how this prediction is dramatically fulfilled is the eye (Ings, [50] 2008). Nevertheless, our prediction is not for a single instance: it must be a universal characteristic of life that must be observed across all functions. In this regard, the Author considers that there are also clear and abundant falsifications to the aforementioned prediction, say, in regard with conserved sequences. The usual explanation that they correspond to structures with maximal fitness of absolutely necessary functions has been tested in one

experiment and found to be wrong (Ahituv et al, [2] 2007). So, at present time this implies a falsification of the seventh prediction.

374. ***Eighth mandatory prediction: ouch!***

The evolutionary theory claims that over two million of species arouse by natural modification of the software that they contain. This forces us to predict that modifying software to design new one is quite an easy task. Besides, experience shows that every software development is the result of a run of modifications over modifications. Henceforth, every development of software is just a directed evolutionary experiment that go downstream. Therefore we predict that software developers must be among the worst paid people of the whole planet and that it is illusory to dream getting rich or even to earn a life by mounting a company to develop software.

375. ***Falsification***

Exercise.

376. ***Ninth mandatory prediction: what a embarrassment!***

The mystery of the genome is that it can be tamed to serve the purpose of artificial selection with fitness functions of the most varied nature in such a way that the corresponding problems are easy for evolution. So, we can produce horses appropriate for races, for hard work, for battle, for home labor, resistant to heat and drowsiness, and so on. And the process of selection is equally successful in every living being from man to bacteria and for the most varied functions.

This tremendous versatility of the natural genetic code is called **evolvability** and is what wonders us. The literature recognizes that the natural genetic code is one in a million but with regard with some informational properties (Freeland and Hurst, [38] 1998). These characteristics are very important but not of first rank. It is as saying that we like spatial navies because they protect people form rain and dust. So, let us emphasize that **the power of the natural genetic code is that it enables a maximally evolvable style both natural and artificial**.

On the other hand, the evolutionary theory is the minimalist theory that converts the fact of the evolvability of the genome into a self explanatory event. Since we can equate software development to software evolution and given that the number of species might arrive to 70 million, and if we suppose that the evolutionary theory is correct, we must predict that the design of evolvable programming languages must be a game for kids.

377. ***Falsification***

If someone considers that the evolutionary theory resists the ninth prediction, let him or her know that the industry of digital games has billions of dollars for a developer capable of composing a language that resists the tremendous complexity of making artificial realities that are always evolving at high

pace (Sweeney, [94] 2008). In any case, artificial realities seem to be inescapable, say, to model the arising of biological forms. So, our community must get prepared for lengthy and heavy duty research programs. We consider that JavaFX is enough for our needs.

378. ***Tenth mandatory prediction: vai me!***

It is marvelous that every developer has its own personal style. A style arises because the mind finds a good strategy to solve a very complex optimization problem, which is how to design software. But evolution has no mind, therefore it has all possible strategies at once but visits the most populated regions in style space. There is indeed just one great region, which is well known to everybody: it is maximal disorder. In fact, extremely hard work is necessary to produce software that could be understandable and evolvable. Thus, we predict: the style of the genome must be dominated by disorder.

379. ***Falsification***

The very fact that genetic engineering exists since long ago shows that the genome is quite easy to read and understand. So, we have here a very elementary and fatal falsification. To reinforce it, let us list some properties of the style of the genome:

- The genome is structured: architectural properties are appreciated at once in genetics if we recall that prokaryotes and eukaryotes differ by the style of design. In fact, the genome of the prokaryotes resembles the style that was permitted by ancient programming languages: instructions must go one after another and this is all to structure. By contrast, the genome of the eukaryotes resembles programs in modern languages, which allow for highly structured products with units and subunits that are ready for use and reuse (Darnell et al, l.c., [16] 1986), a property that lends itself to evolvability towards complexity.

- The genome is documented: the production of tryptophan is encoded in many bacteria by a group of genes that are used, or transcribed, together, so they conform an operon structure. It happens that this operon also encodes for a very short peptide sequence that is rich in tryptophan. Something similar happens with phenylalanine and hystidine (Darnell et al, [16] 1986). From our perspective, this looks as software documentation.

- The genome obeys the **function paradigm,** according to which any procedure or method in a program is the implementation of a recursive function. This paradigm is naturally evolvable, reusable, since functions can be concatenated, composed. It is the dream of clean computing science, a dream that is transformed by every developer into a nightmare because of global variables and by the entanglement that ensues the abundance of mutual calls among methods or procedures. So, one might think that the functional paradigm cannot surpass the realm of abstract mathematics and some few applications. But that is false: let us argue that the

genome obeys that paradigm without one single exception and not by some constraint but by a choice of style. The defense of our proposal is as follows:

In normal conditions, enzymes are targeted to catalyze those reaction whose activated complexes dynamically fit the active site of the enzyme. Now, physic chemical laws cannot interdict the existence of various active centers in the same enzyme. In fact, enzymes exist with various active centers. But, enzymes have modular design and no module has been found with more than one active center.

If we additionally take into account that enzymes are in general highly specific, the overall result is that enzymes catalyze, in general, only one reaction under normal circumstances. That is how pathways arise and with them the biological identity of each species. So, the biological paradigm reads: *clearly specified input that evokes unique and clearly specified products.* This is distinctive of enzymes but, by contrast, free reactions produce many products over the same reactants. So, life obeys precisely the function paradigm but applied not to software but to their enzymatic products.

One can immediately claim that the function paradigm became the actual style of the genome thanks to evolution. That option might be appropriate for the international literature but not here in our community. By contrast, we must proceed like this:

1. Achieve a synthesis of enzymes with two or more active centers in the same module. This can be done in silico (in computer simulations), in vitro or in biotechnology with or without the help of evolution.
2. Prove that in a population relying on enzymes with various active centers on the same module, the winners are those that rely on enzymes with just one center on every module.
3. Prove that the evolutionary process leading to the dominance of enzymes with one active center per module is instantaneous in geological time and so it left no tracks neither in the fossil record nor in remnants in actual populations.

17.6 The reaction of science

Our insight into the evolutionary theory has allowed us to make important predictions, some of which result to be correct while other are crucially false. The form as scientists might react to this result can be diverse, say, some will recur to philosophy to protect the evolutionary theory while others will make proposals to circumvent problems.

380. ***The protection of materialism***

We have elaborated a mandatory prediction in regard with bugs about the evolutionary theory that resulted to be false. To be a **mandatory prediction** implies to get inescapably true under specified circumstances and with abundant evidence. In this regard, science has a criticism that we can understand if we pay attention to the story about the trotter:

One can not deny the possibility that a trotter might violate the prediction of abundant sweat. The reason is that the trotter is a thermodynamically open system that is correctly described by statistical mechanics (Sklar, [92] 2009). So, in strict sense, the prediction is not mandatory but only highly probably. The same happens with evolution: given that matter comes in the form of atoms and that DNA is a finite chain of molecules, there is a very small probability that correct organizations might appear under physical conditions by mere randomness and in the first trial. Anyway, such probability is so small that philosophy is the only actor that can make some use of it. Let us see how:

According to science under the objective materialism, apart from hydrogen, carbon, nitrogen, phosphorus, bosons and fermions, nothing else seems to exist in nature. Therefore, the scientific explanation of our existence must resort to these building blocks no matter how probable or improbable invoked events might be. So, the fact that we exist is a direct and simple proof that events of exceedingly low probability can happen in nature. And under the present form of materialism, this is all to it.

We can accept and understand this position as philosophically valid if we announce the **principle of separation of science from philosophy**: philosophy is separated from science and goes on top of it regulating the acceptance and understanding of scientific facts together with their interpretations. This principle implies, for instance, that probabilities in materialism never will be sufficiently small that a Creator might accepted, invented or predicted.

381. ***Cosmology comes into help***

The problem of low probabilities has reached cosmology. The whole discussion has been aligned along the **anthropic principle** that says in one of its many versions that the very fact that we exist poses extreme bounds on the physics and characteristics of our near universe (Stenger, [93] 2011; Kane et al, [55] 2000). It seems to us that proposed solutions to the problem of small probabilities amount to a change of insight:

Do not isolate from the rest of the world the miracle that a given person won the lottery. Think instead of the lottery itself that creates a miracle every week.

This principle functions like this: suppose we are playing with a coin and need a series of 100 heads. What must we do? Since this event is very rare, we

must throw the coin many, many times that at last the desired run of heads will appear. In this regard, some ideas to examine are the following:

- Evolution runs on molecular events while the Earth and its seas are of astronomic size, so, abundant gambling is guaranteed. This relation is all some scientists need to accept that evolution is the correct explanation of our existence.
- Our Solar System is not special at all in the Universe, rather many such systems with Earth-like planets can exists (Nasa, [73] 2011).
- Our universe is not unique but is just one trial among many, such as it is envisaged by the inflation theory of cosmology (Linde, [65] 1994);

Apart from winning the lottery, people can get very rich if they find an ore deposit of, say, gold. Precisely, Kaufman([56] 1995) argues that our universe, such as it is, comes with rich ore deposits in the form of mundane laws of self-organization that make of life a mandatory event. So, his theories are targeted to explain without Darwinian evolution the freezing of highly organized systems, say, of life. He promised in his book to achieve experimental results by 2000. But none exists. Why? Why does this stupendous program have not sparked thousands of experiments? The only reason seen by the Author is that everyone knows by instinct that life is not mandatory and that, therefore, a flaw must exist in his theories.

382. *Generic predictions*

If you like the idea of using cosmology to convert events of low probability into necessary ones, keep in mind that this program is impossible to fulfill in some cases. Think, for instance, of the following saying:

A liar falls down sooner than a lame.

This saying amounts to a sort of statistical prediction: lies are detectable and easy to unveil. Actually, our world is filled in lies but to unveil them is just a matter of investigation (warning: a cost might be payed, say, your life, your glory, your employment, your friends, your connections). Nevertheless, there is some probability that a lie passes some few tests. Now, the aforementioned saying is true in any universe no matter what its fundamental physics might be if only it is sufficiently complex to admit intelligent life (every universe admits the concept of truth, the most near one is that the universe exists).

Let us define now that a **prediction is generic** when it is valid in all possible universes. A **falsification is generic** when it falsifies a generic prediction. We have now that our falsification of the evolutionary theory deals with a generic prediction: if evolution is the cause of a complex being such as we are, then it must have left in any universe a clear history of bugs whose fixing created more bugs.

The reason for our prediction to be generic is that the laws of physics must be preferentially simple (Schmidhuber,l.c., [90] 2000). This simplicity gets apparent for our world in the Feynman's diagrams associated to the fundamental interactions (Kaiser, [54] 2005; Krešimir, [58] 2011). So, fundamental laws can possibly enclose the high levels of information of some few genomes but not of all of them. Lacking ones demand a process of construction, an algorithm that from the stand of fundamental physics will be classified as complex. So, if those algorithms were eventually generated by self-organization and evolution, a clear track of committed bugs and their fixing that gave rise to more bugs must be apparent in extant populations, and possibly, in the fossil record, if any.

We conclude that the law of abundant bugs in any evolutionary process that battles complexity is a very robust one and the announced falsification of the evolutionary theory is generic. So, let us ask again: Does this implies that science will reject it?

383. ***The great mother***

To clearly understand the relation of science with the evolutionary theory, let us think of the next sociological role of that theory: many, many families have made an easy and luxury living over 150 years thanks to it. So, the evolutionary theory is a great mother, in a literal sense. And, as everyone knows, one readily tolerates in own mother a fault and not only one but as many as might appear. It is also typical that every mother is the best mother of the world: according to the international literature, the theory of evolution is the submit of human wisdom.

Therefore, one might predict that science has arrived to a dead end and that it will remain so until the dead of all descendants of the evolutionary theory (Kuhn, [63] 1969). Nevertheless, this prediction is not sure because science is not that simple. Science is a seven headed monster, as the next paragraph shows.

384. ***Hackers instead of God***

Many scientists make fun of materialism and formulate questions as the next one due to Einstein: "What really interests me is whether God had any choice in the creation of the world" (Linde, l.c., [65] 1994; Howard, [48] 2010). Some insights leaves few room for that (Kane et al, l.c., [55] 2000) while others explicitly invite us to consider a creator, that eventually might be a hacker in some supra-universe (Schmidhuber, l.c., [90] 2000). The difference between being explained by God and been explained by a hacker is that the last will not judge us while God will. But, apart from this, is there another reason that motivates scientists to posit a hacker on top of us? Yes, let us see it.

385. ***The digital universe***

From the very gestation of computing science, around 1930, the problem of modeling our physical world as a digital entity was paid serious attention. A

continuous effort by many important people and their students has produced over the ensuing decades an impressive volume of works, ideas and intriguing questions that group themselves in what is called **digital physics** and its diverse branches. Now, some people working in the field feel strong enough to consider, suggest and even claim that our real world is simply an artificial reality, a sophisticated simulation in a computer, created by somebody in another universe in one of his supercomputers (Fredkin, [37] 2001).

The next question is: why does digital physics need the existence of a hacker? Or, if our world is digital, why must it be a virtual reality and not a real one? A possible answer to this question might be worked along the following lines: if our world is digital then it is the unfolding of an algorithm, and our experience in the construction of digital worlds shows that expertise, fresh ideas and hard work are needed to succeed. Now, algorithms also can be devised automatically, but a long battle with blind alleys and bugs is necessary. So, a prediction results in this last option: a universe that is dominated by local interactions must have diverse physical laws across it, a fact that possible might be detected at the far reaches of the universe. This is not what physicists accept in their ordinary work. So, the uniformity of the universe corresponds to a style of design of a developer: a hacker must exist.

17.7 Evolution in the open market of ideologies

We have declared the evolutionary theory to have been irremediably falsified by the extreme complexity and perfection of life without the evidence for an evolution of perfection through a fierce battle against bugs. As we have remarked, the philosophical solution to this problem is elementary. But the philosophical basis of science also can be questioned. In this way, the evolutionary theory must compete with other beliefs in the open market of ideologies and religions.

386. ***Merging evolution with other ideologies***

Evolution in the public domain is rivaled by diverse forms of creationism, at whose center we find an intelligent being, God or aliens. But on the other hand, no one can deny that the evolutionary theory is strong and appealing. Thus, we expect that many informed people will merge the evolutionary theory with other ideologies. Most possibly, we have a theory for each human in the Earth, a total of 7000 million theories. Some interesting for us formulations might be the following:

- Species arouse by evolution. It is the **ordinary evolutionary theory**.
- Living beings and species spontaneously arouse by evolution while the combinatorial basis on which life is built arouse by prebiotic processes, by randomness and self-organization (the formation of ordered structures or processes by the spontaneous obedience of natural laws). This is the **extended evolutionary theory**.

- Living beings and species spontaneously arouse by evolution but the primordial combinatorial basis was imposed from outside. Say, primordial life came from the outer space, or from a seed by aliens or by the Creator.

- The world exists with a tendency to evolve towards complexity and this tendency is enabled by appropriate forces. Thanks to this, life and species appeared: **Alchemy and Lamarckism**.

- God directly created some species that were next helped to evolve into the various million of species we see today. This formulation is somehow popular among Catholics as an ideological influence of **Theilard de Chardin**.

- Some species were created at the beginning that next evolved into the various million of species we see today. This was the **early Darwinism**.

- An intelligent being created all species directly that were given to evolve but the ensuing evolution has been restrained to micro-evolution and subspecies. Thanks to evolution all humans are fairly different, a fact that is marvelous. This is **Creationism**.

- A Creator used in His labs the great power of evolution as a tool to design the species that next He created at the earth. Species were created with the power to evolve, a matter that we can investigate. This is an option for someone that refuses to let the evolutionary theory die.

- No tool is the best for everything and, by the same token, evolution is not good to battle every aspect of the complexity of life design. So, living beings and species were designed by intelligent beings, say, God or aliens, at outside labs using multiple tools of engineering. Next, that life was created somewhere in the universe and possibly transported to other regions. While life is endowed with evolution, its power is directed to achieve adaptation to local environments but evolution into further phyla is dubious and has not been proved. This credo is appropriate for those that belief in aliens.

The aforementioned positions are just some options as one can plug evolution into own personal credo. Modern science has done its best to prevent such poisonous mixes. Nevertheless, it is readily verified that its efforts have been unable to induce a corresponding conviction in ordinary people and teachers. To be sincere, science has failed to indoctrinate modern scientists, who teach the extended evolutionary theory to their students but plead God for their children when late in the night they do not return back to home. Not to speak about the things they do when children and grandchildren get ill.

It seems that religion and science are in real life like water and salt that are very difficult to separate. Actually, a perfect separation is impossible. So, water and salt are mental categories that hardly are represented in the universe or in the lab. In fact, sophisticated procedures are necessary to get pure water at the

lab. By the same token, science and religion go hand in hand in many ideologies while purists respond with mechanistic concepts, as the next considerations show:

387. ***Natural philosophy and theology***

Some religious people like to see an architecture in the universe and so they like to study nature to capture the style of the architect. Their discipline is **natural theology**. By contrast, in **natural philosophy**, the architecture is an emerging phenomenon and the architect might be randomness or evolution.

Example: a young mother uses her maternal milk to feed her baby. She sees how perfect he is and she also has noticed that he is good. Both characteristics impulse her to thank the Creator for this wonderful world (Armstrong, [4] 1968). This situation illustrates the typical way of thinking in natural theology. By contrast, natural philosophy proposes that the baby is perfect thanks to natural selection and that he is good because otherwise no one will take care of him although the true mother would tolerate him more than the rest of humans. In fact, the mother is programmed by the genes she carries to take care of them be they in her genome or in that of the baby (Dawkins, [19] 1976).

We see that purists know how to defend themselves. Nevertheless, this is not the universal option of renown scientists. Take as example the declaration of a group of members of the USA National Academy of Science (Ayala et al, [6] 2008). They proclaim:

"*Science and religion address separate aspects of human experience. Many scientists have written eloquently about how their scientific studies of biological evolution have enhanced rather than lessened their religious faith. And many religious people and denominations accept the scientific evidence for evolution. Our education system and our society as a whole are best served when we teach science, not religious faith, in science classrooms*".

This important declaration was commented by the Editor of Nature ([29] 2008) as follows:

"*Three cheers for the US National Academy of Sciences for publishing an updated version of its booklet Science, Evolution, and Creationism (see http://www.nap.edu/sec). The document succinctly summarizes what is and isn't science, provides an overview of evidence for evolution by natural selection, and highlights how, time and again, leading religious figures have upheld evolution as consistent with their view of the world.*"

We find wisdom in both declarations but not of a purist type, rather it is practical, recognizing that real scientists are also human beings with personal world views that might be different than the official materialism. Purism is difficult to find but we know they it exists as it is shown by the proclamations of Dawkins ([7] 2001).

388. ***Creation science***

At the other extreme, we find people that consider that science is the best allied to their creationists ideas. In fact, the scientific method is a method that actually is used by many people as tool to filter propositions about nature as well as about inner psychological experiences, say, to maximize happiness. So, it is not the private property of science or of materialism. One can see that every other ideology uses it to enlighten crucial points. Therefore, there are indeed as many types of science as ideologies. By default, **science** means science under objective materialism.

Low probabilities are at the hearth of the scientific method and so they are welcome in various philosophical realms. Nevertheless, low probabilities are not necessary in religions to postulate the existence of God because gods are not invented, rather all gods in all religions have revealed themselves to their prophets in visions, dreams, prophecies that get fulfilled and, in some cases, miracles.

Our problems begin when a religion claims that we will be judged by God, such an in the Islam. To do so, He must have some ethical right, and of course, it is the divine creation of all that exists. But on the other hand, the problems of the religion begin when it gives specific details about the process used by the Creator to create the world. For instance, the gods of the Popol Vuh made the man from maize but just after checking that mud and wood were not that good for the purpose (Meta-religion, [70] 2011).

This detail clearly and distinctly predicts that the man should had appeared after plants, a fact that is scientifically correct. By the same token, some christian people that reason like that consider that their religion can be made into a scientifically robust theory and so they have forged the term "creation science" to name their discipline. Nevertheless, their works use to awake contempt in scientists (Habgood, [44] 2008). This reaction encloses no mystery at all: each world view is and must be spontaneously jealous against other world views otherwise it has not been sufficiently elaborated.

Beyond insults and resentment, one must be aware of the overall situation: the validity of a given argument cannot be discussed if not in the light of a given world view. As a rule, crucial points are judged in different or contrary ways by different world views. So, you can decide that a given fact falsifies or validates a theory but you must be aware of which world view enlivens your discussion. If you make it explicit, it would be easier for everyone to understand you and insults will be anymore necessary.

389. ***Intelligent design***

We see too much suffering in this world, so, one might consider that the idea of a god or a hacker on top of us is just ridiculous. Nevertheless, there are many natural prompts to consider the idea of a creator. For instance, our Java lab allows us to formulate a related intrigue:

A very intelligent person suffers a lot to complete a simple task of programming. So, how can natural evolution, which has no intelligence at all, be responsible of the genome, an example of extremely complex software?

In fact, everyone perceives that living beings are extremely complex and exceedingly perfect. So, everyone knows by instinct that a great intelligence exists behind life. The overall impression is so compelling in some persons that they think that it is God directly speaking to us. They would say that we exist thanks to **intelligent design** (Demski, [20] 2000).

But on the other hand, it was just recently when science became aware of the possibility that the intelligence behind complex objects or processes is not necessarily the exclusive property of God. In fact, we say that humans are very intelligent because they are capable of developing complex software, but genetic programming shows that evolution is also a software developer, and that is why we can say that **evolution is also intelligent**.

So, if someone argues that we exist thanks to intelligent design, he or she is saying something that is part of the credo of modern science. In this regard, we can reformulate our falsification of the evolutionary theory with a metaphor: evolution is similar to a very intelligent child, who by mere playing and gambling can eventually create great things but that is unable to leave the workbench clean after work. Nevertheless, in the last two million events it has appeared perfectly clean. To leave the workbench clean is a style of work that is distinctive of some humans and specially women.

390. *Detecting human-like intelligence*

We have introduced two kinds of intelligence, that of human beings and that of evolution. Can we distinguish one from the other? This question gather together creationists and people searching for extraterrestrial intelligence.

Our tinkering with numerical analysis to calculate p- and critical values has awaken and strengthened two important claims:

- Intelligence of non evolutionary origin can be detected in a family of programs if new and more powerful strategies that can be explained by mathematical reasoning are preferred to those naturally posited by evolution.

- The validation of algorithms is unrealistic in practical terms. Therefore industrial software is always dressed in abundant bugs that companies try to unveil and fix through beta versions and automatic feedback. Additionally, human developers make every effort to erase the tremendous history of the battle with bugs from the final product. We consider that this is distinctive of competitive human-like intelligence. Of course, we can program simulated evolution to produce clean outputs, but nobody expects that from natural evolution.

17.8 Challenges

Is there some work to be done? Yes. We have various challenges that will generate a lot of work and ensuing discussions in our community for a long time. Let us list some of them:

391. ***The omnipresence of bugs***

Developing software is difficult. The **law of the omnipresence of bugs** says that there is no software development without huge amount of bugs ensued by their fixing that produces more bugs. This law is mandatory for every real instance but for the purpose of cosmological discussions, there exists a small probability, too small to be imagined, of developing a large program without committing bugs.

As we see, the problem of bugs is an important one. Therefore, we need to pass from stories about human experiences to automated registration of data, statistics and rigorous testing of hypotheses. This is already part of the ordinary research in computing science (D'Ambros and Lanza, [15] 2006).

392. ***Particular falsifications***

We have formulated a general falsification of the evolutionary theory. Actually, this must be taken as a challenge to study specifically the validity of the theory for each species. Thus, if the evolutionary theory is false, its falsity must be supported by independent and specialized arguments in relation with each species. In conclusion, we have provided over two million of scientific challenges.

393. ***Visual proofs***

Our law of the omnipresence of bugs predicts that we must find in the record fossil and in extant populations a vivid struggle against complexity. For a beginning it is not too bad. But the high potential of Java is actually a challenge to create artificial reality simulations in which anyone will be able to see how the predictions of the law of bugs incarnates in a true evolutionary process through every sort of malformations and malfunctions. To that aim, heavy duty machinery is available: it is JavaFX, the modern Java way of making graphics and animations. You can learn JavaFX beginning from `Vol XII Fossils` from our website.

394. ***Multivariate approach***

We know that a trotter will sweat during a race across the desert as well as any developer will commit many bugs while it, he or she composes a computer program. In each case, we have one single descriptor or variable, say, the number of bugs per 100 lines of code. The challenge for our community is to pass from univariate approach to a multivariate one, such as it happens with a trotter, for whom we have various mandatory predictions: apart from abundant sweat, long trotting produces smell denoting exhaustion, tremendous lost of coordination power, face disfigurement, renal damage and shift of ionic balance of the blood,

damage in the fingers of the feet and in the knees. If the race is too long, the lost of coordination might be so severe that the trotter gets unable of cushioning impacts and every vertebra might get injured. So, if one of these signs is absent, we immediately know that we are looking at a deceiver, actually, a beginner that in vain imagines that people will not unmask him at the first sight.

In some years or decades, this multivariate approach will mature enough that many people will inadvertently consider that evolution is a stupidity. Contrary to them, we will in our community create ways to apply evolution for the solution of the most difficult and intractable problems. In fact, *evolution is as smart as your mind*.

395. ***The greatest triumph***

We use to say that the evolutionary theory has over two million falsifications, one for each species. This generalization is based on our plain confidence in the inviolable power of the law of the omnipresence of bugs. Nevertheless, it will be a great achievement of science if someone can show beyond our criticisms that a given species arouse from another one by natural evolution. Flies seem to be a good candidate and humans a very bad one. In this regard, we have two questions:

- How can the degree of the potential for speciation be finely tuned?
- What is a species?

In regard with the second question, we can say the following:

Powerful as evolution is, mankind has been unable of having it to produce a Griff. We accept this failure as a natural result of the fact that evolution is real and not magic: there are problems that break down every computing system. More specifically:

Natural evolution is a computing system that is unable to compute the solution of the problem of transforming one species into any other one.

This remark is indeed a definition of species: what is a species? A species is for genetic programming a fix point of evolution, i.e., is something that evolution cannot change. This does not exclude the existence of strange attractors, a situation in which permanent evolution exists around a given leitmotif, a domain which evolution revolves around and cannot abandon. The net challenge is to clearly specify the maximal scale of time during which the number of species is greater than two.

396. ***Evolution and teleonomy***

All of our experience in this volume has been with design of programs by human beings. It is perfectly clear that in design we always have a purpose beforehand, so we say that design presupposes **teleonomy**, a specified intention.

But evolution in nature is purposeless. So, we ask: can the mandatory law of bugs, a human-based generalization, be fairly extrapolated to natural evolution? We discuss this question in Vols II and III of this series.

397. ***The evolutionary postulate***

Since developing software is extremely difficult, reuse it as much as possible. In real circumstances, use and reuse includes mutations and recombination of programs. This is the **evolutionary strategy** for easing the development of software.

The problem is that use and reuse has a cost, say, time to understand extant code and the ensuing effort to modify it. Therefore, the cost of reuse must be less than the cost of developing new software ab initio. The **evolutionary postulate** claims that the balance always can be settled in favor of reuse, so that evolution is a tautology, such as many scientists seem to think. Nevertheless, the Author learned from stonemasons that this postulate is false: there are many modifications that they refuse to do and instead they prefer in some cases to build new structures from pure beginning. We have here a theme for investigation: which is the span of validity of the evolutionary postulate?

We might think that if evolution is well programmed, it becomes a tautology. But our experience shows that the ways as evolution can be badly programmed surpasses by far those as evolution can be well programmed. More to the point, even if one makes a significant effort in programming, one may end with very poor results. That is why we are committed to do our best to show others that evolution might be chosen as a profession for a life: one always will plenty of promising, intriguing and challenging problems.

398. ***Evolutionary engineering (corrected)***

All humans automatically try to apply the evolutionary strategy in every task of design but although in an incipient way. To become a professional evolutionist, one must work very hard and commit oneself to continuously improving expertise. Thus, we get the **evolutionary engineering** that studies how to use the evolutionary strategy to get the most of it. In this regard, we already have been acquainted with some few but powerful principles:

1. Use a modular approach: use and reuse well designed and tested modules as building blocks for higher units.The top-down version of reuse is that it is the style of problem solving by recursively dividing a task in subtasks that might be solved by specific encapsulated packages of code. The bottom-up view is that one can encapsulate solutions to problems into modules that in their turn can be encapsulated in higher level packages. We have identified *evolution* with *smart reuse*, in the sense that one modifies and recombines extant modules to craft other ones, an operation that is routine in the work of every developer.

2. The experience in battling the complexity of software design has produced and emphasized a very simple and fruitful solution: complexity comes

structured in substantives + verbs. So, Java allows to define classes that content group of variables + methods to modify them. The resultant technology is the very famous OOP, Object Oriented Programming. Java is an object oriented programming language and this means that the recursiveness or evolvability of the process of encapsulation is not only *enabled but also necessary* for complex projects.

3. In one word, we have clearly shown why evolution is in computing science a style of problem solving that has been engineered, consciously crafted, for complete recursiveness. The existence of the genome demonstrates that this program is well posed:

   The genome is and will remain forever as the most marvelous example of evolvable software: it is so evolvable that its evolvability has been converted into a self contained theory, the evolutionary theory, which pretends to offer a natural and mechanistic explanation to the origin of species and of complexity. While the evolutionary theory is false, the evolvability of the genome is the very fact that allows modern genetic engineers to produce miracles (Epstein, l.c., [31] 2004) even though they possibly never have written a single computer program. In computer science, Functional Programming emphasizes that functions can be used as fundamental building blocks for the construction of evolvable and very complex structures because functions can be naturally composed.

   A fundamental task for evolutionary engineering is to clearly specify how one can design a programming environment to produce products as evolvable as the genome (Mens, [69] 2008).

399. *Can complexity be defeated?*

Our relative successes in alleviating the hard work of developing software invites a question: can computing or evolutionary engineering defeat complexity? We are pessimistic:

A simple reason is that a carefully designed code might look simple and evolvable at small to medium scale. But due to recursiveness, the overall system of mutual calls among methods and classes might get so entangled that human comprehension will simply collapse. It is a situation like this that promotes the desire to abandon the work of perfecting or maintaining very complex projects. Thus, we predict that evolution is broken down by complexity in every practical situation. But if one tries to replace the human developer with a machine, then the necessary time to solve problems may diverge towards infinite. These are more reasons to propose evolution as a profession: we always will be plenty of very hard work.

400. *The problem of styles*

A **style** is the manner as something can be done. There are various and interesting questions in relation with styles of software development:

1. To use and produce many, many descriptors of style that could be statistically studied.

2. To mechanistically understand why the existence of huge numbers of different styles is possible to such a degree that each person may have his or her own style.

3. To study how the style evolves when the capability of solving complex problems is rewarded.

4. The function paradigm to be complete must be extended with the possibility to accept void input and/or void output, a facility that is equivalent to say that one can work over globally defined variables. These variables are loose ends that can get tied to whatever thing transforming the world in a spaghetti jungle that will eat you alive (Foote and Yoder, [35] 1995). Now, globally defined variables can be converted into declared, local ones and so every thing will get clear. Nevertheless, one prefers to not doing that because one divides the world in two parts: a primary part that must remain in focus plus a secondary part that runs in background. And this causes untied ends that leads to a spaghetti jungle, the usual style of programming in real life. We have argued that the function paradigm is observed by biochemistry. What about the genome itself?

5. Human makers prefer to show terminated things erasing everything that might insinuate a process of construction. What is the marginal, additional cost of programming evolution to synthesize clean products?

6. Humans use ideas of geometric origin and abstraction, say, algebra. This corresponds to high level correlations: how is their emerging when simulated evolution battles complexity?

7. Styles correspond to extended strong self-correlations. Can such things indeed emerge in an evolutionary experiment?

17.9 The pedagogic conundrum

If the teachers learns bad, students will learn worst.

401. ***Java vs. Calculus***

One of our aims in this series is to motivate the idea that it is better for students to learn Java in preference to calculus. Nevertheless, the work along the present volume shows that one cannot achieve usual standards in computing if not with some specialized mathematics that must be learned in a not elementary course of calculus. So, one needs both, Java and calculus, and one must decide next how to merge Java and calculus that students could enjoy getting well prepared to face coming challenges.

402. *Two demands*

In regard with Java alone, we have followed our instinct more than anything else. Therefore, we automatically face a peril: Java is not a matter of instinct, rather it is the result of a smart digesting of the suffering of the programming community along decades that came out with a paradigm on its own, OOP -object oriented programming- in an environment that avoid bugs and facilitates their correction. We have arrived to it following certain evolution. But by contrast, experts demands at least three things:

- The OOP style must be native to education, so that students are born to light in this new world and without the karma that encloses having learned ancient and incipient ways of programming.
- Students must be prepared to naturally get experts.
- Good style combines well with good style and only with good style (Riel, [85] 1996)

We have in our community the option to experiment this ideology thanks to a book specially target to suit in this purpose (Eckel, [27] 2003). On the other hand, the fabric of expertise has rendered ab initio professional solutions that include clean implementations of the functional paradigm and its formalization, lambda calculus. People attracted by this ideal can experiment with Scheme, an evolved form of Lisp, (Felleisen et al, [33] 2001) and Squak, the modern version of Small talk (Black et al, [9] 2009; Gomez, [43] 2006). The obvious message of this school is that

1. An expert obeys the functional paradigm. Therefore, everything must be adjusted to follow it, including the programming environment. The function paradigm is evolvable by its very nature, so learning to design programs together with modifications and testing of programs are friendly activities. Pure functional style is very expensive. So, very practical seems to be a wise mixing of functional programming and OOP, such as it is found in Java 8 that we reviewed in Vol XVI.
2. The path to expertise begins with practicing a simple receipt (Klimas et al, [60] 1996) that we summarize as follows: clearly understand problems and check results as soon as possible. This implies before everything else, to having made an exercise by hand to compare results with the output of the program. Next, one must document the purpose of the program or method and to define its input and output before its implementation.

One must welcome every effort to get into expertise according to the most holy standards but it is not recommendable to make that option into a duty: doctors also die. This means that there is at last no expertise against complexity and so Java peacefully stands as a golden weapon for everybody that wants to battle it (by contrast to functional programming, Java is procedural

or imperative). The immanent weakness of glorious programming paradigms must be kept in mind because experts are programmed by their very nature to smash other people. Just think of a Ferrari that overcomes the car of a teenager. Nevertheless, which one is fully protected against accidents?

17.10 A simple test

It is interesting to test the robustness of own ideas.

403. ***Exercise.*** *Consider the case of artificial selection of race horses in which an evolutionary trend towards extreme and specialized perfection is attempted. Clearly depict where are the bugs and where are the tracks of an evolutionary process toward complexity and perfection. If you cannot, reexamine our mandatory law of bugs: is it pathetically false?* *Answer*

17.11 A personal challenge

Use the material of this chapter to wisely update your personal version of discuted thematic. Keep in mind that the main purpose of every document in our community is not to provide a final truth or a complete treatment of a theme but rather to incite others to read further and to work on their own.

17.12 Conclusion

The idea that the genome is software plus our hard work on programming have allowed us to claim that the evolutionary theory contains a great idea but is scientifically false. The great idea is that evolution can be used as a tool to solve every kind of problems. On the other hand, the falsification of the evolutionary theory stems from the fact that developing software is indeed an evolutionary experience and that without exception the design of software is always tied to an evolution of perfection through a fierce battle against bugs. Thus, every person with even a poor experience in software development shall predict that if the evolutionary theory is correct, then we must find in the record fossil and in extant populations a vivid struggle against complexity with abundant witness of malformations and malfunctions. Such testimony is not present in nature. That is why the theory is false. Our position awakes very interesting and difficult problems in mathematics, computing science, biology, genetics, evolution and philosophy.

Answers to exercises

Problems of Chapter 1

10, page 7. Complete instructions are finished with ";". Commentaries are initialized with // and can be ended with any sign. A line that declares the name of the class or of a method does not have ";", instead it has a key pair: {}.

12, page 8.

```
package ejvol5p;


//Program E12 Biography
//Same as Program   A11
//This is my first exercise,
//It prints my name and birthday.
//The name of this class or program is Biography

public class Biography
{
 public static void main(String[] args)
  {
   System.out.println("L.J., the Intelligent.");
   System.out.println("Birthday: March 5th.");
  }
}//End of Program E12 Biography
```

Problems of Chapter 2

22, page 14.

```
//Program E22 OneLine
//Same as program A16
//This program introduces integers
//and their addition.
//Report is printed in one line.

package ejvol5p;

public class OneLine {

    public static void main(String[] args) {
        //Declaration and initialization
        // of variables
        int x = 2;
        int y = 3;
        //Computation
        int Result = x + y;
        // Report i one line
        System.out.println(" The sum of x = " + x + " and "
                + " y = " + y + " is " + Result);
    }
}//End of Program E22 OneLine
```

24, page 14.

```
//Program E24 IntOperations
//Multiplication, division and
//subtraction of  two integers.

package ejvol5p;

public class IntOperations {

    public static void main(String[] args) {
        //Declaration and initialization
        // of variables
        int x = 6;
        int y = 2;
        //Computation
        int result1 = x * y;
        int result2 = x / y;
```

```
        int result3 = x - y;
        // Report
        System.out.print(" The multiplication of x = " + x);
        System.out.println(" and  y = " + y + " is " + result1);
        System.out.print(" The division of x = " + x);
        System.out.println(" by  y = " + y + " is " + result2);
        System.out.print(" x = " + x);
        System.out.println(" minus  y = " + y + " is " + result3);
    }
}//End of Program Program E24 IntOperations
```

27, page 18. The more dominant is a method, the lower is its position in the text.

28, page 18. The order does not matter, the program runs softly no matter the relative position of the methods along the text.

33, page 21.

```
//Program E33 DivisionByZero
//same as Program A25
//Division by zero

package ejvol5p;

public class DivisionByZero {

    public static void main(String[] args) {
        //Declaration and initialization
        // of variables
        int x = 6;
        int y = 0;
        //Computation
        int result = x / y;
        // Report
        System.out.print(" The division of x =" + x);
        System.out.println(" by  y = " + y + " is " + result);
    }
}//End of Program E33 DivisionByZero
```

When Java is commanded to run this program, it answers: 'Exception in thread "main" java.lang.ArithmeticException: / by zero'.

A division by zero leads to a contradiction:

$0 = 3 \times 0 = 5 \times 0 = 0$

Dividing by zero, we get:
$3 = 5$
which is a contradiction.

35, page 22. The code is robust against division by zero but the division of 7 by 2 renders 3 instead of 3.5.

Problems of Chapter 3

48, page 27.

```
//Program E48 Squares
//The for statement with output
//suitable for a graphic.

package ejvol5p;


public class Squares {

    public static void main(String[] args) {
        String tab = "\t";
        for (int i = 1; i < 7; i++) {
            System.out.println(i + tab + i * i);
        }
    }
}//End of Program E48 Squares
```

50, page 27. The new program prints numbers from 1 to 7, including the 7, and their squares.

51, page 28.

```
//Program E51 Squares10
//same as Program A37
//Variation of the for statement.

package ejvol5p;

public class Squares10 {

    public static void main(String[] args) {
```

```
        String tab = "\t";
        for (int i = 0; i <= 10; i++) {
            System.out.println(i + tab + i * i);
        }
    }
}//End of Program E51 Squares10
```

52, page 28.

```
//Program E52 Cubes
//Same as Program A38
//Variation of the for statement.

package ejvol5p;

public class Cubes {

    public static void main(String[] args) {
        String tab = "\t";
        for (int i = 0; i <= 10; i++) {
            System.out.println(i + tab + i * i * i);
        }
    }
}//End of Program E52 Cubes
```

55, page 29. Java over 64 bits accepts 100000000 as an integer.

Problems of Chapter 4

59, page 32. Excel and ancient versions of Calc failed to make a graphic because Java uses a period in decimals numbers while Excel and Calc use a comma. This was a bug of intercommunication of developer teams. This is typical. Modern versions of Calc cause no problem.

60, page 32. The console supports more than 7000 lines of text with two decimal numbers per line. So it is more than enough for a good part of our course. But it is also a restriction one must learn to manage.

62, page 32. In previous programs , we used the expression $i++$ to mean that the index be incremented by one. The new instruction

```
for(int i = 0; i<=100 ; i= i + 7 )
```

says that the index must be incremented by 7 beginning from i=0 and ended with i=100. Moreover, each i was multiplied by 1000. So, the program lists the multiples of 7000.

64, page 33. Program to calculate the squares of the real numbers beginning with 3 and ending with 4 with step 0.05:

```
//Program E64 Fractional2
//Variation of the for statement
//with fractional increments

package ejvol5p;

public class Fractional2 {

    public static void main(String[] args) {
        String tab = "\t";
        // The index is declared as "double",
        // so it accepts a decimal  expression
        // in its increment.
        for (double i = 3; i <= 4; i = i + 0.05) {
            System.out.println(i + tab + i * i);
        }
    }
}//End of Program E64 Fractional2
```

Problems of Chapter 5

67, page 36.

```
//Program E67 ArrayCubes
//Same as Program A82
// Cubes in an Array.

package ejvol5p;

public class ArrayCubes {

    public static void main(String args[]) {
```

```
        // Declaration of an array with integer entries
        int data[];  // data is the name.

        // Assignment of the capacity of the array:
        // ten entries, beginning from zero.
        // data[i] can be defined for i from zero to 11
        // data[12] is not defined
        data = new int[12];

        int limit = 11;
        for (int i = 0; i <= limit; i = i + 1) {
            data[i] = i * i * i;
        }

        // The array is written to the console.
        for (int i = 0; i <= limit; i = i + 1) {
            System.out.println(i + " " + data[i]);
        }
    }
}//End of Program E67 ArrayCubes
```

69, page 37. To make sure that all data are included, one can force the program to write each processed datum. Mean and variance of some numbers of type double:

```
//Program E69 MeanData2
//Same as program A83
//Mean and variance of some data of type double.

package ejvol5p;

public class MeanData2 {

    public static void main(String args[]) {
        //Declaration and assignment of the data array
        double Data[] = {1.7, 5, 2, 4, 7.3, 8, 9, 5, 6.4, 3, 5};
        // Report of data
        System.out.println("Data are");
        // To know the length of the array : Data.length
        for (int i = 0; i < Data.length; i++) {
            System.out.print(Data[i]);
            System.out.println(); //pass to the next line
        }
```

```
        //   Let us calculate the mean
        double sum = 0;
        int n = Data.length;
        for (int i = 0; i < n; i++) {
            sum = sum + Data[i];
        }
        double mean = sum / n;
        System.out.println("Sum \t " + sum);
        System.out.println("Mean \t " + mean);

        //    Let us calculate the variance
        double sum2 = 0;
        for (int i = 0; i < n; i++) {
            sum2 = sum2 + (Data[i] - mean) * (Data[i] - mean);
        }
        double var = sum2 / (n - 1);
        System.out.println("Variance \t " + var);
    }

}//End of program E69 MeanData2
```

Problems of Chapter 6

79, page 44. The following program computes the mean and variance of a list of data.

```
//Program E79 MeanList
//This program computes the mean and
//variance of a list of data.
public class MeanList {

private  static double  DataVect[]  = {6, 8, 7, 6, 5, 7, 8, 6, 9};

//Prints data in vector
private static void printVector(double Vect[])
 {
  System.out.println( "Data are" );
  for ( int i = 0; i < Vect.length; i++ )
   System.out.println( Vect[ i ]  );
 }

 //Sums data in vector with real numbers
```

```
 private static double sumVector(double Vect[])
  {
   int n = Vect.length;
   double sum = 0;
   for ( int i = 0; i < n; i++ )
  sum = sum + Vect[i];
   return sum;
 }

//The mean of a list of data
private static double meanVector(double Vect[])
 {
  double sum = sumVector(Vect);
  int n = Vect.length;
  double mean = sum / n;
  return mean;
 }

//The variance of a list of data
private static double varVector(double Vect[])
 {
  //Mean
  double mean = meanVector(Vect);
  // Variance is calculated by definition
  int n = Vect.length;
  double sum2 = 0;
  for(int i = 0; i < n; i++ )
   sum2 = sum2 + (Vect[i]-mean)*(Vect[i]-mean);
  double var = sum2 / ( n-1);
  return var;
}

//The mean and variance of a list of data
private static void meanVarVector(double DataVect[])
 {
  System.out.println("Data length = " + DataVect.length);
  double mean = meanVector(DataVect);
  System.out.println( "Mean = "+ mean);
  double var = varVector(DataVect);
  System.out.println( "Variance \t "+ var);
  double deviation = Math.pow( var, 0.5);
  System.out.println( "Deviation = "+ deviation);
  double coeffVar =  deviation/mean;
  System.out.println( "Coefficient of variation = " + coeffVar);
}
```

```
public static void main(String[] args)
{
 printVector(DataVect);
 meanVarVector(DataVect);
 }
}
```

80, page 44. The following program computes the mean and variance of numbers from 0 to 999 included:

```
//Program E80 MeanList2
//This program computes the mean and
//variance of numbers from 0 to 999.

package ejvol5p;

public class MeanList2 {

    private static  double DATA[] = new double[1000];

    //Prints data in vector
    private static void printVector(double Vect[]) {
        System.out.println("Data are");
        for (int i = 0; i < Vect.length; i++) {
            System.out.println(Vect[i]);
        }
    }

    //Sums data in vector with real numbers
    private static double sumVector(double Vect[]) {
        int n = Vect.length;
        double sum = 0;
        for (int i = 0; i < n; i++) {
            sum = sum + Vect[i];
        }
        return sum;
    }

    //The mean of a list of data
    private static double meanVector(double Vect[]) {
        double sum = sumVector(Vect);
        int n = Vect.length;
        double mean = sum / n;
```

```
        return mean;
    }

    //The variance of a list of data
    private static double varVector(double Vect[]) {
        //Mean
        double mean = meanVector(Vect);
        // Variance is calculated by definition
        int n = Vect.length;
        double sum2 = 0;
        for (int i = 0; i < n; i++) {
            sum2 = sum2 + (Vect[i] - mean) * (Vect[i] - mean);
        }
        double var = sum2 / (n - 1);
        return var;
    }

    //The mean and variance of a list of data
    private static void meanVarVector(double DATA[]) {
        System.out.println("Data length = " + DATA.length);
        double mean = meanVector(DATA);
        System.out.println("Mean = " + mean);
        double var = varVector(DATA);
        System.out.println("Variance \t " + var);
        double deviation = Math.pow(var, 0.5);
        System.out.println("Deviation = " + deviation);
        double coeffVar = deviation / mean;
        System.out.println("Coefficient of variation = " + coeffVar);
    }

    public static void main(String[] args) {
        for(int i = 0; i < 1000; i++)
            DATA[i] = i;
        printVector(DATA);
        meanVarVector(DATA);
    }
}//End of Program E80 MeanList2
```

81, page 45. A code to calculate the mean and variance of a table of absolute frequencies follows:

```
//Program E81 MeanFreqTableCode
//Finds the mean and variance of a table
//of absolute frequencies.
```

```
package ejvol5p;

public class MeanFreqTableClone {

//Frequency table: first coordinate = class marker
//second coordinate = frequency
    private static final double FREQTABLE[][] = {
        {4, 3},
        {7, 6},
        {10, 10},
        {13, 15},
        {16, 4},
        {19, 2}};

//Prints the frequency table
    private static void printFreqTable(double FreqTable[][]) {
        System.out.println("x and its absolute frequency");
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][]) {
        double n = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][]) {
        double mean;
        double sum = 0;
        double n = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            //Class marker times frequency
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
```

```
            n = n + FreqTable[i][1];
        }
        mean = sum / n;
        return mean;
    }

//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][]) {
        double mean = meanFreqTable(FreqTable);
        int m = FreqTable.length;
        // Let us calculate the variance
        double sum2 = 0;
        for (int i = 0; i < m; i++) {
            sum2 = sum2 + FreqTable[i][1] * (FreqTable[i][0] - mean)
                    * (FreqTable[i][0] - mean);
        }
        double n = 0;
        for (int i = 0; i < m; i++) {
            n = n + FreqTable[i][1];
        }
        double var = sum2 / (n - 1);
        return var;
    }

//Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][]) {
        double var = varFreqTable(FreqTable);
        double dev = Math.pow(var, 0.5);
        return dev;
    }

//Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][]) {
        printFreqTable(FreqTable);
        double n = nFreqTable(FreqTable);
        System.out.println("Number of measured objects = " + n);
        double mean = meanFreqTable(FreqTable);
        System.out.println("Mean = " + mean);
        double var = varFreqTable(FreqTable);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable);
        System.out.println("Deviation =  " + deviation);
    }
```

```
    public static void main(String[] args) {
        meanVarTable(FREQTABLE);
    }

}//End of Program E81 MeanFreqTableClone
```

83, page 45. Program to find by method 2 the mean and variance of a table of absolute frequencies.

```
//Program E83 MeanFreqtable2
//Finds the mean and variance of a table
//of absolute frequencies.
//Method 2

package ejvol5p;


public class MeanFreqtable2 {
//Declaration of data:
//observed value in the first coordinate
//Absolute frequency in the second.

    private static final double FREQTABLE[][] = {
        {4, 3},
        {7, 6},
        {10, 10},
        {13, 15},
        {16, 4},
        {19, 2}};

//Prints the frequency table
    private static void printFreqTable(double FreqTable[][]) {
        System.out.println("x and its absolute frequency");
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][]) {
        double n = 0;
```

```
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

//Calculates the sum XF   of
//a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][]) {
        double sum = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the sum X2F   of
//a table of absolute frequencies
    private static double
            sumX2FFreqTable(double FreqTable[][]) {
        double sum = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][0]
                    * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][]) {
        double n = nFreqTable(FreqTable);
        double mean = sumXFFreqTable(FreqTable) / n;
        return mean;
    }

//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][]) {

        double n = nFreqTable(FreqTable);
        double Sxx = sumX2FFreqTable(FreqTable)
```

```
                - Math.pow(sumXFFreqTable(FreqTable), 2) / n;
        double var = Sxx / (n - 1);
        return var;
    }

// Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][]) {
        double var = varFreqTable(FreqTable);
        double dev = Math.pow(var, 0.5);
        return dev;
    }

// Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][]) {
        printFreqTable(FreqTable);
        double n = nFreqTable(FreqTable);
        System.out.println("Number of measured objects = " + n);
        double sumXF = sumXFFreqTable(FreqTable);
        System.out.println("sum xF  = " + sumXF);
        double mean = meanFreqTable(FreqTable);
        System.out.println("Mean = " + mean);
        double sumX2F = sumX2FFreqTable(FreqTable);
        System.out.println("sum x2F  = " + sumX2F);
        double var = varFreqTable(FreqTable);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable);
        System.out.println("Deviation =  " + deviation);
    }

    public static void main(String[] args) {
        meanVarTable(FREQTABLE);
    }

}//end of Program E83 MeanFreqtable2
```

87, page 50. The code that groups data in interval classes, when the inferior border and the length of the intervals are given as input, follows:

```
//Program E87 Grouping2
//Groups data in interval classes.
//Outputs a frequency table with its
//mean and variance.
//Input: a list of data +
//inferior border and length of the intervals.
```

```
package ejvol5p;

public class Grouping2 {

//Declaration of data:
//observed values:
    private static final double DATAVECT[]
            = {
                48.5, 49.2, 51.0, 50.9, 48.7, 50.5, 49.5, 50.8, 50.0, 51.1,
                50.7, 51.1, 49.3, 49.1, 50.0, 48.7, 51.2, 49.2, 49.0, 49.3,
                49.1, 48.7, 48.6, 49.2, 49.9, 50.1, 50.1, 50.9, 52.4, 50.2,
                49.5, 50.9, 51.4, 49.7, 49.8, 50.8, 50.3, 51.8, 50.0, 51.1,
                51.3, 50.1, 50.4, 51.1, 49.8, 49.8, 50.1, 50.7, 50.1, 49.9,
                49.7, 51.1, 49.7, 49.9, 49.8, 50.5, 49.7, 50.8, 50.7, 50.7,
                50.8, 51.5, 51.0
            };

//The left border of intervals, less or equal to the
//minimum value of data.
    private static final double INFLIMIT = 48.5;
//Interval length
    private static final double INTERVALLENGTH = 1;
//Max number of allowed classes
    private static final int MAXNCLASSES = 20;
//Borders of the intervals
    private static final double BARRIERS[] = new double[MAXNCLASSES + 1];
    private static final double CLASSMARKERS[] = new double[MAXNCLASSES];
    private static int nClasses;
    private static final double FREQTABLE[][] = new double[MAXNCLASSES][2];

//Prints data in vector
    private static void printVector(double Vect[]) {
        for (int i = 0; i < Vect.length; i++) {
            System.out.println(Vect[i]);
        }
    }

//The border of each interval is calculated
    private static void getBorders(double Vect[]) {
        BARRIERS[0] = INFLIMIT;
        CLASSMARKERS[0] = INFLIMIT + INTERVALLENGTH / 2;
        //Calculate maximal value of data
        double max = Vect[0];
        for (int j = 1; j < Vect.length; j++) {
            if (Vect[j] > max) {
```

```
                max = Vect[j];
            }
        }
        System.out.println("Max value = " + max);
        //Calculates number of classes
        nClasses = (int) ((max - INFLIMIT) / INTERVALLENGTH) + 1;
        System.out.println("Number of classes = " + nClasses);
        //Calculates interval borders and class markers
        for (int j = 1; j <= nClasses; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + INTERVALLENGTH;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + INTERVALLENGTH;
        }
        System.out.println("Borders are");
        printVector(BARRIERS);
        System.out.println("Class markers are");
        printVector(CLASSMARKERS);
    }

//Data are grouped into interval classes.
    private static void group(double Vect[]) {
        getBorders(Vect);
        int nData = Vect.length;
        //Class markers a re defined
        for (int j = 0; j < nClasses; j++) {
            FREQTABLE[j][0] = CLASSMARKERS[j];
            FREQTABLE[j][1] = 0;
        }
        //Frequencies are calculated.
        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < nClasses; j++) {
                if ((BARRIERS[j] <= Vect[i]) & (Vect[i] < BARRIERS[j + 1]))
                    FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
                }
            }

        }
    }

//Prints the frequency table
    private static void printFreqTable(double FreqTable[][]) {
        System.out.println("x and its absolute frequency");
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
```

```
    }

//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][]) {
        double n = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

//Calculates the sum XF   of
//a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][]) {
        double sum = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the sum X2F   of
//a table of absolute frequencies
    private static double
            sumX2FFreqTable(double FreqTable[][]) {
        double sum = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][0]
                    * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][]) {
        double n = nFreqTable(FreqTable);
        double mean = sumXFFreqTable(FreqTable) / n;
        return mean;
    }
```

```
//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][]) {

        double n = nFreqTable(FreqTable);
        double Sxx = sumX2FFreqTable(FreqTable)
                - Math.pow(sumXFFreqTable(FreqTable), 2) / n;
        double var = Sxx / (n - 1);
        return var;
    }

// Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][]) {
        double var = varFreqTable(FreqTable);
        double dev = Math.pow(var, 0.5);
        return dev;
    }

//Calculates the coefficient of variation of
//a table of absolute frequencies
    private static double coefficient(double FreqTable[][]) {
        double c = devFreqTable(FreqTable) / meanFreqTable(FreqTable);
        return c;
    }

// Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][]) {
        printFreqTable(FreqTable);
        double n = nFreqTable(FreqTable);
        System.out.println("Number of measured objects = " + n);
        double sumXF = sumXFFreqTable(FreqTable);
        System.out.println("sum xF  = " + sumXF);
        double mean = meanFreqTable(FreqTable);
        System.out.println("Mean = " + mean);
        double sumX2F = sumX2FFreqTable(FreqTable);
        System.out.println("sum x2F  = " + sumX2F);
        double var = varFreqTable(FreqTable);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable);
        System.out.println("Deviation =  " + deviation);
        double coefficient = coefficient(FreqTable);
        System.out.println("Coefficient of variation =  " + coefficient);
    }

    public static void main(String[] args) {
```

```
        group(DATAVECT);
        meanVarTable(FREQTABLE);
    }

}//End of Program E87 Grouping2
```

101, page 58. The next code composes the PSTricks code that draws a bar chart associated to a given frequency table. The PSTricks code includes instructions to print the heigth of each colum over its roof.

```
//Program E101 PstricksCode2
//The next program receives as input a frequency table
//and produces the (Latex) PSTricks code for
//the corresponding bar graph.
//Labels for heights are included.

package ejvol5p;

public class PstricksCode2 {
    //Width of each column

    private static final double DELTAX = 4;

    //Declaration of input: a frequency table
    //x and its frequency
    /*
private static double    FREQTABLE[][] =
{
{3, 3},
        {4, 4},
        {5, 6},
        {6, 2},
        {7, 1},
        {8, 5}
      };
     */

    private static final double FREQTABLE[][]
            = {
                {0, 3},
                {1, 2},
                {2, 0},
                {3, 5}
```

```
        };

    //Produce the pstricks  code for the bar graph
    //of a frequency table
    public static void makeCode(double FreqTable[][]) {

        int n = FreqTable.length;
        //The minimum x-coordinate is found
        double Minxx = 1000;
        for (int i = 0; i < n; i++) {
            if (FreqTable[i][0] < Minxx) {
                Minxx = FreqTable[i][0];
            }
        }
        int Minxxx = (int) (DELTAX * Minxx - DELTAX);
        //The maximum x-coordinate is found
        double Maxx = 0;
        for (int i = 0; i < n; i++) {
            if (FreqTable[i][0] > Maxx) {
                Maxx = FreqTable[i][0];
            }
        }
        int Maxxx = (int) (DELTAX * Maxx + DELTAX);
        //The maximum y-coordinate is found
        int Maxyy = 0;
        for (int i = 0; i < n; i++) {
            if (FreqTable[i][1] > Maxyy) {
                Maxyy = (int) FreqTable[i][1] + 2;
            }
        }
        //Starting clause
        //x- and y-units.
        //The needed space is defined
        System.out.println("\\begin{center}"
                + "\n\\psset{xunit=0.15,yunit=0.3}"
                + "\n\\begin{pspicture}(" + Minxxx + ",-0.5)("
                + Maxxx + "," + Maxyy + ")");
        //A rectangle is drawn for each entry in the
        //frequency table

        int lim1, lim2;
        for (int i = 0; i < n; i++) {
            lim1 = (int) (DELTAX * FreqTable[i][0] - DELTAX / 2);
            lim2 = (int) (lim1 + DELTAX);
            System.out.print(
                    "\n\\psline(" + lim1 + ",0)"
```

```
                    + "(" + lim1 + "," + FreqTable[i][1] + ")"
                    + "(" + lim2 + "," + FreqTable[i][1] + ")"
                    + "(" + lim2 + "," + "0)"
            );
        }

        //Horizontal axis
        System.out.println("\n\\psline("
                + Minxxx + ",0)(" + Maxxx + ",0)");

        //Class markers (first coordinates) are indicated
        for (int i = 0; i < n; i++) {
            int xposition = (int) (DELTAX * FreqTable[i][0]);
            int yposition = - 1;
            System.out.print(
                    "\n\\rput*(" + xposition + "," + yposition + "){"
                    + (int) (FreqTable[i][0]) + "}"
            );
        }
        System.out.println();

//Labels for heights (second coordinates) are indicated
        for (int i = 0; i < n; i++) {
            int xposition = (int) (DELTAX * FreqTable[i][0]);
            int yposition = (int) (1 + FreqTable[i][1]);
            System.out.print(
                    "\n\\rput*(" + xposition + "," + yposition + "){"
                    + (int) (FreqTable[i][1]) + "}"
            );
        }

        //End clause
        System.out.println("\n\\end{pspicture}");
        System.out.println("\\end{center}");
    }

    public static void main(String[] args) {
        makeCode(FREQTABLE);
    }
}//End of Program E101 PstricksCode2
```

Problems of Chapter 8

123, page 82. The next code studies a null hypothesis in regard with the mean and makes some checking operations to test the correctness of the code.

```
//Program E123 TestMean2
//This program contrasts   a    fact, xo = 21,
//against the null hypothesis
//that x must follow a normal distribution with
//mean = 3
//(knowing that   the deviation is  0.8).
//The program contains some tests
//for correctness.

package ejvol5v3p;

import java.util.Random;

public class TestMean2 {

    //fact
    private static final double XO = 21;
    //idea
    private static final double MEAN = 3;
    //restriction
    private static final double DEVIATION = 0.8;
    //number of radom trials
    private static final int NTRIALS = 7435;
    //Significance level
    private static final double ALPHA = 0.05;
    //Turn on of the random generator
    static Random r = new Random();

    //Numbers generated at random
    private static final double EVENTS[] = new double[50000];
    //Discrepancy between fact and idea
    private static final double DISCREPANCY[] = new double[50000];
    private static double min;
    private static double max;
    //The left border of intervals, less or equal to the
    //minimum value of data.
    private static double infLimit;
    //Interval length
    private static double intervalLength;
    //Max number of allowed classes
    private static final int MAXNCLASSES = 100;
    //Borders of the intervals
    private static final double BARRIERS[] = new double[MAXNCLASSES + 1];
    private static final double CLASSMARKERS[] = new double[MAXNCLASSES];
    private static int nClasses = 20;
```

```
//Distribution of the discrepancies
private static final double FREQTABLE[][] = new double[MAXNCLASSES][2];

//Generates    random numbers
//with the requested normal distribution
//This is a virtual world that obeys the
//null hypothesis.
public static void virtualWorld(double mean, double deviation) {
    //We generate a random sequence
    for (int i = 0; i < NTRIALS; i++) {
        //Random numbers with Gaussian distribution
        //mean zero and deviation 1.
        double d = r.nextGaussian();
        // System.out.println(d ) ;
        // Rescale numbers: new mean = mean  and
        // new deviation = deviation
        EVENTS[i] = deviation * d + mean;
    }
}

//Measures the discrepancy between an event
//and an idea (the mean)
public static double discrepancy(double event,
        double mean,
        double deviation) {
    double d;
    //Measuring of discrepancy
    d = (event - mean) / deviation;
    return d;
}

//Measures the discrepancy between each fact of
//the virtual world and idea (the mean)
public static void discrepancy(double Events[],
        double mean) {
    double d;
    for (int i = 0; i < NTRIALS; i++) {
        //Measuring of discrepancy
        d = discrepancy(Events[i], mean, DEVIATION);
        DISCREPANCY[i] = d;
    }
}

//The range of discrepancies is evaluated
//and is used to calculate the length of each
//interval to group Discrepancies.
```

```
public static double intervalLength() {
    min = 100000;
    max = 0;
    for (int i = 0; i < NTRIALS; i++) {
        if (DISCREPANCY[i] < min) {
            min = DISCREPANCY[i];
        }
        if (DISCREPANCY[i] > max) {
            max = DISCREPANCY[i];
        }
    }
    System.out.println("min = " + min);
    System.out.println("max = " + max);
    double range = max - min;
    double length = range / nClasses;
    return length;
}

//Prints the frequency table
private static void printFreqTable(double FreqTable[][],
        int nClasses) {
    System.out.println("x and its absolute frequency");
    for (int i = 0; i < nClasses; i++) {
        System.out.println(FreqTable[i][0]
                + "\t " + FreqTable[i][1]);
    }
}

//Calculates the sum    of absolute frequencies
//of a frequency table
private static void
        test(double FreqTable[][]) {
    double sum = 0;
    for (int i = 0; i < nClasses; i++) {
        sum = sum + FreqTable[i][1];
    }
    System.out.println("Number of events = " + sum);
}

//The border of each interval is calculated
private static void getBorders(double Vect[]) {
    BARRIERS[0] = infLimit;
    CLASSMARKERS[0] = infLimit + intervalLength / 2;
    //Calculate maximal value of data
    double maxValue = Vect[0];
    for (int j = 1; j < Vect.length; j++) {
```

```
            if (Vect[j] > maxValue) {
                maxValue = Vect[j];
            }
        }
        System.out.println("Max value = " + max);
        //Calculates number of classes
        nClasses = (int) ((maxValue - infLimit) / intervalLength) + 1;
        System.out.println("Number of classes = " + nClasses);
        //Calculates interval borders and class markers
        for (int j = 1; j <= nClasses; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + intervalLength;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + intervalLength;
        }
        /*System.out.println( "Borders are" );
        printVector(Barriers);
        System.out.println( "Class markers are" );
        printVector(classMarkers);*/
    }

    //Data are grouped into interval classes.
    private static void group(double Vect[]) {
        intervalLength = intervalLength();
        infLimit = min - intervalLength / 2;
        getBorders(Vect);
        int nData = NTRIALS;
        //Class markers a re defined
        for (int j = 0; j < nClasses; j++) {
            FREQTABLE[j][0] = CLASSMARKERS[j];
            FREQTABLE[j][1] = 0;
        }
        //Frequencies are calculated.
        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < nClasses; j++) {
                if ((Vect[i] >= BARRIERS[j])
                        & (Vect[i] < BARRIERS[j + 1])) {
                    FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
                }
            }

        }
    }

    //The critical value in the upper tail is calculated
    //with significance level = ALPHA = 0.05
    private static double criticalValue(double Vect[]) {
        double c = 0;
```

```
        //Number of events that classify as extreme
        int e = (int) (NTRIALS * ALPHA);
        System.out.println("Number of extreme events = " + e);
        //Clon of Vect[]
        double Clon[] = new double[NTRIALS];
        /*
        for (int i = 0; i < NTRIALS; i++) {
        Clon[i] = Vect[i];
        }
         */
        System.arraycopy(Vect, 0, Clon, 0, NTRIALS);
        //Discrepancies are sorted
        //in decreasing order until completion of
        //e events.
        int Champ;
        for (int i = 0; i < e; i++) {
            Champ = 0;
            for (int j = 1; j < NTRIALS; j++) {
                if (Clon[j] >= Clon[Champ]) {
                    Champ = j;
                }
            }

            System.out.println(i + "th ind. is  No "
                    + Champ + " Discrep =  " + Clon[Champ]);
            c = Clon[Champ];
            Clon[Champ] = 0;
        }
        return c;
    }

    //Test: grouping must produce a Gauss'Bell
    private static void tableForExcel() {
        System.out.println("\nPaste next numbers to Excel, "
                + "LibreOffice or Gnumeric."
                + "\nMake a bar char."
                + "\nA Gauss'bell shall appear else there is a bug.");
        System.out.println("Absolute frequency of discrepancies:");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FREQTABLE[i][1]);
        }
    }

    /*Test with mean and deviation of events*/
    //Sums data in vector with real numbers
    private static double sumVector(double Vect[],
```

```
        int lim) {
    double sum = 0;
    for (int i = 0; i < lim; i++) {
        sum = sum + Vect[i];
    }
    return sum;
}

//The mean of a list of data
private static double meanVector(double Vect[],
        int lim) {
    double sum = sumVector(Vect, lim);
    int n = lim;
    double mean = sum / n;
    return mean;
}

//The variance of a list of data
private static double varVector(double Vect[],
        int lim) {
    //Mean
    double mean = meanVector(Vect, lim);
    // Variance is calculated by definition
    int n = lim;
    double sum2 = 0;
    for (int i = 0; i < n; i++) {
        sum2 = sum2 + (Vect[i] - mean) * (Vect[i] - mean);
    }
    double var = sum2 / (n - 1);
    return var;
}

//The mean and variance of a list of data
private static void meanVarVector(double DataVect[],
        int lim) {
    System.out.println("Sampled data");
    System.out.println("Data length = " + lim);
    double SampleMean = meanVector(DataVect, lim);
    System.out.println("Mean = " + SampleMean);
    double SampleVar = varVector(DataVect, lim);
    System.out.println("Variance \t " + SampleVar);
    double SampleDeviation = Math.pow(SampleVar, 0.5);
    System.out.println("Deviation = " + SampleDeviation);
    double coeffVar = SampleDeviation / SampleMean;
    System.out.println("Coefficient of variation = "
            + coeffVar);
```

```
        System.out.println("Expected mean = " + MEAN);
        System.out.println("Expected deviation = " + DEVIATION);
    }

    /*Test with mean and deviation of frequency table*/
    //Calculates the number of objects registered in
    //the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][],
            int nClasses) {
        double n = 0;
        for (int i = 0; i < nClasses; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

    //Calculates the sum XF   of
    //a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

    //Calculates the sum X2F   of
    //a table of absolute frequencies
    private static double
            sumX2FFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][0]
                    * FreqTable[i][1];
        }
        return sum;
    }

    //Calculates the mean   of
    //a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][],
            int nClasses) {
        double n = nFreqTable(FreqTable, nClasses);
```

```
    double mean = sumXFFreqTable(FreqTable, nClasses) / n;
    return mean;
}

//Calculates the variance of
//a table of absolute frequencies
private static double varFreqTable(double FreqTable[][],
        int nClasses) {
    double n = nFreqTable(FreqTable, nClasses);
    double Sxx = sumX2FFreqTable(FreqTable, nClasses)
            - Math.pow(sumXFFreqTable(FreqTable, nClasses), 2) / n;
    double var = Sxx / (n - 1);
    return var;
}

// Calculates the standard deviation of
//a table of absolute frequencies
private static double devFreqTable(double FreqTable[][],
        int nClasses) {
    double var = varFreqTable(FreqTable, nClasses);
    double dev = Math.pow(var, 0.5);
    return dev;
}

// Mean and variance of a frequency table
private static void meanVarTable(double FreqTable[][],
        int nClasses) {
    double n = nFreqTable(FreqTable, nClasses);
    System.out.println("Number of measured objects = " + n);
    double sumXF = sumXFFreqTable(FreqTable, nClasses);
    System.out.println("sum xF  = " + sumXF);
    double mean = meanFreqTable(FreqTable, nClasses);
    System.out.println("Mean = " + mean);
    double sumX2F = sumX2FFreqTable(FreqTable, nClasses);
    System.out.println("sum x2F  = " + sumX2F);
    double var = varFreqTable(FreqTable, nClasses);
    System.out.println("Variance = " + var);
    double deviation = devFreqTable(FreqTable, nClasses);
    System.out.println("Deviation =  " + deviation);
}

private static void testZone() {
    System.out.println("\n\nTest zone");
    //Mean and variance of events
    System.out.println("\nMean and Variance of events:");
    meanVarVector(EVENTS, NTRIALS);
```

```
        //Insert here a z-test for a mean and
        //a chi-square test for the variance.
        System.out.println("\nThe table of absolute "
                + "frequencies of discrepancies"
                + " \nmust represent"
                + " a normal distribution \nwith mean zero"
                + " and standard deviation one.");
        //Mean and variance of frequency table
        System.out.println("\nMean and Variance of Frequency"
                + " table of discrepancies: \nExpected values: \n"
                + "mean = 0 \nDeviation =   1");
        meanVarTable(FREQTABLE, nClasses);
        //Insert here a z-test for a mean and
        //a chi-square test for the variance.
        //Prints the absolute frequencies of discrepancies
        //they can be pasted to Excel, OpenOffice or Gnumeric
        tableForExcel();
        //Insert here a test for normality
    }

    public static void main(String[] args) {
        virtualWorld(MEAN, DEVIATION);
        System.out.println("Discrepancies");
        discrepancy(EVENTS, MEAN);
        group(DISCREPANCY);
        System.out.println("Frequency table of discrepancies");
        printFreqTable(FREQTABLE, nClasses);
        test(FREQTABLE);
        double criticalValue = criticalValue(DISCREPANCY);
        System.out.println("The expected critical value "
                + "is 1.56");
        System.out.println("Our critical value is = "
                + criticalValue);
        System.out.println("Experimental event = " + XO);
        //Discrepancy due to the    experiment
        double dExp = discrepancy(XO, MEAN, DEVIATION);
        System.out.println("Discrepancy due "
                + " to the experiment = " + dExp);
        if (dExp > criticalValue) {
            System.out.println("Extreme event according to Ho:"
                    + " invent another theory");
        } else {
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }
        testZone();
```

```
    }

}//End of Program E123 TestMean2
```

130, page 98. The next program runs a t-test in which the mechanistic arising of the t-distribution is shown.

```
//Program E130 TSimulation
//This program contrasts   a    fact
// and a null hypothesis, an idea.
//Fact:  the set of random data  of X
//3, 4, 5, 3, 5, 5, 4, 3
//has mean 4,
//Null hypothesis:
//X must follow a normal distribution with
//mean = 2.
//No assumption about the deviation of X is  done,
//so one must rely on the value of the deviation
//of the sample.
//The program contains some tests
//for correctness.


package ejvol5p;


import java.util.Random;

public class TSimulation {
    //fact =   experimental sample

    private static final double DATA[] = {3, 4, 5, 3, 5, 5, 4, 3};
    //Mean and deviation of exp sample
    private static double muHat, sigmaHat;
    //Size of the sample
    private static final int SAMPLESIZE = 8;
    //idea = null hypothesis, expected mean
    private static final double MEAN = 2;
    //restriction
    private static final double SIGMAX = 0.7;
    //size of the random sequence
    private static final int SIZESEQUENCE = 7435;
    //Significance level
```

```
private static final double ALPHA = 0.05;
//Turn on of the random generator
static Random r = new Random();
//A vector with a random sample
private static final double VECTSAMPLE[]
        = new double[SAMPLESIZE];
//Means   of sampleSize
//numbers generated at random
private static final double XBARS[] = new double[50000];
//Deviations of sampleSize
//numbers generated at random
private static final double DEVIATIONS[] = new double[50000];
//Discrepancy between fact and idea
private static final double DISCREPANCY[] = new double[50000];
private static double min;
private static double max;
//The left border of intervals, less or equal to the
//minimum value of data.
private static double infLimit;
//Interval length
private static double intervalLength;
//Max number of allowed classes
private static final int MAXNCLASSES = 100;
//Borders of the intervals
private static final double BARRIERS[]
        = new double[MAXNCLASSES + 1];
private static final double CLASSMARKERS[]
        = new double[MAXNCLASSES];
private static int nClasses = 20;
//Distribution of the discrepancies
private static final double FREQTABLE[][]
        = new double[MAXNCLASSES][2];

//Sums data in vector with real numbers
private static double sumVector(double Vect[],
        int lim) {
    double sum = 0;
    for (int i = 0; i < lim; i++) {
        sum = sum + Vect[i];
    }
    return sum;
}

//The mean of a list of data
private static double meanVector(double Vect[],
        int lim) {
```

```
    double sum = sumVector(Vect, lim);
    int n = lim;
    double myMean = sum / n;
    return myMean;
}

//The variance of a list of data
private static double varVector(double Vect[],
        int lim) {
    //Mean
    double myMean = meanVector(Vect, lim);
    // Variance is calculated by definition
    int n = lim;
    double sum2 = 0;
    for (int i = 0; i < n; i++) {
        sum2 = sum2 + (Vect[i] - myMean) * (Vect[i] - myMean);
    }
    double var = sum2 / (n - 1);
    return var;
}

//Generates   sampleSize  random numbers
//with the requested normal distribution.
public static void randomSample(double mean,
        double deviation,
        int sampleSize) {
    //We generate a random sequence
    //of size equal to that of the sample
    double event;
    for (int i = 0; i < sampleSize; i++) {
        //Random numbers with Gaussian distribution
        //mean zero and deviation 1.
        double d = r.nextGaussian();
        //System.out.println(d ) ;
        // Rescale numbers: mean = mean  and
        // deviation = deviation
        event = deviation * d + mean;
        //System.out.println(event ) ;
        VECTSAMPLE[i] = event;
    }
}

//We repeat lim times the following procedure:
//sampleSize random  numbers are generated,
//whose mean and deviation are kept in XBARS[]
//and DEVIATIONS[]
```

```
//This is a virtual world that obeys the
//null hypothesis.
public static void virtualWorld(double mean,
        double deviation,
        int sampleSize) {
    //We generate a random sequence
    //of size equal to that of the sample
    for (int i = 0; i < SIZESEQUENCE; i++) {
        //sampleSize random numbers with normal distribution
        randomSample(mean, deviation, sampleSize);
        //mean of sample
        double xBar = meanVector(VECTSAMPLE, sampleSize);
        //deviation
        double s = Math.sqrt(
                varVector(VECTSAMPLE, sampleSize));
        XBARS[i] = xBar;
        DEVIATIONS[i] = s;
    }
}

//Measures the discrepancy between an event
//and an idea (the mean)
public static double discrepancy(double event,
        double deviation,
        double mean) {
    double d;
    //Measuring of discrepancy
    d = (event - mean) / (deviation / Math.sqrt(SAMPLESIZE));
    return d;
}

//Measures the discrepancy between each fact of
//the virtual world and idea (the mean)
public static void discrepancy(double xbars[],
        double xdeviations[],
        double mean) {
    double d;
    for (int i = 0; i < SIZESEQUENCE; i++) {
        //Measuring of discrepancy
        d = discrepancy(xbars[i], xdeviations[i], mean);
        DISCREPANCY[i] = d;
    }
}

//The range of discrepancies is evaluated
//and is used to calculate the length of each
```

```
    //interval to group Discrepancies.
    public static double intervalLength() {
        min = 100000;
        max = 0;
        for (int i = 0; i < SIZESEQUENCE; i++) {
            if (DISCREPANCY[i] < min) {
                min = DISCREPANCY[i];
            }
            if (DISCREPANCY[i] > max) {
                max = DISCREPANCY[i];
            }
        }
        System.out.println("min = " + min);
        System.out.println("max = " + max);
        double range = max - min;
        double length = range / nClasses;
        return length;
    }

    //Prints the frequency table
    private static void printFreqTable(double FreqTable[][],
            int nClasses) {
        System.out.println("x and its absolute frequency");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

    //Calculates the sum    of absolute frequencies
    //of a frequency table
    private static void
            test(double FreqTable[][]) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][1];
        }
        System.out.println("\nNumber of events = " + sum);
    }

//The border of each interval is calculated
    private static void getBorders(double Vect[]) {
        BARRIERS[0] = infLimit;
        CLASSMARKERS[0] = infLimit + intervalLength / 2;
        //Calculate maximal value of data
        double myMax = Vect[0];
```

```
        for (int j = 1; j < Vect.length; j++) {
            if (Vect[j] > myMax) {
                myMax = Vect[j];
            }
        }
        System.out.println("Max value = " + myMax);
        //Calculates number of classes
        nClasses = (int) ((myMax - infLimit) / intervalLength) + 1;
        System.out.println("Number of classes = " + nClasses);
        //Calculates interval borders and class markers
        for (int j = 1; j <= nClasses; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + intervalLength;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + intervalLength;
        }
        /*System.out.println( "Borders are" );
 printVector(Barriers);
 System.out.println( "Class markers are" );
 printVector(classMarkers);*/
    }

//Data are grouped into interval classes.
    private static void group(double Vect[]) {
        intervalLength = intervalLength();
        infLimit = min - intervalLength / 2;
        getBorders(Vect);
        int nData = SIZESEQUENCE;
        //Class markers a re defined
        for (int j = 0; j < nClasses; j++) {
            FREQTABLE[j][0] = CLASSMARKERS[j];
            FREQTABLE[j][1] = 0;
        }
        //Frequencies are calculated.
        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < nClasses; j++) {
                if ((Vect[i] >= BARRIERS[j])
                        & (Vect[i] < BARRIERS[j + 1])) {
                    FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
                }
            }

        }
    }

    //The critical value with two  tails is calculated
    //with significance level = ALPHA = 0.05
    private static double criticalValue(double Vect[]) {
```

```
        double c = 0;
        //Number of events that classify as extreme
        //in either tail
        int e = (int) (SIZESEQUENCE * ALPHA / 2);
        System.out.println("Number of extreme events in "
                + "each tail = " + e);
        //Clon of Vect[]
        double Clon[] = new double[SIZESEQUENCE];
        /*
        for (int i = 0; i < SIZESEQUENCE; i++) {
        Clon[i] = Vect[i];
        }
         */
        System.arraycopy(Vect, 0, Clon, 0, SIZESEQUENCE);
        //Discrepancies are sorted
        //in decreasing order until completion of
        //e events.
        System.out.println("\nDiscrepancies are sorted in "
                + "\ndecreasing order until completion of demanded"
                + "\nnumber of extreme  events.");
        int Champ;
        for (int i = 0; i < e; i++) {
            Champ = 0;
            for (int j = 1; j < SIZESEQUENCE; j++) {
                if (Clon[j] >= Clon[Champ]) {
                    Champ = j;
                }
            }

            System.out.println(i + "th ind. is  No "
                    + Champ + " Discrep =  " + Clon[Champ]);
            c = Clon[Champ];
            Clon[Champ] = 0;
        }
        return c;
    }

    /*Test with mean and deviation of events*/
//The mean and variance of a list of data
    private static void meanVarVector(double DataVect[],
            int lim) {
        System.out.println("Sampled data");
        System.out.println("Data length = " + lim);
        double SampleMean = meanVector(DataVect, lim);
        System.out.println("Mean = " + SampleMean);
        double SampleVar = varVector(DataVect, lim);
```

```
        System.out.println("Variance \t " + SampleVar);
        double SampleDeviation = Math.pow(SampleVar, 0.5);
        System.out.println("Deviation = " + SampleDeviation);
        double coeffVar = SampleDeviation / SampleMean;
        System.out.println("Coefficient of variation = "
                + coeffVar);
        System.out.println("Expected mean = " + MEAN);
        System.out.println("Expected deviation = "
                + SIGMAX / Math.sqrt(SAMPLESIZE));
    }

    /*Test with mean and deviation of frequency table*/
//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][],
            int nClasses) {
        double n = 0;
        for (int i = 0; i < nClasses; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

//Calculates the sum XF   of
//a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the sum X2F   of
//a table of absolute frequencies
    private static double
            sumX2FFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][0]
                    * FreqTable[i][1];
        }
        return sum;
```

```
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][],
            int nClasses) {
        double n = nFreqTable(FreqTable, nClasses);
        double myMean = sumXFFreqTable(FreqTable, nClasses) / n;
        return myMean;
    }

//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][],
            int nClasses) {

        double n = nFreqTable(FreqTable, nClasses);
        double Sxx = sumX2FFreqTable(FreqTable, nClasses)
                - Math.pow(sumXFFreqTable(FreqTable, nClasses), 2) / n;
        double var = Sxx / (n - 1);
        return var;
    }

// Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][],
            int nClasses) {
        double var = varFreqTable(FreqTable, nClasses);
        double dev = Math.pow(var, 0.5);
        return dev;
    }

// Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][],
            int nClasses) {
        double n = nFreqTable(FreqTable, nClasses);
        System.out.println("Number of measured objects = " + n);
        double sumXF = sumXFFreqTable(FreqTable, nClasses);
        System.out.println("sum xF  = " + sumXF);
        double myMean = meanFreqTable(FreqTable, nClasses);
        System.out.println("Mean = " + myMean);
        double sumX2F = sumX2FFreqTable(FreqTable, nClasses);
        System.out.println("sum x2F  = " + sumX2F);
        double var = varFreqTable(FreqTable, nClasses);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable, nClasses);
```

```
        System.out.println("Deviation =  " + deviation);
    }

//Test: grouping must produce a wide Gauss'Bell
    private static void tableForExcel() {
        System.out.println("\nPaste next numbers to Excel, "
                + "LibreOffice or Gnumeric."
                + "\nMake a bar char."
                + "\nA wide Gauss'bell shall appear "
                + "else there is a bug.");
        System.out.println("Absolute frequency of discrepancies:");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FREQTABLE[i][1]);
        }
    }

    private static void testZone() {
        System.out.println("\n\nTest zone");
        //Mean and variance of events
        System.out.println("\nMean and Variance of sample mean:");
        meanVarVector(XBARS, SIZESEQUENCE);
        //Insert here a z-test for a mean and
        //a chi-square test for the variance.
        //Theoretical deviation of the t-distribution
        double nu = SAMPLESIZE - 1;
        double devT = Math.sqrt(nu / (nu - 2));
        System.out.println("\nThe table of absolute "
                + "frequencies of discrepancies"
                + " \nmust represent"
                + " a t-distribution \nwith mean zero"
                + " and standard deviation " + devT);
        //Mean and variance of frequency table
        System.out.println("\nMean and Variance of Frequency"
                + " table of discrepancies: \nExpected values: \n"
                + "mean = 0 \nDeviation = " + devT);
        meanVarTable(FREQTABLE, nClasses);
        //Insert here a z-test for a mean and
        //a chi-square test for the variance.
        //Prints the absolute frequencies of discrepancies
        //they can be pasted to Excel, OpenOffice or Gnumeric
        tableForExcel();
        //Insert here a test for normality
    }

    public static void main(String[] args) {
        virtualWorld(MEAN, SIGMAX, SAMPLESIZE);
```

```
        System.out.println("\nDiscrepancies");
        discrepancy(XBARS, DEVIATIONS, MEAN);
        group(DISCREPANCY);
        System.out.println("Frequency table of discrepancies");
        printFreqTable(FREQTABLE, nClasses);
        //Minor test for correct grouping
        test(FREQTABLE);
        double criticalValue = criticalValue(DISCREPANCY);
        //Test for critical value
        System.out.println("The expected critical value for "
                + "\n7 degrees of freedom is 2.365");
        System.out.println("Found upper critical value  = "
                + criticalValue);

        //Mean of experimental sample
        muHat = meanVector(DATA, SAMPLESIZE);
        //deviation of experimental sample
        System.out.println("Experimental sample mean = " + muHat);
        sigmaHat = Math.sqrt(
                varVector(DATA, SAMPLESIZE));
        //Discrepancy due to the    experiment
        double dExp = discrepancy(muHat, MEAN, sigmaHat);
        System.out.println("Discrepancy due  to the "
                + "experimental event = " + dExp);
        if (dExp > criticalValue) {
            System.out.println("The event is extreme according to Ho:"
                    + " invent another theory.");
        } else {
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }
        testZone();
    }
}//End of Program E130 TSimulation
```

132, page 100. The next program runs a test of a null hypothesis about a variance and shows the mechanistic arising of the *chi*-square-distribution:

```
//Program E132 Chi2Simulation
//This program contrasts   a    fact and an idea.
//Fact:
//the set of random data  of X
//3, 4, 5, 3, 5, 5, 4, 3
//has variance 0.85.
```

```
//Idea = null hypothesis:
// X must follow a normal distribution with
//variance  = 1.2
//and mean = mean of experimental sample.
//The program contains some tests
//for correctness.

package ejvol5p;

import java.util.Random;

public class Chi2Simulation {
    //fact =   experimental sample

    private static final double DATA[] = {3, 4, 5, 3, 5, 5, 4, 3};
    private static double varHat;
    //Size of the sample
    private static final int SAMPLESIZE = 8;
    //idea = null hypothesis, expected variance
    private static final double VARXZERO = 1.2;
    private static final double SIGMAX = Math.sqrt(VARXZERO);
    //restriction
    private static double mean;
    //number of trials in the virtual world
    private static final int NTRIALS = 7435;
    //Significance level
    private static final double ALPHA = 0.05;
    //Turn on of the random generator
    static Random r = new Random();
    //A list with a random sample
    private static final double VECTSAMPLE[]
            = new double[SAMPLESIZE];
    //Means of sampleSize
    //numbers generated at random
    private static final double XBARS[] = new double[50000];
    //Variances of sampleSize
    //numbers generated at random
    private static final double VAROFSAMPLE[] = new double[50000];
    //Discrepancy between fact and idea
    private static final double DISCREPANCY[] = new double[50000];
    private static double min;
    private static double max;
    //The left border of intervals, less or equal to the
    //minimum value of data.
    private static double infLimit;
    //Interval length
```

```
private static double intervalLength;
//Max number of allowed classes
private static final int MAXNCLASSES = 100;
//Borders of the intervals
private static final double BARRIERS[]
        = new double[MAXNCLASSES + 1];
private static final double CLASSMARKERS[]
        = new double[MAXNCLASSES];
private static int nClasses = 20;
//Distribution of the discrepancies
private static final double FREQTABLE[][]
        = new double[MAXNCLASSES][2];

//Sums data in vector with real numbers
private static double sumVector(double Vect[],
        int lim) {
    double sum = 0;
    for (int i = 0; i < lim; i++) {
        sum = sum + Vect[i];
    }
    return sum;
}

//The mean of a list of data
private static double meanVector(double Vect[],
        int lim) {
    double sum = sumVector(Vect, lim);
    int n = lim;
    double myMean = sum / n;
    return myMean;
}

//The variance of a list of data
private static double varVector(double Vect[],
        int lim) {
    //Mean
    double myMean = meanVector(Vect, lim);
    // Variance is calculated by definition
    int n = lim;
    double sum2 = 0;
    for (int i = 0; i < n; i++) {
        sum2 = sum2 + (Vect[i] - myMean) * (Vect[i] - myMean);
    }
    double var = sum2 / (n - 1);
    return var;
}
```

```
//Generates   sampleSize  random numbers
//with the requested normal distribution.
public static void randomSample(double mean,
        double deviation,
        int sampleSize) {
    //We generate a random sequence
    //of size equal to that of the sample
    double event;
    for (int i = 0; i < sampleSize; i++) {
        //Random numbers with Gaussian distribution
        //mean zero and deviation 1.
        double d = r.nextGaussian();
        //System.out.println(d ) ;
        // Rescale numbers: mean = mean  and
        // deviation = deviation
        event = deviation * d + mean;
        //System.out.println(event ) ;
        VECTSAMPLE[i] = event;
    }
}

//We repeat lim times the following procedure:
//sampleSize random  numbers are generated,
//whose mean and deviation are kept in simXBar[]
//and simXS[]
//This is a virtual world that obeys the
//null hypothesis.
public static void virtualWorld(double mean,
        double deviation,
        int sampleSize) {
    //We generate a random sequence
    //of size equal to that of the sample
    for (int i = 0; i < NTRIALS; i++) {
        //sampleSize random numbers with normal distribution
        randomSample(mean, deviation, sampleSize);
        //mean of sample
        double xBar = meanVector(VECTSAMPLE, sampleSize);
        //variance
        double xVariance = varVector(VECTSAMPLE, sampleSize);
        XBARS[i] = xBar;
        VAROFSAMPLE[i] = xVariance;
    }
}

//Measures the discrepancy between an event
```

```
//and an idea (the mean)
public static double discrepancy(double simXVar,
        double xVarZero) {
    double d;
    //Measuring of discrepancy
    d = (SAMPLESIZE - 1) * simXVar / xVarZero;
    return d;
}

//Measures the discrepancy between each fact of
//the virtual world and the idea (the expected variance)
public static void discrepancy(double simXVar[],
        double mean) {
    double d;
    for (int i = 0; i < NTRIALS; i++) {
        //Measuring of discrepancy
        d = discrepancy(simXVar[i], VARXZERO);
        DISCREPANCY[i] = d;
    }
}

//The range of discrepancies is evaluated
//and is used to calculate the length of each
//interval to group Discrepancies.
public static double intervalLength() {
    min = 100000;
    max = 0;
    for (int i = 0; i < NTRIALS; i++) {
        if (DISCREPANCY[i] < min) {
            min = DISCREPANCY[i];
        }
        if (DISCREPANCY[i] > max) {
            max = DISCREPANCY[i];
        }
    }
    System.out.println("min = " + min);
    System.out.println("max = " + max);
    double range = max - min;
    double length = range / nClasses;
    return length;
}

//Prints the frequency table
private static void printFreqTable(double FreqTable[][],
        int nClasses) {
    System.out.println("x and its absolute frequency");
```

```
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

    //Calculates the sum    of absolute frequencies
    //of a frequency table
    private static void
            test(double FreqTable[][]) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][1];
        }
        System.out.println("\nNumber of events = " + sum);
    }

    //The border of each interval is calculated
    private static void getBorders(double Vect[]) {
        BARRIERS[0] = infLimit;
        CLASSMARKERS[0] = infLimit + intervalLength / 2;
        //Calculate maximal value of data
        double myMax = Vect[0];
        for (int j = 1; j < Vect.length; j++) {
            if (Vect[j] > myMax) {
                myMax = Vect[j];
            }
        }
        System.out.println("Max value = " + myMax);
        //Calculates number of classes
        nClasses = (int) ((myMax - infLimit) / intervalLength) + 1;
        System.out.println("Number of classes = " + nClasses);
        //Calculates interval borders and class markers
        for (int j = 1; j <= nClasses; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + intervalLength;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + intervalLength;
        }
        /*System.out.println( "Borders are" );
        printVector(Barriers);
        System.out.println( "Class markers are" );
        printVector(classMarkers);*/
    }

    //Data are grouped into interval classes.
    private static void group(double Vect[]) {
        intervalLength = intervalLength();
```

```
    infLimit = min - intervalLength / 2;
    getBorders(Vect);
    int nData = NTRIALS;
    //Class markers a re defined
    for (int j = 0; j < nClasses; j++) {
        FREQTABLE[j][0] = CLASSMARKERS[j];
        FREQTABLE[j][1] = 0;
    }
    //Frequencies are calculated.
    for (int i = 0; i < nData; i++) {
        for (int j = 0; j < nClasses; j++) {
            if ((BARRIERS[j] <= Vect[i])
                    & (Vect[i] < BARRIERS[j + 1])) {
                FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
            }
        }

    }
}

//The critical value with two  tails is calculated
//with significance level = ALPHA = 0.05
private static double criticalValue(double Vect[]) {
    double c = 0;
    //Number of events that classify as extreme
    //in either tail
    int e = (int) (NTRIALS * ALPHA / 2);
    System.out.println("Number of extreme events in "
            + "each tail = " + e);
    //Clon of Vect[]
    double Clon[] = new double[NTRIALS];
    /*
    for (int i = 0; i < NTRIALS; i++) {
    Clon[i] = Vect[i];
    }
     */
    System.arraycopy(Vect, 0, Clon, 0, NTRIALS);
    //Discrepancies are sorted
    //in decreasing order until completion of
    //e events.
    System.out.println("\nDiscrepancies are sorted in "
            + "\ndecreasing order until completion of demanded"
            + "\nnumber of extreme  events.");
    int Champ;
    for (int i = 0; i < e; i++) {
        Champ = 0;
```

```
            for (int j = 1; j < NTRIALS; j++) {
                if (Clon[j] >= Clon[Champ]) {
                    Champ = j;
                }
            }

            System.out.println(i + "th ind. is  No "
                    + Champ + " Discrep =  " + Clon[Champ]);
            c = Clon[Champ];
            Clon[Champ] = 0;
        }
        return c;
    }

    /*Test with mean and deviation of events*/
    //The mean and variance of a list of data
    private static void meanVarVector(double DataVect[],
            int lim) {
        System.out.println("Sampled data");
        System.out.println("Data length = " + lim);
        double SampleMean = meanVector(DataVect, lim);
        System.out.println("Mean = " + SampleMean);
        double SampleVar = varVector(DataVect, lim);
        System.out.println("Variance = " + SampleVar);
        double SampleDeviation = Math.pow(SampleVar, 0.5);
        System.out.println("Deviation = " + SampleDeviation);
        double coeffVar = SampleDeviation / SampleMean;
        System.out.println("Coefficient of variation = "
                + coeffVar);
        System.out.println("Expected mean = " + mean);
        //Central limit theorem
        System.out.println("Expected deviation = "
                + SIGMAX / Math.sqrt(SAMPLESIZE));
    }

    /*Test with mean and deviation of frequency table*/
    //Calculates the number of objects registered in
    //the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][],
            int nClasses) {
        double n = 0;
        for (int i = 0; i < nClasses; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }
```

```
//Calculates the sum XF   of
//a table of absolute frequencies
private static double
        sumXFFreqTable(double FreqTable[][],
                int nClasses) {
    double sum = 0;
    for (int i = 0; i < nClasses; i++) {
        sum = sum + FreqTable[i][0] * FreqTable[i][1];
    }
    return sum;
}

//Calculates the sum X2F   of
//a table of absolute frequencies
private static double
        sumX2FFreqTable(double FreqTable[][],
                int nClasses) {
    double sum = 0;
    for (int i = 0; i < nClasses; i++) {
        sum = sum + FreqTable[i][0] * FreqTable[i][0]
                * FreqTable[i][1];
    }
    return sum;
}

//Calculates the mean   of
//a table of absolute frequencies
private static double meanFreqTable(double FreqTable[][],
        int nClasses) {
    double n = nFreqTable(FreqTable, nClasses);
    double myMean = sumXFFreqTable(FreqTable, nClasses) / n;
    return myMean;
}

//Calculates the variance of
//a table of absolute frequencies
private static double varFreqTable(double FreqTable[][],
        int nClasses) {

    double n = nFreqTable(FreqTable, nClasses);
    double Sxx = sumX2FFreqTable(FreqTable, nClasses)
            - Math.pow(sumXFFreqTable(FreqTable, nClasses), 2) / n;
    double var = Sxx / (n - 1);
    return var;
}
```

```
// Calculates the standard deviation of
//a table of absolute frequencies
private static double devFreqTable(double FreqTable[][],
        int nClasses) {
    double var = varFreqTable(FreqTable, nClasses);
    double dev = Math.pow(var, 0.5);
    return dev;
}

// Mean and variance of a frequency table
private static void meanVarTable(double FreqTable[][],
        int nClasses) {
    double n = nFreqTable(FreqTable, nClasses);
    System.out.println("Number of measured objects = " + n);
    double sumXF = sumXFFreqTable(FreqTable, nClasses);
    System.out.println("sum xF  = " + sumXF);
    double myMean = meanFreqTable(FreqTable, nClasses);
    System.out.println("Mean = " + myMean);
    double sumX2F = sumX2FFreqTable(FreqTable, nClasses);
    System.out.println("sum x2F  = " + sumX2F);
    double var = varFreqTable(FreqTable, nClasses);
    System.out.println("Variance = " + var);
    double deviation = devFreqTable(FreqTable, nClasses);
    System.out.println("Deviation =  " + deviation);
}

//Test: grouping must produce a typical chi-square
//distribution
private static void tableForExcel() {
    System.out.println("\nPaste next numbers to Excel, "
            + "OpenOffice or Gnumeric."
            + "\nMake a bar char."
            + "\nThe typical chi-square shape  shall appear "
            + "\nelse there is a bug.");
    System.out.println("Absolute frequency of discrepancies:");
    for (int i = 0; i < nClasses; i++) {
        System.out.println(FREQTABLE[i][1]);
    }
}

//Tests for correctness
private static void testZone() {
    System.out.println("\n\nTest zone");
    //Mean and variance of events
    System.out.println("\nMean and Variance of sample mean:");
```

```
    meanVarVector(XBARS, NTRIALS);
    //Insert here a z-test for a mean and
    //a chi-square test for the variance.
    //Theoretical mean and deviation of the
    //chi-2-distribution
    double nu = SAMPLESIZE - 1;
    double meanChi = nu;
    double devChi = Math.sqrt(2 * nu);
    System.out.println("\nThe table of absolute "
            + "frequencies of discrepancies"
            + " \nmust represent"
            + " a chi-distribution \nwith "
            + "\nexpected mean " + meanChi
            + " \nand standard deviation " + devChi);
    //Mean and variance of frequency table
    System.out.println("\nMean and Variance of Frequency"
            + " Table of discrepancies");
    meanVarTable(FREQTABLE, nClasses);
    //Prints the absolute frequencies of discrepancies
    //they can be pasted to Excel, OpenOffice or Gnumeric
    tableForExcel();
}

public static void main(String[] args) {
    mean = meanVector(DATA, SAMPLESIZE);
    System.out.println("\nDiscrepancies");
    virtualWorld(mean, SIGMAX, SAMPLESIZE);
    discrepancy(VAROFSAMPLE, VARXZERO);
    group(DISCREPANCY);
    System.out.println("Frequency table of discrepancies");
    printFreqTable(FREQTABLE, nClasses);
    //Minor test for correct grouping
    test(FREQTABLE);
    double criticalValue = criticalValue(DISCREPANCY);
    //Test for critical value
    System.out.println("The expected critical value for "
            + "\n7 degrees of freedom is 16.013");
    System.out.println("Found upper critical value  = "
            + criticalValue);
    //Variance of experimental sample
    varHat = varVector(DATA, SAMPLESIZE);
    System.out.println("Experimental sample variance = " + varHat);
    //Discrepancy due to the    experiment
    double dExp = discrepancy(varHat, VARXZERO);
    System.out.println("Discrepancy due  to the "
            + "experimental event = " + dExp);
```

```
        if (dExp > criticalValue) {
            System.out.println("The event is extreme according to Ho:"
                    + " invent another theory.");
        } else {
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }
        testZone();
    }
}//End of Program E132 Chi2Simulation
```

134, page 102. The next program runs a F- test in which the mechanistic arising of the F -distribution is shown.

```
//Program E134 FSimulation
//This program contrasts   a  null hypothesis
//about the ratio of two variances
//against experimental facts.
//Specifically, the facts are:
//the set of random data  of X
//6, 8, 7, 6, 5, 7, 8, 6, 9
//has variance 1.611;
//the set of random data Y
//3, 4, 5, 3, 5, 5, 4, 3
//has variance 0.85;
//the relation of variances is
//1.611 / 0.85 = 1.895.
//The null hypothesis  is
//that X and Y must follow  normal distributions with
// a relation among variances  =  R = 5.
//The mean of X is assumed to be 6.88 since that
//is the mean of the experimental sample.
//The mean of Y is assumed to be 4 by the same reason.
//The program contains some tests
//for correctness.
//It shows mechanistically the origin of the F distribution.


package ejvol5p;

import java.util.Random;

public class FSimulation {
    //fact =   experimental samples
```

```
private static final double DATA1[] = {6, 8, 7, 6, 5, 7, 8, 6, 9};
private static final double DATA2[] = {3, 4, 5, 3, 5, 5, 4, 3};

//Means and deviations of experimental samples
private static double muHat1, sigmaHat1, varHat1;
private static double muHat2, sigmaHat2, varHat2;
//Size of  samples
private static final int SAMPLESIZE1 = 9;
private static final int SAMPLESIZE2 = 8;
//Degrees of freedom
private static double nu1 = SAMPLESIZE1 - 1;
private static double nu2 = SAMPLESIZE2 - 1;
//idea = null hypothesis,
//expected ratio between  variances
private static final double R = 5;
//Ratio between observed variances
private static double rExp;
//Joint deviation
private static double sigmaHatJ;
//number of trials in the virtual world
private static final int NTRIALS = 74350;
//Significance level
private static final double ALPHA = 0.05;
//Turn on of the random generator
static Random r = new Random();
//A list with a random sample
private static final double VECTSAMPLE[]
        = new double[1000];
//Dimension of arrays
private static final int N = 100000;
//Means   of samples   generated at random
private static final double XBARS1[] = new double[N];
private static final double XBARS2[] = new double[N];
//Variances of samples  generated at random
private static final double VARS1[] = new double[N];
private static final double VARS2[] = new double[N];
//Discrepancy between fact and idea
private static final double DISCREPANCY[] = new double[N];
private static double min;
private static double max;
//The left border of intervals, less or equal to the
//minimum value of data.
private static double infLimit;
//Interval length
private static double intervalLength;
//Max number of allowed classes
```

```
private static final int MAXNCLASSES = 1000;
//Borders of the intervals
private static final double BARRIERS[]
        = new double[MAXNCLASSES + 1];
private static final double CLASSMARKERS[]
        = new double[MAXNCLASSES];
private static int nClasses = 250;
//Distribution of the discrepancies
private static final double FREQTABLE[][]
        = new double[MAXNCLASSES][2];

//Sums data in vector with real numbers
private static double sumVector(double Vect[],
        int lim) {
    double sum = 0;
    for (int i = 0; i < lim; i++) {
        sum = sum + Vect[i];
    }
    return sum;
}

//The mean of a list of data
private static double meanVector(double Vect[],
        int lim) {
    double sum = sumVector(Vect, lim);
    int n = lim;
    double mean = sum / n;
    return mean;
}

//The variance of a list of data
private static double varVector(double Vect[],
        int lim) {
    //Mean
    double mean = meanVector(Vect, lim);
    // Variance is calculated by definition
    int n = lim;
    double sum2 = 0;
    for (int i = 0; i < n; i++) {
        sum2 = sum2 + (Vect[i] - mean) * (Vect[i] - mean);
    }
    double var = sum2 / (n - 1);
    return var;
}

//Generates   sampleSize  random numbers
```

```
//with the requested normal distribution.
public static void randomSample(double mean,
        double deviation,
        int sampleSize) {
    //We generate a random sequence
    //of size equal to that of the sample
    double event;
    for (int i = 0; i < sampleSize; i++) {
        //Random numbers with Gaussian distribution
        //mean zero and deviation 1.
        double d = r.nextGaussian();
        //System.out.println(d ) ;
        // Rescale numbers: mean = mean  and
        // deviation = deviation
        event = deviation * d + mean;
        //System.out.println(event ) ;
        VECTSAMPLE[i] = event;
    }
}

//We repeat lim times the following procedure:
//sampleSize random  numbers are generated,
//whose mean and deviation are kept in simXBar[]
//and simXS[]. This is done twice for we
//have two populations.
//This is a virtual world that obeys the
//null hypothesis.
public static void virtualWorld() {
    //We run the virtual world lim times
    for (int i = 0; i < NTRIALS; i++) {
        //SAMPLESIZE1 random numbers with normal distribution
        randomSample(muHat1, Math.sqrt(R) * sigmaHatJ, SAMPLESIZE1);
        //mean of sample
        double xBar = meanVector(VECTSAMPLE, SAMPLESIZE1);
        //variance
        double xVariance = varVector(VECTSAMPLE, SAMPLESIZE1);
        XBARS1[i] = xBar;
        VARS1[i] = xVariance;
        //SAMPLESIZE2 random numbers with normal distribution
        //Variance2 = R times variance1
        randomSample(muHat2, sigmaHatJ, SAMPLESIZE2);
        //mean of sample
        xBar = meanVector(VECTSAMPLE, SAMPLESIZE2);
        //variance
        xVariance = varVector(VECTSAMPLE, SAMPLESIZE2);
        XBARS2[i] = xBar;
```

```
            VARS2[i] = xVariance;
        }
    }

    //Measures the discrepancy between observed r
    //and expected R
    public static double discrepancy(double r,
            double R) {
        double d;
        //Measuring of discrepancy
        d = r / R;
        return d;
    }

    //Measures the discrepancy between each fact of
    //the virtual world and the idea
    public static void discrepancy(double simVar1[],
            double simVar2[]) {
        double d;
        for (int i = 0; i < NTRIALS; i++) {
            //Measuring of discrepancy
            //between observed r and thought R
            double rr = simVar1[i] / simVar2[i];
            d = discrepancy(rr, R);
            DISCREPANCY[i] = d;
        }
    }

    //The range of discrepancies is evaluated
    //and is used to calculate the length of each
    //interval to group Discrepancies.
    public static double intervalLength() {
        min = 100000;
        max = 0;
        for (int i = 0; i < NTRIALS; i++) {
            if (DISCREPANCY[i] < min) {
                min = DISCREPANCY[i];
            }
            if (DISCREPANCY[i] > max) {
                max = DISCREPANCY[i];
            }
        }
        System.out.println("min = " + min);
        System.out.println("max = " + max);
        double range = max - min;
        double length = range / nClasses;
```

```
        return length;
    }

    //Prints the frequency table
    private static void printFreqTable(double FreqTable[][],
            int nClasses) {
        System.out.println("x and its absolute frequency");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

    //Calculates the sum    of absolute frequencies
    //of a frequency table
    private static void
            test(double FreqTable[][]) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][1];
        }
        System.out.println("\nNumber of events = " + sum);
    }

//The border of each interval is calculated
    private static void getBorders(double Vect[]) {
        BARRIERS[0] = infLimit;
        CLASSMARKERS[0] = infLimit + intervalLength / 2;
        //Calculate maximal value of data
        double myMax = Vect[0];
        for (int j = 1; j < Vect.length; j++) {
            if (Vect[j] > myMax) {
                myMax = Vect[j];
            }
        }
        System.out.println("Max value = " + myMax);
        //Calculates number of classes
        nClasses = (int) ((myMax - infLimit) / intervalLength) + 1;
        System.out.println("Number of classes = " + nClasses);
        //Calculates interval borders and class markers
        for (int j = 1; j <= nClasses; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + intervalLength;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + intervalLength;
        }
        /*System.out.println( "Borders are" );
 printVector(Barriers);
```

```
 System.out.println( "Class markers are" );
 printVector(classMarkers);*/
    }

//Data are grouped into interval classes.
    private static void group(double Vect[]) {
        intervalLength = intervalLength();
        infLimit = min - intervalLength / 2;
        getBorders(Vect);
        int nData = NTRIALS;
        //Class markers a re defined
        for (int j = 0; j < nClasses; j++) {
            FREQTABLE[j][0] = CLASSMARKERS[j];
            FREQTABLE[j][1] = 0;
        }
        //Frequencies are calculated.
        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < nClasses; j++) {
                if ((BARRIERS[j] <= Vect[i])
                        & (Vect[i] < BARRIERS[j + 1])) {
                    FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
                }
            }

        }
    }

    //The upper critical value with two  tails is calculated
    //with significance level = ALPHA = 0.05
    private static double criticalValueUp(double Vect[]) {
        double c = 0;
        //Number of events that classify as extreme
        //in either tail
        int e = (int) (NTRIALS * ALPHA / 2);
        System.out.println("Number of extreme events in "
                + "each tail = " + e);
        //Clon of Vect[]
        double Clon[] = new double[NTRIALS];
        /*
        for (int i = 0; i < NTRIALS; i++) {
            Clon[i] = Vect[i];
        }
        */
        System.arraycopy(Vect, 0, Clon, 0, NTRIALS);
        //Discrepancies are sorted
        //in decreasing order until completion of
```

```
    //e events.
    System.out.println("\nDiscrepancies are sorted in "
            + "\ndecreasing order until completion of demanded"
            + "\nnumber of extreme  events.");
    int Champ;
    for (int i = 0; i < e; i++) {
        Champ = 0;
        for (int j = 1; j < NTRIALS; j++) {
            if (Clon[j] >= Clon[Champ]) {
                Champ = j;
            }
        }
        //Instruction to print outliers
        /* System.out.println(  i + "th ind. is  No "
             + Champ + " Discrep =  " + Clon[Champ]);*/
        c = Clon[Champ];
        Clon[Champ] = 0;
    }
    return c;
}

//The lower critical value with two  tails is calculated
//with significance level = ALPHA = 0.05
private static double criticalValueDown(double Vect[]) {
    double c = 0;
    //Number of events that classify as extreme
    //in either tail
    int e = (int) (NTRIALS * ALPHA / 2);
    //Clon of Vect[]
    double Clon[] = new double[NTRIALS];
    /*
    for (int i = 0; i < NTRIALS; i++) {
    Clon[i] = Vect[i];
    }
     */
    System.arraycopy(Vect, 0, Clon, 0, NTRIALS);
    //Discrepancies are sorted
    //in increasing order until completion of
    //e events.
    int Champ;
    for (int i = 0; i < e; i++) {
        Champ = 0;
        for (int j = 1; j < NTRIALS; j++) {
            if (Clon[j] <= Clon[Champ]) {
                Champ = j;
            }
```

```
            }
            /*
   System.out.println(  i + "th ind. is  No "
                + Champ + " Discrep =  " + Clon[Champ]);*/
            c = Clon[Champ];
            Clon[Champ] = 10000000;
        }
        return c;
    }

    /*Test with mean and deviation of events*/
//The mean and variance of a list of data
    private static void meanVarVector(double DataVect[],
            int lim) {
        System.out.println("Sampled data");
        System.out.println("Data length = " + lim);
        double SampleMean = meanVector(DataVect, lim);
        System.out.println("Mean = " + SampleMean);
        double SampleVar = varVector(DataVect, lim);
        System.out.println("Variance = " + SampleVar);
        double SampleDeviation = Math.pow(SampleVar, 0.5);
        System.out.println("Deviation = " + SampleDeviation);
        double coeffVar = SampleDeviation / SampleMean;
        System.out.println("Coefficient of variation = "
                + coeffVar);
    }

    /*Test with mean and deviation of frequency table*/
//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][],
            int nClasses) {
        double n = 0;
        for (int i = 0; i < nClasses; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

//Calculates the sum XF   of
//a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
```

```
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the sum X2F   of
//a table of absolute frequencies
    private static double
            sumX2FFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][0]
                    * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][],
            int nClasses) {
        double n = nFreqTable(FreqTable, nClasses);
        double mean = sumXFFreqTable(FreqTable, nClasses) / n;
        return mean;
    }

//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][],
            int nClasses) {

        double n = nFreqTable(FreqTable, nClasses);
        double Sxx = sumX2FFreqTable(FreqTable, nClasses)
                - Math.pow(sumXFFreqTable(FreqTable, nClasses), 2) / n;
        double var = Sxx / (n - 1);
        return var;
    }

// Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][],
            int nClasses) {
        double var = varFreqTable(FreqTable, nClasses);
        double dev = Math.pow(var, 0.5);
        return dev;
```

```
    }

// Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][],
            int nClasses) {
        double n = nFreqTable(FreqTable, nClasses);
        System.out.println("Number of measured objects = " + n);
        double sumXF = sumXFFreqTable(FreqTable, nClasses);
        System.out.println("sum xF  = " + sumXF);
        double mean = meanFreqTable(FreqTable, nClasses);
        System.out.println("Mean = " + mean);
        double sumX2F = sumX2FFreqTable(FreqTable, nClasses);
        System.out.println("sum x2F  = " + sumX2F);
        double var = varFreqTable(FreqTable, nClasses);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable, nClasses);
        System.out.println("Deviation =  " + deviation);
    }

//Test: grouping must produce a typical chi-square
//distribution
    private static void tableForExcel() {
        System.out.println("\nPaste next numbers to Excel, "
                + "OpenOffice or Gnumeric."
                + "\nMake a bar char."
                + "\nThe typical F shape  shall appear "
                + "\nelse there is a bug."
                + "\nTry out various truncations.");
        System.out.println("Absolute frequency of discrepancies:");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FREQTABLE[i][1]);
        }
    }

//Tests for correctness
    private static void testZone() {
        System.out.println("\n\nTest zone");
        System.out.println("\n\nFirst random variable");
        //Mean and variance of events
        System.out.println("\nMean and Variance of sample mean:");
        meanVarVector(XBARS1, NTRIALS);
        System.out.println("Expected mean = " + muHat1);
        //Central limit theorem
        System.out.println("Expected deviation = "
                + Math.sqrt(R) * sigmaHatJ / Math.sqrt(SAMPLESIZE1));
        //Mean and variance of events
```

```
    System.out.println("\n\nSecond random variable");
    System.out.println("\nMean and Variance of sample mean:");
    meanVarVector(XBARS2, NTRIALS);
    System.out.println("Expected mean = " + muHat2);
    //Central limit theorem
    System.out.println("Expected deviation = "
            + sigmaHatJ / Math.sqrt(SAMPLESIZE2));
    //Theoretical mean and deviation of the
    //F-distribution
    double meanF = nu2 / (nu2 - 2);
    double varF = meanF * meanF * 2 * (nu1 + nu2 - 2) / (nu1 * (nu2 - 4));
    System.out.println("\nThe table of absolute "
            + "frequencies of discrepancies"
            + " \nmust represent"
            + " a F-distribution \nwith mean " + meanF
            + " and variance " + varF);
    System.out.println("Found values in simulation");
    //Mean and variance of frequency table
    meanVarTable(FREQTABLE, nClasses);
    //Prints the absolute frequencies of discrepancies
    //they can be pasted to Excel, OpenOffice or Gnumeric
    tableForExcel();
}

public static void meansAndVars() {
    //Mean of experimental sample1
    muHat1 = meanVector(DATA1, SAMPLESIZE1);
    //Variance of experimental sample
    varHat1 = varVector(DATA1, SAMPLESIZE1);
    //Deviation
    sigmaHat1 = Math.sqrt(varHat1);
    //Mean of experimental sample1
    muHat2 = meanVector(DATA2, SAMPLESIZE2);
    //Variance of experimental sample
    varHat2 = varVector(DATA2, SAMPLESIZE2);
    //Deviation
    sigmaHat2 = Math.sqrt(varHat2);
    System.out.println(" ");
    nu1 = SAMPLESIZE1 - 1;
    nu2 = SAMPLESIZE2 - 1;
    double a = (sigmaHat1 * nu1 + sigmaHat2 * nu2) / (nu1 + nu2 - 2);
    sigmaHatJ = Math.sqrt(a);
    System.out.println("Mean of X = " + muHat1);
    System.out.println("Var of  of X = " + varHat1);
    System.out.println("Deviation  of  of X = " + sigmaHat1);
    System.out.println("Mean of Y = " + muHat2);
```

```
        System.out.println("Var of  of Y = " + varHat2);
        System.out.println("Deviation  of  of Y = " + sigmaHat2);
    }

    public static void main(String[] args) {
        meansAndVars();
        virtualWorld();
        System.out.println("\nDiscrepancies");
        discrepancy(VARS1, VARS2);
        group(DISCREPANCY);
        System.out.println("Frequency table of discrepancies");
        printFreqTable(FREQTABLE, nClasses);
        //Minor test for correct grouping
        test(FREQTABLE);
        double criticalValueUp = criticalValueUp(DISCREPANCY);
        double criticalValueDown = criticalValueDown(DISCREPANCY);
        System.out.println("Experimental sample variance 1 = "
                + varHat1);
        System.out.println("Experimental sample variance 2 = "
                + varHat2);
        //Discrepancy due to the experiment
        rExp = varHat1 / varHat2;
        System.out.println("rExp = var1/var2 = " + rExp);
        System.out.println("Ho-expected ratio R = " + R);
        double dExp = discrepancy(rExp, R);
        System.out.println("Discrepancy due  to the "
                + "experimental event rExp/R  = " + dExp);
        //Test for critical value
        System.out.println("The expected upper critical value for \n"
                + nu1 + " degrees of freedom in the numerator and \n"
                + nu2 + " degrees of freedom in the denominator is 4.89"
                + "\nThe lower bound is 0.22"
        );
        System.out.println("Found upper critical value  = "
                + criticalValueUp);
        System.out.println("Found lower critical value  = "
                + criticalValueDown);
        if ((dExp > criticalValueUp) || (dExp < criticalValueDown)) {
            System.out.println("The event is extreme "
                    + " according to Ho: invent another theory.");
        } else {
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }
        testZone();
    }
```

```
}//End of Program E134 FSimulation
```

148, page 136. The simplified code follows:

```
/*Program E148 KolSmirTest2
 We simulate a Kolmogorov-Smirnov test
 in which a null hypothesis for normality is checked out.
 Procedure:
 We group raw experimental data into a frequency table T.
 We  use  table T to calculate
 the mean,  variance and deviation of the sample.
 We calculate the discrepancy due to the experimental
 value DExp as follows:
 we construct   the
 cumulative function of data Phi(x) together with
 the cumulative distribution of the standard normal
 distribution Psi(z(x)), which is the null hypothesis.
 We measure the discrepancy between observed Phi(x) and
 expected Psi(z(x)) according to
 | Phi(x)-Psi(z(x)) |
 and take the maximal value as the measure of the
 discrepancy DExp.
 Next, we generate a random sequence of exactly
 the same size as that of the experimental sample
 and with a normal distribution with mean and
 deviation equal  to the mean and deviation of
 table T. We group simulated data following the
 same protocol that was made to produce table  T.
 Next, we measure the discrepancy between observed
 in the virtual world and expected under Ho by
 exactly the same procedure used to calculate DExp.
  We repeat  this procedure many times
 to estimate the distribution of  D  and to define in
 agreement with the chosen level of significance
 what is normal and what is   outlier.
 Our test is one tailed since we take absolute value
 and are interested in large discrepancies only.
 At last, we judge the discrepancy due to the experiment:
 if it normal, we accept the null hypothesis
 but  if it is outlier, we reject it.
 This is a simplified version of program E146
 */

package ejvol5p;
```

```
import java.util.Random;

public class KolSmirTest2 {
    //Experimental data

    private static final double EXPDATA[] = {
        145, 164, 171, 183, 177, 184, 168, 163, 157, 172,
        179, 168, 172, 170, 172, 157, 162, 159, 164, 158,
        173, 181, 162, 159, 167, 162, 166, 147, 163, 152,
        156, 150, 170, 149, 180, 148, 163, 168, 169, 158,
        150, 158, 174, 179, 168, 167, 182, 174, 168, 160,
        180, 152, 160, 160, 148, 155, 173, 151, 149, 169,
        161, 155, 160, 176, 177, 170, 169, 165, 146, 175,
        156, 166
    };

//The left border of intervals, less or equal to the
//minimum value of data.
    private static double infLimit = 145;
//Interval length
    private static double intervalLength = 10;
//Max number of allowed classes
    private static final int MAXNCLASSES = 1000;
//Borders of tallness intervals
    private static final double BARRIERSTALLNESS[]
            = {145, 155, 165, 175, 185};
    private static final double CLASSMARKERSTALLNESS[]
            = {150, 160, 170, 180};
    private static int nClasses;
    private static final double FREQTABLE[][]
            = new double[MAXNCLASSES][2];
//Observed cumulative distribution
    private static final double PHI[]
            = new double[MAXNCLASSES];
//Expected cumulative distribution
    private static final double PSI[]
            = {0.059, 0.301, 0.699, 0.941};
//Parameters according to Ho
    private static double muHat;
    private static double sigmaHat;
//Size of  sample
    private static final int SAMPLESIZE = 72;
//number of trials in the virtual world
    private static final int NTRIALS = 87650;
//Significance level
    private static final double ALPHA = 0.05;
```

```
//Turn on of the random generator
    static Random r = new Random();
//A list with a random sample
    private static final double VECTSAMPLE[]
            = new double[1000];
//Dimension of arrays
    private static final int N = 100000;
//Means   of simulated samples   generated at random
    private static final double SIMSAMPLES[] = new double[N];
//Variances of simulated samples  generated at random
//    private static final double SIMVAR[] = new double[N];
//Discrepancy due to experiment
    private static double dExp;
//Discrepancy between fact and idea
    private static final double DISCREPANCY[] = new double[N];
//Borders of the intervals
    private static final double BARRIERS[]
            = new double[MAXNCLASSES + 1];
    private static final double CLASSMARKERS[]
            = new double[MAXNCLASSES];
    private static double min;
    private static double max;
    private static boolean print;

//Sums data in vector with real numbers
    private static double sumVector(double Vect[],
            int n) {
        double sum = 0;
        for (int i = 0; i < n; i++) {
            sum = sum + Vect[i];
        }
        return sum;
    }

//The mean of a list of data
    private static double meanVector(double Vect[],
            int n) {
        double sum = sumVector(Vect, n);
        double mean = sum / n;
        return mean;
    }

//The variance of a list of data
    private static double varVector(double Vect[],
            int n) {
        //Mean
```

```
        double mean = meanVector(Vect, n);
        // Variance is calculated by definition
        double sum2 = 0;
        for (int i = 0; i < n; i++) {
            sum2 = sum2 + (Vect[i] - mean) * (Vect[i] - mean);
        }
        double var = sum2 / (n - 1);
        return var;
    }

//The mean and variance of a list of data
    private static void meanVarVector(double DataVect[],
            int lim) {
        System.out.println("Data length = " + lim);
        double SampleMean = meanVector(DataVect, lim);
        System.out.println("Mean = " + SampleMean);
        double SampleVar = varVector(DataVect, lim);
        System.out.println("Variance = " + SampleVar);
        double SampleDeviation = Math.pow(SampleVar, 0.5);
        System.out.println("Deviation = " + SampleDeviation);
        double coeffVar = SampleDeviation / SampleMean;
        System.out.println("Coefficient of variation = "
                + coeffVar);
    }

//Prints data in vector
    private static void printVector(double Vect[], int n) {
        for (int i = 0; i < n; i++) {
            System.out.println(i + "   " + Vect[i]);
        }
    }

//Prints the frequency table
    private static void printFreqTable(double FreqTable[][],
            int nClasses) {
        System.out.println("x and its absolute frequency");
        for (int i = 0; i <= nClasses; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

//Calculates the number of objects registered in
//the table of absolute frequencies
    private static double nFreqTable(double FreqTable[][],
            int nClasses) {
```

```
        double n = 0;
        for (int i = 0; i < nClasses; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

//Calculates the sum XF   of
//a table of absolute frequencies
    private static double
            sumXFFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the sum X2F   of
//a table of absolute frequencies
    private static double
            sumX2FFreqTable(double FreqTable[][],
                    int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][0]
                    * FreqTable[i][1];
        }
        return sum;
    }

//Calculates the mean   of
//a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][],
            int nClasses) {
        double n = nFreqTable(FreqTable, nClasses);
        double mean = sumXFFreqTable(FreqTable, nClasses) / n;
        return mean;
    }

//Calculates the variance of
//a table of absolute frequencies
    private static double varFreqTable(double FreqTable[][],
            int nClasses) {
```

```
        double n = nFreqTable(FreqTable, nClasses);
        double Sxx = sumX2FFreqTable(FreqTable, nClasses)
                - Math.pow(sumXFFreqTable(FreqTable, nClasses), 2) / n;
        double var = Sxx / (n - 1);
        return var;
    }

//Calculates the standard deviation of
//a table of absolute frequencies
    private static double devFreqTable(double FreqTable[][],
            int nClasses) {
        double var = varFreqTable(FreqTable, nClasses);
        double dev = Math.pow(var, 0.5);
        return dev;
    }

//Calculates the coefficient of variation of
//a table of absolute frequencies
    private static double coefficient(double FreqTable[][]) {
        double c = devFreqTable(FreqTable, nClasses)
                / meanFreqTable(FreqTable, nClasses);
        return c;
    }

// Mean and variance of a frequency table
    private static void meanVarTable(double FreqTable[][],
            int nClasses) {
        printFreqTable(FreqTable, nClasses);
        double n = nFreqTable(FreqTable, nClasses);
        System.out.println("Number of measured objects = " + n);
        double sumXF = sumXFFreqTable(FreqTable, nClasses);
        System.out.println("sum xF  = " + sumXF);
        double mean = meanFreqTable(FreqTable, nClasses);

        System.out.println("Mean = " + mean);
        double sumX2F = sumX2FFreqTable(FreqTable, nClasses);
        System.out.println("sum x2F  = " + sumX2F);
        double var = varFreqTable(FreqTable, nClasses);
        System.out.println("Variance = " + var);
        double deviation = devFreqTable(FreqTable, nClasses);
        System.out.println("Deviation =  " + deviation);
        double coefficient = coefficient(FreqTable);
        System.out.println("Coefficient of variation =  "
                + coefficient);
    }
```

```
//***********Grouping*****************************
//Data are grouped into interval classes.
    private static void group(double Vect[],
            double classMarkers[],
            double Barriers[],
            int nClasses, int nEvents) {

//Class markers are defined
        for (int j = 0; j < nClasses; j++) {
            FREQTABLE[j][0] = classMarkers[j];
            FREQTABLE[j][1] = 0;
        }
//Frequencies are calculated.
        for (int i = 0; i < nEvents; i++) {
            for (int j = 0; j < nClasses; j++) {
                if ((Barriers[j] <= Vect[i])
                        & (Vect[i] < Barriers[j + 1])) {
                    FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
                }
            }
        }
    }

//***********Measuring of discrepancy************
//Cumulative distribution of observed frequency table
    public static void Phi(double FreqTable[][],
            int nClasses) {
        double sum = 0;
        for (int i = 0; i < nClasses; i++) {
            sum = sum + FreqTable[i][1] / SAMPLESIZE;
            PHI[i] = sum;
        }
    }

//Measures the discrepancy between observed PHI(x)
//and expected PSI(z(x))
    public static double discrepancy(double Phi[],
            int nClasses, String s) {
        double D = 0;
        double d;
        if (print) {
            System.out.println("\nLine per line discrepancies");
        }
        for (int i = 0; i < nClasses; i++) {
            d = Math.abs(Phi[i] - PSI[i]);
            if (print) {
```

```
                System.out.println(i + "   " + d);
            }
            if (d > D) {
                D = d;
            }
        }
        if (print) {
            System.out.println("Discrepancy due to "
                    + s + " =  " + D);
        }
        return D;
    }

//*************Simulation zone***************
//Generates   SAMPLESIZE  random numbers
//with the requested normal distribution.
    public static void randomSample(double mean,
            double deviation,
            int sampleSize) {
        //We generate a random sequence
        //of size equal to that of the sample
        double event;
        for (int i = 0; i < sampleSize; i++) {
            //Random numbers with Gaussian distribution
            //mean zero and deviation 1.
            double d = r.nextGaussian();
            //System.out.println(d ) ;
            // Rescale numbers: mean = mean  and
            // deviation = deviation
            event = deviation * d + mean;
            //System.out.println(event ) ;
            VECTSAMPLE[i] = event;
        }
    }

//Extreme events are associated to available extreme
//classes and output data are grouped
    public static void GroupAndCutTails(double vectSample[],
            int sampleSize) {
        //Extreme events are smeared out
        double Min = infLimit;
        double Max = infLimit + nClasses * intervalLength;
        for (int i = 0; i < sampleSize; i++) {
            if (vectSample[i] < Min) {
                vectSample[i] = Min;
            }
```

```
            if (vectSample[i] > Max) {
                vectSample[i] = Max;
            }
        }
        //Data are grouped
        group(vectSample, CLASSMARKERSTALLNESS, BARRIERSTALLNESS,
                4, sampleSize);
    }

    /*We repeat lim times the following procedure:
sampleSize random  numbers are generated.
Next, those numbers are grouped
in a frequency table T with the same form as
experimental data. This implies taking care of outliers.
Cumulative distribution for T is calculated
and the discrepancy that is expected by the null hypothesis
is measured. Result is kept in DISCREPANCY[].
     */
    public static void virtualWorld() {
        //We run the virtual world lim times
        for (int i = 0; i < NTRIALS; i++) {
            //sampleSize1 random numbers with normal distribution
            randomSample(muHat, sigmaHat, SAMPLESIZE);
            //mean of sample to be used in a test
            double xBar = meanVector(VECTSAMPLE, SAMPLESIZE);
            //variance
            //double xVariance = varVector(VECTSAMPLE, SAMPLESIZE);
            SIMSAMPLES[i] = xBar;
            //SIMVAR[i] = xVariance;
            GroupAndCutTails(VECTSAMPLE, SAMPLESIZE);
            Phi(FREQTABLE, nClasses);
            double D = discrepancy(PHI, nClasses, " simulation");
            DISCREPANCY[i] = D;
        }
    }

//The range of discrepancies is evaluated
//and is used to calculate the length of each
//interval to group Discrepancies.
    public static double intervalLength() {
        min = 100000;
        max = 0;
        for (int i = 0; i < NTRIALS; i++) {
            if (DISCREPANCY[i] < min) {
                min = DISCREPANCY[i];
            }
```

```
            if (DISCREPANCY[i] > max) {
                max = DISCREPANCY[i];
            }
        }
        System.out.println("min = " + min);
        System.out.println("max = " + max);
        double range = max - min;
        double length = range / nClasses;
        return length;
    }

//Calculates the sum    of absolute frequencies
//of a frequency table
    private static void
            test(double FreqTable[][]) {
        double sum = 0;
        for (int i = 0; i <= nClasses; i++) {
            sum = sum + FreqTable[i][1];
        }
        System.out.println("Number of events = " + sum);
    }

//The border of each interval is calculated
    private static void getBorders(double Vect[]) {
        BARRIERS[0] = infLimit;
        CLASSMARKERS[0] = infLimit + intervalLength / 2;
//Calculate maximal value of data
        double myMax = Vect[0];
        for (int j = 1; j < Vect.length; j++) {
            if (Vect[j] > myMax) {
                myMax = Vect[j];
            }
        }
        System.out.println("Max value = " + myMax);
//Calculates number of classes
        nClasses = (int) ((myMax - infLimit) / intervalLength) + 1;
        System.out.println("Number of classes = " + nClasses);
//Calculates interval borders and class markers
        for (int j = 1; j <= nClasses; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + intervalLength;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + intervalLength;
        }
        /*System.out.println( "Borders are" );
printVector(Barriers);
System.out.println( "Class markers are" );
printVector(CLASSMARKERS);*/
```

```
    }

//Data are grouped into interval classes.
    private static void groupMake(double Vect[],
            int nClasses,
            int lim) {
        intervalLength = intervalLength();
        infLimit = min - intervalLength / 2;
        getBorders(Vect);
        group(Vect, CLASSMARKERS, BARRIERS, nClasses + 1, lim);
    }

//The upper critical value with two  tails is calculated
//with significance level = ALPHA = 0.05
    private static double criticalValueUp(double Vect[]) {
        double c = 0;
        //Number of events that classify as extreme
        //in either tail
        int e = (int) (NTRIALS * ALPHA);
        System.out.println("Number of extreme events in "
                + "upper tail = " + e);
        //Clon of Vect[]
        double Clon[] = new double[NTRIALS];
        /*
        for (int i = 0; i < NTRIALS; i++) {
        Clon[i] = Vect[i];
        }
         */
        System.arraycopy(Vect, 0, Clon, 0, NTRIALS);
        //Discrepancies are sorted
        //in decreasing order until completion of
        //e events.
        System.out.println("\nDiscrepancies are sorted in "
                + "\ndecreasing order until completion of demanded"
                + "\nnumber of extreme  events.");
        int Champ;
        for (int i = 0; i < e; i++) {
            Champ = 0;
            for (int j = 1; j < NTRIALS; j++) {
                if (Clon[j] >= Clon[Champ]) {
                    Champ = j;
                }
            }

            /* System.out.println(  i + "th ind. is  No "
               + Champ + " Discrep =  " + Clon[Champ]);*/
```

```
            c = Clon[Champ];
            Clon[Champ] = 0;
        }
        return c;
    }


    /*Test with mean and deviation of events*/
 /*Test with mean and deviation of frequency table*/
//Test: grouping must produce a central tendency
//distribution
    private static void tableForExcel() {
        System.out.println("\nPaste next numbers to Excel, "
                + "OpenOffice or Gnumeric."
                + "\nMake a bar char."
                + "\nA central tendency is  expected "
                + "\nelse there is a bug."
                + "\n");
        System.out.println("Absolute frequency "
                + "of discrepancies:");
        for (int i = 0; i < nClasses; i++) {
            System.out.println(FREQTABLE[i][1]);
        }
    }

//Tests for correctness
    private static void testZone() {
        System.out.println("\n\nTest zone");
        //Mean and variance of events
        System.out.println("\nMean and Variance "
                + "of sample mean:");
        meanVarVector(SIMSAMPLES, NTRIALS);

        System.out.println("Expected mean = " + muHat);
        //Central limit theorem
        System.out.println("Expected deviation = "
                + sigmaHat / Math.sqrt(SAMPLESIZE));
        //Prints the absolute frequencies of discrepancies
        //they can be pasted to Excel, OpenOffice or Gnumeric
        tableForExcel();
    }

    public static void main(String[] args) {
        System.out.println("**** EXPERIMENT ****");
        System.out.println("\nUngrouped  data: \n");
        meanVarVector(EXPDATA, EXPDATA.length);
```

```
System.out.println("\nGrouped experimental data: \n");
nClasses = 4;
intervalLength = 10;
min = infLimit;
group(EXPDATA, CLASSMARKERSTALLNESS,
        BARRIERSTALLNESS, 4, SAMPLESIZE);
//Mean is captured
muHat = meanFreqTable(FREQTABLE, nClasses);
//Deviation is captured
sigmaHat = devFreqTable(FREQTABLE, nClasses);
meanVarTable(FREQTABLE, nClasses);
//Cumulative distribution for experiment
Phi(FREQTABLE, nClasses);
System.out.println("\nObserved cumulative Distribution");
printVector(PHI, nClasses);
System.out.println("\n**** EXPECTED UNDER Ho ****");
System.out.println("\nExpected Cumulative Distribution.");
printVector(PSI, nClasses);
//Discrepancy due to the experiment
dExp = discrepancy(PHI, nClasses, "Experiment");
System.out.println("\n**** OBSERVED DISCREPANCY ****");
System.out.println("\nDiscrepancy due  to the "
        + "experimental event   = " + dExp);

//**********SIMULATION*******
System.out.println("\n**** SIMULATION ****");
print = false;
//Discrepancies in the virtual world
virtualWorld();
//MODIFY THIS PARAMETER
nClasses = 20;
System.out.println("NTRIALS = " + NTRIALS);
groupMake(DISCREPANCY, nClasses, NTRIALS);
System.out.println("Frequency table of discrepancies");
printFreqTable(FREQTABLE, nClasses);
meanVarTable(FREQTABLE, nClasses);
//Minor test for correct grouping
test(FREQTABLE);
double criticalValueUp = criticalValueUp(DISCREPANCY);
testZone();
System.out.println("Discrepancy due  to the "
        + "experimental event   = " + dExp);
//Test for critical value
System.out.println("The expected upper critical "
        + "\nvalue  (with infinitely many classes) = 0.16"
);
```

```
        System.out.println("Found upper critical value with "
                + nClasses + " classes  = " + criticalValueUp);

        if ((dExp > criticalValueUp)) {
            System.out.println("The event is extreme "
                    + " according to Ho: invent another theory.");
        } else {
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }

    }

}//End of Program E148 KolSmirTest2
```

155, page 139. Answer: The chi-square test for our random generator follows:

```
//Program E155 WhateverD2
//The program generates random integers
//with a given distribution.
//A chi^2 test is executed to asses
//the quality of the simulation.

package ejvol5p;


import java.util.Random;

public class WhateverD2 {
    // Absolute frequencies are defined:
    //The frequency of 0 is 3, that of 1 is 2,
    //that of 3 is zero and that of four is 5.

    private static final int DIST[] = {3, 2, 0, 5};
    private static final int NCLASSES = DIST.length;
    private static double sizeSample;
    private static final int SIZESIMULATION = 10000;
    // Counters of simulated events are in count
    private static final double COUNT[] = new double[NCLASSES];
    //Predicted distribution
    private static final double PREDICTED[] = new double[NCLASSES];
    //Discrepancies line per line
    private static final double CRITICALCHI2 = 7.8;
```

```
private static final double DISCREPANCY[] = new double[NCLASSES];
// Turn on of the random generator
private static final Random R = new Random();

//Prints data in vector
private static void printVector(double Vect[]) {
    for (int i = 0; i < Vect.length; i++) {
        System.out.println(Vect[i]);
    }
}

private static void printTwoVectors(double Vect1[], double Vect2[]) {
    for (int i = 0; i < Vect1.length; i++) {
        System.out.println(Vect1[i] + "   " + Vect2[i]);
    }
}

//Returns the size of the sample
private static double sizeSample() {
    double s = 0;
    for (int i = 0; i < NCLASSES; i++) {
        if (DIST[i] > 0) {
            s = s + DIST[i];
        }
    }
    return s;
}

//Predicts what must be observed
private static void prediction() {
    for (int i = 0; i < NCLASSES; i++) {
        double probSubI = DIST[i] / sizeSample;
        PREDICTED[i] = probSubI * SIZESIMULATION;
    }
}

//Runs a chi^2-test to check the accuracy
//of our source of random numbers
//that must fit the distribution given by DIST
private static double chiTest() {
    double d = 0;
    for (int i = 0; i < NCLASSES; i++) {
        if (PREDICTED[i] > 0) {
            DISCREPANCY[i]
                    = Math.pow((COUNT[i] - PREDICTED[i]), 2)
                    / PREDICTED[i];
```

```
            }
            d = d + DISCREPANCY[i];
        }
        return d;
    }

    public static void main(String[] args) {
        for (int i = 0; i < DIST.length; i++) {
            COUNT[i] = 0;
        }
        for (int i = 0; i < SIZESIMULATION; i++) {
            //A random integer less than 10 is generated
            //with a uniform distribution
            int n = R.nextInt(10);
            //Outputs are renamed
            //Outputs are renamed
            if (n < 3) {
                COUNT[0] = COUNT[0] + 1;
                n = 0;
            }
            if ((3 <= n) & (n < 5)) {
                COUNT[1] = COUNT[1] + 1;
                n = 1;
            }
            if ((5 <= n) & (n < 10)) {
                COUNT[3] = COUNT[3] + 1;
                n = 3;
            }
            //System.out.println( n);
        }

        sizeSample = sizeSample();
        prediction();
        System.out.println("Observed  and predicted values");
        printTwoVectors(COUNT, PREDICTED);
        double dExp = chiTest();
        System.out.println("\nHo: observed values fit "
                + "predicted ones.");
        System.out.println("\nDiscrepancies");
        printVector(DISCREPANCY);
        System.out.println("Experimental discrepancy = " + dExp);
        System.out.println("Critical chi2 = " + CRITICALCHI2);
        if (dExp > CRITICALCHI2) {
            System.out.println("The event is extreme "
                    + " according to Ho: invent another theory.");
        } else {
```

```
            System.out.println("Normal event according to Ho:"
                    + "there is no reason to reject Ho");
        }
    }//End of main
}//End of Program E155 WhateverD2
```

159, page 144. Uniformity test for a generator of random numbers that must fit a continuous distribution.

```
//Program E159 WhateverCTest
//A source generates random number of type double
//with a given distribution.
//but with the condition that the   numbers must have
//a uniform distribution in within each one of  intervals
//whose borders are the classes of the distribution.
//A test for this claim is included,
//whose output must be graphicated.

package ejvol5v3p;

import java.util.Random;

public class WhateverCTest {

    //Absolute frequencies are defined:
    //The frequency of 0 is 3, that of 1 is 2,
    //that of 3 is zero and that of four is 5.
    private static final int DIST[] = {3, 2, 0, 5};
    private static  int nClasses = DIST.length;
    private static double sizeSample;
    private static final int SIZESIMULATION = 100000;
    private static final double EVENTS[]
            = new double[SIZESIMULATION];
    //Max number of allowed classes
    private static final int MAXNCLASSES = 1000;
    //Counters of simulated events are in count
    private static final double COUNT[] = new double[MAXNCLASSES];
    //Predicted distribution
    private static final double PREDICTED[] = new double[MAXNCLASSES];
    //Discrepancies line per line
    private static final double CRITICALCHI2 = 7.8;
    private static final double DISCREPANCY[] = new double[MAXNCLASSES];
    //Turn on of the random generator
    private static final Random R = new Random();
```

```
    private static final boolean PRINTALL = true;

    private static double min;
    private static double max;
    //The left border of intervals, less or equal to the
    //minimum value of data.
    private static double infLimit;
    //Interval length
    private static double intervalLength;
    //Borders of the intervals
    private static final double BARRIERS[]
            = new double[MAXNCLASSES + 1];
    private static final double CLASSMARKERS[]
            = new double[MAXNCLASSES];

    //Distribution of the discrepancies
    private static final double FREQTABLE[][]
            = new double[MAXNCLASSES][2];

    //Prints data in vector
    private static void printVector(double Vect[]) {
        for (int i = 0; i < nClasses; i++) {
            System.out.println(i + "    " + Vect[i]);
        }
    }


    private static void printTwoVectors(double Vect1[], double Vect2[]) {
        for (int i = 0; i < nClasses; i++) {
            System.out.println(Vect1[i] + "   " + Vect2[i]);
        }
    }


    //Returns the size of the sample
    private static double sizeSample() {
        double s = 0;
        for (int i = 0; i < nClasses; i++) {
            if (DIST[i] > 0) {
                s = s + DIST[i];
            }
        }
        return s;
    }

    //PREDICTEDs what must be observed
```

```
private static void prediction() {
    for (int i = 0; i < nClasses; i++) {
        double probSubI = DIST[i] / sizeSample;
        PREDICTED[i] = probSubI * SIZESIMULATION;
    }
}

//Runs a chi^2-test to check the accuracy
//of our source of random numbers
//that must fit the distribution given by DIST
private static double chiTest() {
    double d = 0;
    for (int i = 0; i < nClasses; i++) {
        if (PREDICTED[i] > 0) {
            DISCREPANCY[i]
                    = Math.pow((COUNT[i] - PREDICTED[i]), 2)
                    / PREDICTED[i];
        }
        d = d + DISCREPANCY[i];
    }
    return d;
}

//The range of discrepancies is evaluated
//and is used to calculate the length of each
//interval to group Discrepancies.
public static double intervalLength() {
    min = 100000;
    max = 0;
    int lim = SIZESIMULATION;
    for (int i = 0; i < lim; i++) {
        if (EVENTS[i] < min) {
            min = EVENTS[i];
        }
        if (EVENTS[i] > max) {
            max = EVENTS[i];
        }
    }
    System.out.println("min = " + min);
    System.out.println("max = " + max);
    double range = max - min;
    double length = range / nClasses;
    return length;
}

//Prints the frequency table
```

```
    private static void printFreqTable(double FreqTable[][],
            int nClasses) {
        System.out.println("x and its absolute frequency");
        for (int i = 0; i <= nClasses; i++) {
            System.out.println(FreqTable[i][0]
                    + "     " + (int) (FreqTable[i][1]));
        }
    }

    //Calculates the sum    of absolute frequencies
    //of a frequency table
    private static void
            sum(double FreqTable[][]) {
        double sum = 0;
        for (int i = 0; i <= nClasses; i++) {
            sum = sum + FreqTable[i][1];
        }
        System.out.println("Number of events = " + sum);
    }

    //The border of each interval is calculated
    private static void getBorders(double Vect[]) {
        BARRIERS[0] = infLimit;
        CLASSMARKERS[0] = infLimit + intervalLength / 2;
        //Calculate maximal value of data
        double myMax = Vect[0];
        for (int j = 1; j < Vect.length; j++) {
            if (Vect[j] > myMax) {
                myMax = Vect[j];
            }
        }
        System.out.println("Max value = " + myMax);
        //Calculates number of classes
        nClasses = (int) ((myMax - infLimit) / intervalLength) + 1;
        System.out.println("Number of classes = " + nClasses);
        //Calculates interval borders and class markers
        for (int j = 1; j <= nClasses; j++) {
            BARRIERS[j] = BARRIERS[j - 1] + intervalLength;
            CLASSMARKERS[j] = CLASSMARKERS[j - 1] + intervalLength;
        }
        /*System.out.println( "Borders are" );
        printVector(BARRIERS);
        System.out.println( "Class markers are" );
        printVector(CLASSMARKERS);*/
    }
```

```
//Data are grouped into interval classes.
private static void group(double Vect[]) {
    intervalLength = intervalLength();
    infLimit = min - intervalLength / 2;
    getBorders(Vect);
    int nData = SIZESIMULATION;
    //Class markers a re defined
    for (int j = 0; j < nClasses; j++) {
        FREQTABLE[j][0] = CLASSMARKERS[j];
        FREQTABLE[j][1] = 0;
    }
    //Frequencies are calculated.
    for (int i = 0; i < nData; i++) {
        for (int j = 0; j < nClasses; j++) {
            if ((Vect[i] >= BARRIERS[j])
                    & (Vect[i] < BARRIERS[j + 1])) {
                FREQTABLE[j][1] = FREQTABLE[j][1] + 1;
            }
        }

    }
}

//The claim that there is a uniform distribution
//in within each inner barrier is tested
private static void test() {
    group(EVENTS);
    System.out.println("Frequency table of events");
    printFreqTable(FREQTABLE, nClasses);
    //Minor test for correct grouping
    sum(FREQTABLE);
    System.out.println("Frequencies   of events");
    for (int i = 0; i <= nClasses; i++) {
        System.out.println(FREQTABLE[i][1]);
    }
}

public static void main(String[] args) {
    for (int i = 0; i < DIST.length; i++) {
        COUNT[i] = 0;
    }
    for (int i = 0; i < SIZESIMULATION; i++) {
        //A random decimal number in (0,10) is generated
        //r.nextDouble() generates a random decimal number
        //with a uniform distribution in within 0 and 1.
        double e = 10 * R.nextDouble();
```

```
        if (PRINTALL)
            System.out.println(e);
        //Outputs are renamed
        if (e < 3) {
            e = e / 3;
            COUNT[0] = COUNT[0] + 1;
        }
        if ((3 <= e) & (e < 5)) {
            e = 1 + (e - 3) / 2;
            COUNT[1] = COUNT[1] + 1;
        }
        if ((5 <= e) & (e < 10)) {
            e = 3 + (e - 5) / 5;
            COUNT[3] = COUNT[3] + 1;
        }
        EVENTS[i] = e;
    }

    if (PRINTALL) {
        System.out.println(" Simulated events");
        printVector(EVENTS);
    }

    sizeSample = sizeSample();
    prediction();
    System.out.println("Observed and predicted values");
    printTwoVectors(COUNT, PREDICTED);
     double dExp = chiTest();
    System.out.println("Ho: observed values fit "
            + "predicted ones");
    System.out.println("Discrepancies");
    printVector(DISCREPANCY);
    System.out.println("Experimental discrepancy = " + dExp);
    System.out.println("Critical chi2 = " + CRITICALCHI2);
    if (dExp > CRITICALCHI2) {
        System.out.println("The event is extreme "
                + " according to Ho: invent another theory.");
    } else {
        System.out.println("Normal event according to Ho:"
                + "there is no reason to reject Ho");
    }
    //We open 10 classes for each interval
    nClasses = 10 * nClasses;
    //test for uniformity in within integer borders
    System.out.println(" \nTEST zone");
    test();
```

```
    }//End of main
}//End of Program E159 WhateverCTest
```

160, page 144. We can refurnish the generator with the following transformation

$w = 1.4(u - 0.5) + 0.5$

which generates numbers in within -0.2 and +1.2. You must reject those w that are less than zero or greater than 1.

Problems of Chapter 9

175, page 160.

a+b) Names were given to emphasize different styles of software development.

c) By looking to beginners in the trade, it gets clear that the degree of entanglement of software can be raised as much as desired while keeping full functionality. So, it is important to explain where the disentangled style of the genome comes from.

d+e) Every developer creates his or her own style: this is possible because the different ways as a problem can be solved grows (exponentially?) with the complexity of the problem (measures in, say, lines of code or in bits of the shorter programs that solves it).

f) Most possibly yes.

g) Fully structured style of software development is very expensive in additional code and additional structures that increase the complexity of the language.

h) The degree of entanglement or of disentanglement can be chosen in genetic programming. It is important to keep in mind that every extreme is possible. So, we have a task for a near future: to characterize the degree of disentanglement of the genome and to explain it. To begin with, the genomes of prokaryotes are not structured but have a disentangled style, while those of eukaryotes are very structured, highly disentangled and documented (a gen of alanine has many codons for alanine).

Problems of Chapter 10

191, page 176. The static declaration of methods and of variables was removed from everywhere.

192, page 176. The compiler cannot resolve the `Freqtable`. Suitable modifications follow:

```
//Program E192 Cell2
//We include a program or class, Chloroplast,
//in within another, Cell2.
```

```
//The ensemble is appropriately coordinated
//to perfectly function.

package ejvol5v2p;


public class Cell2 {


    //Variables in the outer class are declared
    private static final double[][] FREQTABLE
            = {{4, 3},
            {7, 6},
            {10, 10},
            {13, 15},
            {16, 4},
            {19, 2}};

    //Chloroplast is a class in within another one,
    //it is an inner class.
    //It is prototype or a type.
    //Their variables and methods cannot be static.
    private static class Chloroplast {

        //Variables in the inner class are declared
        //and initialized
        double FreqTable2[][] = {{20, 2},
        {32.5, 5},
        {37.5, 8},
        {42.5, 10},
        {47.5, 3},
        {60, 2}};

        //It is mandatory to initialize variables.
        //A non initialized variable is slack.
        //A process for initialization is called a CONSTRUCTOR.
        //We have here a void constructor because
        //the only variable, FreqTable2[][],
        //already has been initialized
        //as a constant
        Chloroplast() {

        }

        //*****Diverse methods follow
        //Prints the frequency table
```

```
    private void printFreqTable(double FreqTable[][]) {
        System.out.println("\n***************\n");
        System.out.println("x and its absolute frequency");
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            System.out.println(FreqTable[i][0]
                    + "\t " + FreqTable[i][1]);
        }
    }

    //Calculates the number of objects registered in
    //the table of absolute frequencies
    private double nFreqTable(double FreqTable[][]) {
        double n = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            n = n + FreqTable[i][1];
        }
        return n;
    }

    //Calculates the sum XF   of
    //a table of absolute frequencies
    private double
            sumXFFreqTable(double FreqTable[][]) {
        double sum = 0;
        int m = FreqTable.length;
        for (int i = 0; i < m; i++) {
            sum = sum + FreqTable[i][0] * FreqTable[i][1];
        }
        return sum;
    }

    //Calculates the mean   of
    //a table of absolute frequencies
    private double meanFreqTable(double FreqTable[][]) {
        double n = nFreqTable(FreqTable);
        double mean = sumXFFreqTable(FreqTable) / n;
        return mean;
    }

}//end of inner class

//***Outer class continues****
//Chloroplast does not exists apart from being a possibility.
//The prototype Chloroplast is instantiated by MYCHLOROPLAST.
```

```
    //MYCHLOROPLAST is a real object that exists in this virtual world.
    //MYCHLOROPLAST is born with its associated FreqTable2
    private static final Chloroplast MYCHLOROPLAST =  new Chloroplast();

    //The methods of the inner class are reused
    //by the outer class.
    private static void nsmFreqTable(double FreqTable[][]) {

        double n = MYCHLOROPLAST.nFreqTable(FreqTable);
        double sum = MYCHLOROPLAST.sumXFFreqTable(FreqTable);
        double mean = MYCHLOROPLAST.meanFreqTable(FreqTable);
        MYCHLOROPLAST.printFreqTable(FreqTable);
        System.out.println("Number of Objects = " + n);
        System.out.println("the pondered sum is = " + sum);
        System.out.println("The mean is = " + mean);
    }

    //Main of outer class
    //Must be public and static
    public static void main(String[] args) {
        //The method of the inner class  is used to
        //process objects of the outer class.
        nsmFreqTable(FREQTABLE);
        //The method of the inner class  is used to
        //process objects of the inner class s.
        nsmFreqTable(MYCHLOROPLAST.FreqTable2);
    }

}//end of Program E192 Cell2
```

193, page 177. The next is another example of the partial migration of the code of the inner class to the outer one.

```
//Program E193 Cell2b
//We include a program or class, Chloroplast,
//in within another, Cell2b.
//The ensemble is appropriately coordinated
//to perfectly function.
//Here we show another form of partial
//migration of the code of the inner program
//to the outer one.

package ejvol5v2p;
```

```
public class Cell2b {

    //private static final long serialVersionUID = 1L;
    //The outer application is instantiated
    static Cell2b a = new Cell2b();

    //Variables in the outer class are declared
    private static final double[][] FREQTABLE
            = {{4, 3},
            {7, 6},
            {10, 10},
            {13, 15},
            {16, 4},
            {19, 2}};

    //Chloroplast is a class in within another one,
    //it is an inner class.
    //It is prototype or a type.
    //Their variables and methods cannot be static.
    private static class Chloroplast {

        //Variables in the inner class are declared
        //and initialized
        double FREQTABLE2[][] = {{20, 2},
        {32.5, 5},
        {37.5, 8},
        {42.5, 10},
        {47.5, 3},
        {60, 2}};

        //It is mandatory to initialize variables.
        //A non initialized variable is slack.
        //A process for initialization is called a CONSTRUCTOR.
        //We have here a void constructor because
        //the only variable, FREQTABLE2[][],
        //already has been initialized
        //as a constant
        Chloroplast() {

        }

        //*****Diverse methods follow
        //Prints the frequency table
        private void printFreqTable(double FreqTable[][]) {
```

```
            System.out.println("\n***************\n");
            System.out.println("x and its absolute frequency");
            int m = FreqTable.length;
            for (int i = 0; i < m; i++) {
                System.out.println(FreqTable[i][0]
                        + "\t " + FreqTable[i][1]);
            }
        }

        //Calculates the number of objects registered in
        //the table of absolute frequencies
        private double nFreqTable(double FreqTable[][]) {
            double n = 0;
            int m = FreqTable.length;
            for (int i = 0; i < m; i++) {
                n = n + FreqTable[i][1];
            }
            return n;
        }

        //Calculates the sum XF   of
        //a table of absolute frequencies
        private double
                sumXFFreqTable(double FreqTable[][]) {
            double sum = 0;
            int m = FreqTable.length;
            for (int i = 0; i < m; i++) {
                sum = sum + FreqTable[i][0] * FreqTable[i][1];
            }
            return sum;
        }

    }//end of inner class


//***Outer class continues****
//Chloroplast does not exists apart from being a possibility.
//The prototype Chloroplast is instantiated by s.
//s is a real object that exists in this virtual world.
//s is born with its associated FreqTable2
    private static final Chloroplast MYCHLOROPLAST =  new Chloroplast();

    //Calculates the mean   of
    //a table of absolute frequencies
    private static double meanFreqTable(double FreqTable[][]) {
        double n = MYCHLOROPLAST.nFreqTable(FreqTable);
```

```
        double mean = MYCHLOROPLAST.sumXFFreqTable(FreqTable) / n;
        return mean;
    }

    //The methods of the inner class are reused
    //by the outer class.
    private static void nsmFreqTable(double FreqTable[][]) {
    //a method of the inner class s is used
        double n = MYCHLOROPLAST.nFreqTable(FreqTable);
        double sum = MYCHLOROPLAST.sumXFFreqTable(FreqTable);
    //A method of the outer class is used
        double mean = meanFreqTable(FreqTable);
        MYCHLOROPLAST.printFreqTable(FreqTable);
        System.out.println("Number of Objects = " + n);
        System.out.println("the pondered sum is = " + sum);
        System.out.println("The mean is = " + mean);
    }

    //Main of outer class
    //Must be public and static
    public static void main(String[] args) {
        //The method of the inner class  is used to
        //process objects of the outer class.
        nsmFreqTable(FREQTABLE);
        //The method of the inner class  is used to
        //process objects of the inner class s.
        nsmFreqTable(MYCHLOROPLAST.FREQTABLE2);
    }

}//end of main class and Program E193 Cell2b
```

197, page 181. A template for the choloroplast strategy:

```
//Program E197 Cell4
//The general scheme of the chloroplast strategy.

package ejvol5v2p;

//Outer class
public class Cell4 {

//An outer variable is declared and initialized
```

```
    private static final double T1[][]
            = {{4, 3},
            {7, 6},
            {10, 10},
            {13, 15},
            {16, 4},
            {19, 2}};

//Inner class
    private static class Chloroplast {

//Variable in the inner class is declared
//and initialized as a constant
        double FreqTable2[][] = {{20, 2},
        {32.5, 5},
        {37.5, 8},
        {42.5, 10},
        {47.5, 3},
        {60, 2}};

//Variable is declared
        double[][] FreqTable = new double[1000][2];

//Constructor 1 for the initialization of variables
        Chloroplast() {
            for (int j = 0; j < 2; j++) {
                for (int i = 0; i < 1000; i++) {
                    FreqTable[i][j] = 0;
                }
            }
        }

//Constructor 2 for the initialization of variables
        Chloroplast(double A[][], int r) {
            for (int j = 0; j < 2; j++) {
                for (int i = 0; i < r; i++) {
                    FreqTable[i][j] = A[i][j];
                }
            }
        }

//*****Methods of the inner class, not static
//Prints the frequency table with m rows and 2 columns
        private void printFreqTable(double FreqTable[][], int r) {
            System.out.println("\n***************\n");
            System.out.println("x and its absolute frequency");
```

```
            for (int i = 0; i < r; i++) {
                System.out.println(FreqTable[i][0]
                        + "\t " + FreqTable[i][1]);
            }
        }
    }//end of inner class

//***** Outer class continues *************
//Methods of the outer class are static
//Global instantiation
//An object with a  freqTable
//is instantiated by cloning from T1, in the outer class.
    private static final int J = 6;
    private static Chloroplast MYCHLOROPLAST = new Chloroplast(T1, J);

//The methods of the inner class are reused
//by the outer class.
    private static void nsmFreqTable(double FreqTable[][], int r) {
        MYCHLOROPLAST.printFreqTable(FreqTable, r);

    }

//Main of outer class
//Must be public and static
    public static void main(String[] args) {
        //Local instantiation
        //An object with
        //a  freqTable is birth by zeroed initialization
        Chloroplast t = new Chloroplast();
        //A zeroed FreqTable is processed
        int l = 10;
        nsmFreqTable(t.FreqTable, l);
        //A non zero FreqTable is processed
        l = MYCHLOROPLAST.FreqTable2.length;
        nsmFreqTable(MYCHLOROPLAST.FreqTable2, l);
    }

}//end of  Program E197 Cell4
```

201, page 184. Program without slack variables to calculate the variance, OOP.

```
//Program E201 VarList
//Calculates the variance of a list of data.
```

```
//OOP without slack variables.

package ejvol5v2p;

public class VarList {



    //Data
    private static final double L[] = {1, 2, 3, 4, 5};

    //******Inner class definition*******
    //This inner class defines a new type: dVector
    //It converts a 1d-array into a class or object.
    private static class dVector {

        int length;
        double F[] = new double[2000];

        //An instance of dVector can be
        //initialized in various ways:
        //first: Automatic zeroed initialization
        dVector(int l) {
            length = l;
            for (int i = 0; i < length; i++) {
                F[i] = 0;
            }
        }

        //Second: Initialization from  a 1d- array
        dVector(double[] A, int m) {
            length = m;
            /*
            for (int i = 0; i < length; i++) {
            F[i] = A[i];
             */
            System.arraycopy(A, 0, F, 0, length);
        }

        //Third: Initialization by cloning from a dVector
        dVector(dVector A) {
            length = A.length;
            /*
            for (int i = 0; i < A.length; i++) {
            F[i] = A.F[i];
             */
```

```
        System.arraycopy(A.F, 0, F, 0, A.length);
    }

    private void printVector(dVector Vect, String s) {
        System.out.println(s);
        for (int i = 0; i < Vect.length; i++) {
            System.out.println(Vect.F[i]);
        }
    }
}//end of inner class

//****Outer class continues**********
//Initialization of a dVector from an array
private static final dVector LIST1 =  new dVector(L, L.length);

//Initialization of a dVector from a dVector
private static dVector list2 =  new dVector(LIST1);

//Data are summed.
//The input is a dVector, the output is of type double.
private static double sum(dVector V) {
    double sum = 0;
    for (int i = 0; i < V.length; i++) {
        sum = sum + V.F[i];
    }
    return sum;
}

//The mean of a dVector is calculated
//The input is a dVector, the output is of type double.
private static double mean(dVector V) {
//The square of each entry is calculated
    double sum = sum(V);
    return sum / V.length;
}

//Deviations away from the mean are calculated
//for every entry of a dVector.
//The input is a dVector, the output is also  dVector.
private static dVector deviations(dVector V) {
//Zeroed initialization of a dVector
    dVector list =  new dVector(V.length);
//The square of each entry is calculated
    for (int i = 0; i < V.length; i++) {
        list.F[i] = V.F[i] - mean(V);
    }
```

```
        return list;
    }

    //Every entry of a dVector is squared.
    //The input is a dVector, the output is also  dVector.
    private static dVector squared(dVector V) {
        //Zeroed initialization of a dVector
        dVector list =  new dVector(V.length);
        //The square of each entry is calculated
        for (int i = 0; i < V.length; i++) {
            list.F[i] = V.F[i] * V.F[i];
        }
        return list;
    }

    //Main of outer class
    //Must be public and static
    public static void main(String[] args) {
        LIST1.printVector(LIST1, "Original data");
        double mean = mean(LIST1);
        System.out.println("Mean = " + mean);
        list2 = deviations(LIST1);
        list2 = squared(list2);
        System.out.println();
        list2.printVector(list2, "Squared deviations");
        double var = sum(list2) / (list2.length - 1);
        System.out.println("Variance = " + var);
    }
}//End of outer class and Program E201 VarList
```

Problems of Chapter 11

207, page 187. The code appears as part of the program, E209, page 189.

210, page 191. The trapezoidal rule is better: using the same partition, this rule gets more correct ciphers.

211, page 191. Answer: The next code can be ipso facto adapted to any-one density function. The code compares the outputs of the previous program with a suitable modification for reuse. The two methods produce results that differ in the 14th order and further. This is therefore the limit of accuracy of our approach. To remove this limit, one must use appropriate Java classes that allow for arbitrary precision. Let us observe that the reusable code is just a faith-fully implementation of the abstract mathematical formula for the trapezoidal

rule. We conclude that *abstract mathematics are reusable, i.e., evolvable, by construction and that a style of programming exists which transparently reflects that evolvability.* This explain in part the tremendous success of abstraction. The code follows.

```
//Program E211 TheZetaReuse
//Computes the  integral under the standard Gauss bell
//in within 0 and a positive z.
//Method: trapezoidal rule.
//The method is implemented in two forms:
//an ordinary one and
//other specially targeted for reuse, for evolution.

package ejvol5v2p;

public class TheZReuse {

    private static final double PI = 3.14159265358979323846;
    private static double z;

    //*****************Style: ordinary**************
    //Returns the integral under the z-density function
    //in within 0 and z. Precision=1/N.
    //Trapezoidal rule. Ordinary style.
    private static double zHalfBodyTrap(double z, long N) {
        /**
         * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciphers.
         */
        double h = z / N;
        double k = 1 / Math.pow(2 * PI, 0.5);
        double sum = 0;
        for (int j = 1; j < N; j++) {
            sum = sum + Math.exp(-h * h * j * j / 2);
        }
        sum = 2 * sum + 1 + Math.exp(-z * z / 2);
        sum = sum * k * h / 2;
        return sum;
    }

    //****************Style: reuse*************
    //Returns the density function of the standardized
    //normal distribution.
    private static double zFunction(double z) {

        return 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2);
```

```
}

//To use this method for another distribution,
//instead of  zFunction(z), write and make a call for your
//new distribution.
private static double f(double z) {
    double f = zFunction(z);
    return f;
}

//Returns the integral under the z-density function
//in within 0 and z. Precision=1/N.
//Trapezoidal rule adapted for reuse.
private static double zHalfBodyTrapReuse(double z, long N) {
    /**
     * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciph
     */

    double h = z / N;
    double sum = 0;
    for (int j = 1; j < N; j++) {
        sum = sum + f(h * j);
    }
    sum = 2 * sum + f(0) + f(z);
    sum = sum * h / 2;
    return sum;
}

//Here we verify that the code specially suited for reuse
//produce the very same results of the ordinary code.
private static void demo() {
    System.out.println("Area under the standardized bell "
            + "in within zero and 1.");
    System.out.println("N in 10^N, Trapezoidal rule, "
            + "  Trap  rule for reuse");
    int N;
    z = 1;
    double answer1, answer2;
    for (int i = 4; i < 7; i++) {
        N = (int) Math.pow(10, i);
        answer1 = zHalfBodyTrap(z, N);
        answer2 = zHalfBodyTrapReuse(z, N);
        System.out.println(i + "\t" + answer1 + "\t" + answer2);
    }
    System.out.println("The answer of Gnumeric  is     "
```

```
                + " 0.34134474606854 (for z= 1)");
    }

    public static void main(String[] args) {
        demo();
    }
}//End of Program E211 TheZetaReuse
```

212, page 191. p-value of the Z-distribution:

```
//Program E212 PValue
//Computes the  p-value for an event  z
//for the Z-distribution.
//Method: trapezoidal rule.
//The method is implemented in two forms:
//an ordinary one and
//other specially targeted for reuse, for evolution.

package ejvol5v2p;

public class PValue {

    private static final double PI = 3.14159265358979323846;
    private static double z;

    //****************Style: reuse*************
    //Returns the density function of the standardized
    //normal distribution.
    private static double zFunction(double z) {
        double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2);
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double z) {
        double f = zFunction(z);
        return f;
    }

    //Returns the integral under the z-density function
    //in within 0 and z. Precision=1/N.
    //Trapezoidal rule adapted for reuse.
```

```
    private static double zHalfBodyTrapReuse(double z, long N) {
        /**
         * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciph
         */

        double h = z / N;
        double sum = 0;
        for (int j = 1; j < N; j++) {
            sum = sum + f(h * j);
        }
        sum = 2 * sum + f(0) + f(z);
        sum = sum * h / 2;
        return sum;
    }

    //the p-value associated to a given z is calculated.
    private static void pvalue(double z) {
        System.out.println("p-value for z = " + z);
        System.out.println("N in 10^N, Trapezoidal rule, "
                + "one and two tails");
        int N;
        double answer, answer1, answer2;
        for (int i = 4; i < 7; i++) {
            N = (int) Math.pow(10, i);
            answer = zHalfBodyTrapReuse(z, N);
            answer1 = 1 - (0.5 + answer);
            answer2 = 1 - (2 * answer);
            System.out.println(i + "\t" + answer1 + "\t" + answer2);
        }
    }

    public static void main(String[] args) {
        z = 1.95995;
        pvalue(z);
    }
}//End of Program E212 PValue
```

Problems of Chapter 12

217, page 198. Simpson's rule

```
//Program E217 Simpson
//Computes the  p-value with one tail for a positive z
//for the Z distribution.
```

```
//Method 1: trapezoidal rule.
//Method 2: Simpson

package ejvol5v2p;

public class Simpson {

    private static double pi = 3.14159265358979323846;
    private static double z;

    //****************Style: reuse*************
    //Returns the density function of the standardized
    //normal distribution.
    private static double zFunction(double z) {
        double f = 1 / Math.pow(2 * pi, 0.5) * Math.exp(-z * z / 2);
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double z) {
        double f = zFunction(z);
        return f;
    }

    //Returns the integral under the z-density function
    //in within 0 and z. Precision=1/N.
    //Trapezoidal rule adapted for reuse.
    private static double zHalfBodyTrapReuse(double z, long N) {
        /**
         * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciphers.
         */

        double h = z / N;
        double sum = 0;
        for (int j = 1; j < N; j++) {
            sum = sum + f(h * j);
        }
        sum = 2 * sum + f(0) + f(z);
        sum = sum * h / 2;
        return sum;
    }

    //Returns the integral under the z-density function
    //in within 0 and z. Precision=1/N.
```

```
    //Simpson's rule adapted for reuse.
    private static double zHalfBodySimpson(double z, long N) {

        double h = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * h);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * h);
        }
        sum = sum + f((2 * m) * h);
        sum = (h / 3) * sum;
        return sum;
    }

    //the p-value associated to a given z is calculated.
    private static void pvalue(double z) {
        System.out.println("p-value for z = " + z);
        System.out.println("N in 10^N, Trapezoidal rule, "
                + "Simpson's rule, one   tail.");
        int N;
        double answer1, answer2, pValue1, pValue2;
        for (int i = 3; i < 5; i++) {
            N = (int) Math.pow(10, i);
            answer1 = zHalfBodyTrapReuse(z, N);
            pValue1 = 1 - (0.5 + answer1);
            answer2 = zHalfBodySimpson(z, N);
            pValue2 = 1 - (0.5 + answer2);
            System.out.println(i + "\t" + pValue1 + "\t" + pValue2);
        }
        System.out.println("The answer of Gnumeric  is     "
                + " 0.02275013194818 (for z= 2)");
    }

    public static void main(String[] args) {
        z = 2;
        pvalue(z);
    }
}//End of Program E217 Simpson
```

221, page 201.

We use the inverse transform $t = \frac{\sqrt{1+4x^2}-1}{2x}$ and reuse the code of the previous program to integrate in within 0 and resultant t.

```
//Program E221 FastSimpson
//Combines Simpson's rule
//with a change of scale to
//swiftly calculate the integral
//under the bell in within 0 and any z.

package ejvol5v2p;

public class FastSimpson {

    private static final double PI = 3.14159265358979323846;
    private static double z;

    //****************Style: reuse*************

    //Returns the transformed density function of the standardized
    //normal distribution.
    private static double zFunction(double t) {
        z = t / (1 - t * t);
        double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2)
                * (1 + t * t) / ((1 - t * t) * (1 - t * t));
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double z) {
        double f = zFunction(z);
        return f;
    }

    //Returns the integral under the z-density function
    //in within 0 and z. Precision=1/N.
    //Simpson's rule adapted for reuse.
    private static double zHalfBodySimpson(double z, long N) {

        double h = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * h);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * h);
```

```
        }
        sum = sum + f((2 * m) * h);
        sum = (h / 3) * sum;
        return sum;
    }

    //The area of the upper half of the bell is calculated
    //for various degrees of precision.
    private static void work(double z) {

        System.out.println("N in 10^N  Simpson's rule,"
                + " Area under the bell in within 0 and " + z);
        z = (Math.sqrt(1 + 4 * z * z) - 1) / (2 * z);
        int N;
        double answer;
        for (int i = 3; i < 5; i++) {
            N = (int) Math.pow(10, i);
            answer = zHalfBodySimpson(z, N);
            System.out.println(i + "\t" + answer);
        }
    }

    public static void main(String[] args) {
        double z = 50;
        work(z);
    }
}//End of Program E221 FastSimpson
```

Problems of Chapter 13

225, page 207. The code functions well for z in within zero and two and in this case M can be set to 20. Our Taylor's polynomial algorithm gets crazy under two circumstances: when the degree of the polynomial is raised above 70 and when $z > 2$.

251, page 237. The error seems to be less than 0.02. The error function takes value zero at zero, grows to a maximum value when $z = 0,5$ and then decreases close to zero.

252, page 237. The maximal error depends on the accuracy (from N) and stabilizes for high precision around 0.0175. The code follows:

```
//Program E252 HFamily2
```

```
//Computes the  integral under the standard Gauss bell
//in within 0 and a positive z.
//Method 1: trapezoidal rule.
//Method 2: algebraic approximation given by
//h(z) = 0.5 + (07z)/(2 fourthRoot((0.7z)^4 + 1));
//The maximal approximation error is reported.

package ejvol5v2p;

public class HFamily2 {

    private static final double PI = 3.14159265358979323846;
    //Values of the true cumulative function
    private static final double ZVECT[] = new double[10000001];

    // Mandatory initialization
    private static void initialize(long N) {
        for (int i = 0; i <= N; i++) {
            ZVECT[i] = 0.5;
        }
    }

    //*********Method 1: Trapezoidal rule*****************
    //Returns the density function of the standardized
    //normal distribution.
    private static double zFunction(double z) {
        double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2);
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double z) {
        double f = zFunction(z);
        return f;
    }

    //Returns 0.5 + the integral under the z-density function
    //in within 0 and z. Precision=1/N.
    //Trapezoidal rule adapted for reuse.
    private static void zHalfBodyTrapReuse(double zMax,
            long N) {
        /**
         * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciphers.
         */
```

```
        double Nr = N;
        double h = zMax / Nr;
        //System.out.println("h="+h);
        double sum = 0;
        for (int j = 0; j <= N; j++) {
            sum = sum + (h / 2) * (f(h * (j + 1)) + f(h * j));
            ZVECT[j] = sum + 0.5;
        }
    }

    //************** Method 2: algebraic fitting**********
    //Returns an ad hoc algebraic approximation
    //to the cumulative function of the
    //standardized normal distribution.
    private static double hFunction(double z) {
        double m = 4;
        double k = 0.7;
        double b = 1;
        double p = 1;
        p = p / m;
        double hz = 0.5 + (k * z)
                / (2 * Math.pow(Math.pow((k * z), m) + b, p));
        return hz;
    }

    //To use this method for another approximation,
    //instead of  hFunction(z), write and make a call for your
    //new proposal.
    private static double fitting(double z) {
        double f = hFunction(z);
        return f;
    }

    //**************Comparison of two methods**************
    private static void title(int N) {
        System.out.println("cum = 0.5 + Area under "
                + " the standardized bell in within zero and z.");
        System.out.println("That area is calculated by "
                + "the Trapezoidal rule");
        System.out.println("with " + N + " divisions "
                + "in within 0 and 10.");
        System.out.println("hz is our "
                + "algebraic approximation + 0.5.");
        System.out.println("Error = cum - hz.");
    }
```

```
//The maximal error of approximation is found
private static double findMaxError(double zMax, int N) {
    double maxError = 0;
    double error;
    double z;
    double Nr = N;
    double h = zMax / Nr;
    //System.out.println("h = " + h);
    for (int i = 0; i < N; i++) {
        z = i * h;
        error = ZVECT[i] - fitting(z);
        //System.out.print("z="+z + "  ZVECT = " +  ZVECT[i]);
        //System.out.print("hz="+ hz );
        error = Math.abs(error);
        //System.out.println("error="+ error );
        if (error > maxError) {
            maxError = error;
        }
    }
    return maxError;
}

private static void comparison() {
    //Calculations are done for z in (0, zMax)
    double zMax = 10;
    //N= Number of subdivisions of the interval (0,zMax).

    int N = 1000000;
    initialize(N);
    title(N);
    //Trapezoidal rule
    zHalfBodyTrapReuse(zMax, N);
    double hz;
    System.out.println("z from 0 to 1");
    for (int i = 0; i < 101; i++) {
        int j = i * N / 100;
        double z = i;
        z = z / 10;
        System.out.print("z=" + z + "  cum = " + ZVECT[j]);
        //Our algebraic fitting
        hz = fitting(z);
        System.out.print(" hz= " + hz);
        double error = ZVECT[j] - hz;
        System.out.println(" Error = " + error);
    }
    double error = findMaxError(zMax, N);
```

```
        System.out.print("maxerror  = " + error);
    }

    public static void main(String[] args) {
        comparison();
    }
}//End of Program E252 HFamily2
```

Problems of Chapter 13

260, page 253. Evolution with mutation and recombination:

```
//Program E260 Letterzymes3
//Clone of Program A140 Letterzymes3
//Evolution of a population
//of strings.
//Recombination is added to
//selection and mutation.

//Strings are at the same time
//genotype and phenotype.
//They are like ribozymes.


package ejvol5v2p;



import java.util.Random;

public class Letterzymes3
{

//Pre-declaration of global variables.
//They are used all throughout the whole class.

//Individuals are kept in the array
//INDIVIDUAL[]. It is an array of strings.
//The number of individuals must be
//less than LIMIT

private static final int LIMIT = 1000;
//How many generation to run
```

```
private static final int  NGEN = 200;
//Define here when a string is long enough
//to be purged
private static final int LARGESIZE = 1;
private static final String INDIVIDUAL[ ] = new String[100];
private static final int NINDIV = 15;
//Rank  keeps the rank of each individual.
private static final int RANK[] = new int[LIMIT];
static private String b;

static private int ReportMin[], ReportMax[];


//Turn on of the random generator
private static final Random RANDOM  = new Random();

//This method generates random chars
private  static  Character  randomChar()
{
    return  ( char ) ( RANDOM.nextInt( 26 ) + 65  );
}

/* We generate NINDIV individuals (strings)
 ten characters long.
 Sequences are completely random */
private static void Initialization( )
{
    System.out.println("ORIGINAL POPULATION");
    for(int i = 0; i< NINDIV; i++)
    {
        //Initialization  in blank.
        INDIVIDUAL[i]="";

        //An individual is assembled char by char
        for(int j = 0; j< 10; j++)
        {
            //We generate a random char
            Character c =  randomChar();
            //System.out.println( "char = " + c);
            //Char c is converted into a string s
            String s = ""+c;
            //s is concatenated to INDIVIDUAL[i]
            INDIVIDUAL[i] = INDIVIDUAL[i]+s;
        }
        System.out.println("Individual " +  i);
        System.out.println( INDIVIDUAL[i]);
```

```
    }



    for(int i = 0; i< NINDIV; i++)
        RANK[i]=0;

    ReportMin= new int[NGEN];
    ReportMax= new int[NGEN];
    for(int gen = 0; gen < NGEN; gen++)
    {
        ReportMax[gen]=0;
        ReportMin[gen]=0;
    }



}


//This method takes substring a
//and posits it at place Start inside c
private  static  String
      Insert(String a, int Start, String c)
{

    String  s1, s2;
    // We copy  the left part of c
    s1 = c.substring(0, Start);
    // We copy  the right part of c
    s2 = c.substring(Start);
    // We concatenate the left part of c
    //to a to the right part of c
    String z = s1 + a + s2;
    /*System.out.println("Original c " + c);
    System.out.println("Original a " + a);
    System.out.println("Start " + Start);
    System.out.println("PosInsertion c = " + z);
    */
    return z;
}

//The letters of "ALGAE" are inserted
//into the string of individual i
private static  void feeding(int i)
{
```

```
    String a;
    // We measure the length of INDIVIDUAL[i]
    int l = INDIVIDUAL[i].length();
    //Null strings cannot feed anyhow
    if (l == 0) ;
    else //Non null strings  feed on ALGAE
    {
        a = "A";
        int Start = RANDOM.nextInt(l);
        INDIVIDUAL[i] = Insert(a, Start, INDIVIDUAL[i]);
        a = "L";
        l = INDIVIDUAL[i].length();
        Start = RANDOM.nextInt(l);
        INDIVIDUAL[i] = Insert(a, Start, INDIVIDUAL[i]);
        a = "G";
        l = INDIVIDUAL[i].length();
        Start = RANDOM.nextInt(l);
        INDIVIDUAL[i] = Insert(a, Start, INDIVIDUAL[i]);
        a = "A";
        l = INDIVIDUAL[i].length();
        Start = RANDOM.nextInt(l);
        INDIVIDUAL[i] = Insert(a, Start, INDIVIDUAL[i]);
        a = "E";
        l = INDIVIDUAL[i].length();
        Start = RANDOM.nextInt(l);
        INDIVIDUAL[i] = Insert(a, Start, INDIVIDUAL[i]);
    }
}


//This Function replaces all occurrences of
//substring a by b inside c
private static String
Substitution(String a, String b, String c)
{
    String z = c.replace(a,b);
    /*System.out.println("original a " + a);
    System.out.println("Original b " + b);
    System.out.println("Original c " + c);
    System.out.println("replaced c " + z);
    */
    return z;
}


//This function or method deletes all occurrences
```

```
//of a from c.
static private String Delete(String a, String c)
{
    b = "";
    return Substitution(a, b, c);
}


//Here we see what a PARA-SITE  does to individuals.
private  static void parasitation(int i)
{
    int l  =  INDIVIDUAL[i].length();
    //The parasite deletes from the
    //beginning of the individual
    //any substring   matching "PARA".
    //We copy to Beginning the first
    //4 chars of INDIVIDUAL[i]
    if ( l > 4)
    {
        String Beginning =  INDIVIDUAL[i].substring(0,4);
        String s = "PARA";
        // We compare Beginning and s
        if (Beginning.equals(s))
            INDIVIDUAL[i] = INDIVIDUAL[i].substring(4);

        //The parasite deletes from the tail
        //of the individual
        //any substring   matching "SITE".
        String Ending =  INDIVIDUAL[i].substring(4);

        String t = "SITE";
        // We compare Ending and t
        if (Ending.equals(t))  INDIVIDUAL[i] =
                    INDIVIDUAL[i].substring( 0,l - 4);
    }
 }


//In this method individuals are mutilated
//as a measure to hinder overgrowth.
//We delete from the original string
//a substring of a random length beginning with
//a random Start position.
private static void Purge(int i)
{
    String c, s1, s2;
```

```
    int lc, Start, n;
    c = INDIVIDUAL[i];
    lc = c.length();
    if (lc > LARGESIZE)
    {
        Start = RANDOM.nextInt(lc);
        //We copy  the first part of c
        s1 = c.substring(0, Start);
        // We throw a random number
        n = RANDOM.nextInt(lc - Start+1);
        //We copy  the right part of c
        // after Start + n places
        s2 = c.substring( Start+ n);
        //We concatenate the left part of c
        //to  the right part of c
        INDIVIDUAL[i] = s1 + s2;
    }
}


//Individuals are sorted by fitness.
//Fitness = length, as an indicator of complexity.
private static void Sorting()
{
    int   Champ;
    //Necessary default initialization
    int Fitness[] = new int[100];
    for(int i = 0; i< NINDIV; i++)
        Fitness[i] =0;
    //We define a fitness function equal to the length
    //of the string
    for(int i = 0; i< NINDIV; i++)
    {
        String s = INDIVIDUAL[i];
        Fitness[i] = s.length();
        //System.out.println(INDIVIDUAL[i] + Fitness[i]);
    }
    //We sort individuals by fitness
    for(int i = 0; i< NINDIV;i++)
    {
        Champ = 0;
        for(int j = 0; j< NINDIV;j++)
            if (Fitness[j] > Fitness[Champ])   Champ = j;

        //The array Rank keeps a record of fitness
        //by decreasing order.
```

```
        RANK[i] = Champ;
        //The champ leaves the game
        Fitness[Champ] = 0;

    }
    /*
    //Report in decreasing order
    System.out.println("\nSorting :");
    for(int i = 0; i< NINDIV;i++)
        System.out.println(INDIVIDUAL[RANK[i]]);
    */
}


//The top ten fill the world
    private  static void Copying()
    {
        int   k;
        for(int i = 0; i< 10; i++)
        {
            int  l = 10*(10-i);
            for(int j = 0; j< 10; j++)
            {
                k = l -j-1;
                INDIVIDUAL[k] = INDIVIDUAL[RANK[i]];
                /*System.out.println("Ind " + k
                        + " replaced by top" + i);*/

            }
        }

        //System.out.println(  "THE NEW POPULATION IS");

        // for(int i = 0; i <  NINDIV;i++)
        //System.out.println(  INDIVIDUAL[i]);
    }

private static void Reproduction()
{
    //Individuals are sorted by length
    Sorting();
    //for(int i = 0; i< 10;i++)
    //System.out.println(  Individual(Rank(i))
    //The top ten produce a copy that
    //substitutes the bottom ten.
    Copying();
```

```
}

// From individual i
// the fist n  chars are deleted
private static   void Deletion(int i)
{
    int l, n;
    l = INDIVIDUAL[i].length();
    n = 0;
    if (l > 0)  n = RANDOM.nextInt(l);
    INDIVIDUAL[i] = INDIVIDUAL[i].substring(n);
 }

//This method inverts a
static private   String Inversion(String a)
{
    int  la = a.length();
    //Output
    String  d  = "";

    //We copy the first character of a to the last place
    //and so on.
    for(int  i = 0; i < la; i++)
     {
        char charI = a.charAt(i);
        d = charI + d;
     }
    return d;
}

//Local inversion of a substring inside individual i.
private static void innerInversion(int i)
{
    String sLeft, sRight;
    int start;
    int n;
    String c = INDIVIDUAL[i];
    int lc = c.length();
    if (lc > 1)
    {
        start = RANDOM.nextInt(lc);
        //We copy  the first part of c
        sLeft = c.substring(0, start);
        // We throw a random number
        n = RANDOM.nextInt(lc - start+1);
        //Inner substring
```

```
        String a = c.substring(start, start + n);
        String d = Inversion(a);
        //We copy  the right part of c
        // after Start + n places
        sRight = c.substring( start+ n);
        //We concatenate the left part of c
        //to  the right part of c
        INDIVIDUAL[i] = sLeft + d + sRight;
    }
    /*System.out.println("Input  =   " + c + "  " + n);
    System.out.println("Output  =  " + INDIVIDUAL[i]);
    */
}

//We consider only two types of mutation:
//deletion of the beginning
//and inversion of an inner part.
private static void Mutation()
{
    //Which mutation affects which individual
    //is a matter of gambling.
    //We throw a 0 else a 1
    for(int i = 0; i< NINDIV; i++)
    {
        int Luck = RANDOM.nextInt(2);
        if ( Luck == 1) Deletion(i);
        else  innerInversion(i);
    }
}

private static String recombinant(int i)
{

    int n = INDIVIDUAL.length;
    //A partner for i is found
    int j = RANDOM.nextInt(n);

    //A place for recombination
    int lengthI =  INDIVIDUAL[i].length();
    int lengthJ =  INDIVIDUAL[j].length();
    int minLength = Math.min(lengthI,lengthJ);
    int place = RANDOM.nextInt(minLength);
    String recombinant = INDIVIDUAL[i].substring(0, place)
            + INDIVIDUAL[j].substring(place);
    /*
    System.out.println( "Inputs = " + INDIVIDUAL[i]
```

```
            + "    " + INDIVIDUAL[j]
    + " Recombinant = " + recombinant);
    */
    return recombinant;
}


//Two strings are taken as input
//and one recombinant is produced
private static void Recombination()
{


    //Array with recombinant individuals
    String Individual2[ ] = new String[100];
    for(int i = 0; i< NINDIV; i++)
    {
        Individual2[i] = recombinant(i);
    }
    //Recombinants become the official population
    /*
    for(int i = 0; i< NINDIV; i++)
    {
    INDIVIDUAL[i] = Individual2[i];
    }
     */
    System.arraycopy(Individual2, 0, INDIVIDUAL, 0, NINDIV);

}



//This is the heart of the process
private static void Dynamics()
{
    //All individuals feed
    for(int i = 0; i< NINDIV; i++)
    {
        //The string Algae is digested and inserted
        //into the individual[i] char by char
        /*System.out.println( "The hungry individual "
                    + i + " is " + INDIVIDUAL[i]); */
        feeding(i);
        /*System.out.println(  "The full individual "
             + i + " is " + Individual(i)); */
        //All individuals are tested by the parasite,
```

```
        //which corrodes any  large substring
        //similar to it located at any end.
        parasitation(i);
        /*System.out.println(  "Pos-parasitation individual "
             + i + " is " + Individual(i));*/
        //Very long chains are attacked recurrently,
        //the same as very large fishes in the sea.
        Purge(i);
        /*System.out.println(  "The purged  individual "
             + i + " is " + Individual(i)); */
    }
    //The top ten are allowed to  reproduce:
    //a child occupies the place of the shortest individual
    Reproduction();
    //The new population is subjected to mutation
    Mutation();
    Recombination();
}


private static void report(int gen )
{
    System.out.println("\nPopulation at gen = " + gen );
    for(int j = 0; j< NINDIV; j++)
    {
        System.out.println("Gen " + gen + " Ind "
                    + j +" " +  INDIVIDUAL[j]);
    }
}

//Min and Max lengths are detected
private static void Observation(int gen)
{
    int m, Min, Max;
    Max = 0;
    Min = 32000000;
    for(int j = 0; j< NINDIV; j++)
    {
        m =  INDIVIDUAL[j].length();
        /*System.out.println(  "The Individual  "
           + Individual(j) + " has length " + m);*/
        if (m > Max)  Max = m;
        if (m < Min)   Min = m;
    }

    /*System.out.println(  "Generation "  + Gen
```

```
    + " MinLength = " + Min + " MaxLength = " + Max);*/
    ReportMin[gen] = Min;
    ReportMax[gen] = Max;
}

public static void main(String[] args)
{
    Initialization( );

    for(int gen = 0; gen < NGEN; gen++)
    {
        //System.out.println( "Beginning3");
        Dynamics();
        report(gen);
        //The aim of observation is to record
        //the minimum and maximum lengths
        //of the individuals in each generation.
        Observation(gen);
    }
    //Report of the dynamics of min and max indicators.
    System.out.println( "GENERATION MIN AND MAX VALUES");
    for(int Gen = 0; Gen < NGEN; Gen++)
        System.out.println( Gen + "\t" + ReportMin[Gen]
        + "\t" + ReportMax[Gen]);

}

}//End of Program E260 Letterzymes3
```

Problems of Chapter 14

267, page 274. When the number of sample points, N, is set to 10000, the algorithm finds sometimes in less than 300 generations a pseudo-optimal fitting around the next values:

Error = 0.00870886784648739
m = 4
k = 0.703103473039358
b = 0.8321561029594701

For $N = 100000$, the next values were found: Gen = 50
Error = 0.008197813433183598
m = 4
k = 0.4107104045719
b = 0.09608897101868201

272, page 296. The best fitting, that was attained after 200 generations in one among many runs, was the following:

Error = 0.004788698731374996 m = 4 k = 3.66597538866524 b = 686.892300121571 a = 0.007976843891721054 e = 0.220220132076153 i = 1 l = 4.

277, page 317. No improvement over previous programs was visible in a horizon of 200 generations.

278, page 317. The waiting time is not an exponential function of its order of appearance. Velocity is certainly a decreasing function of time (in generations) but eternal stagnation follows after some few activity while the goal to be achieved is far in the distance somewhere amidst heaven and earth. The code follows:

```
//Program E278 WaitingT
//Reports the waiting time for a new improvement.
//This is the number of generations required
//to achieve an error less than the actual one.

package ejvol5v2p;

import java.util.Random;

public class WaitingT {

    private static double pi = 3.14159265358979323846;
    //Values of the true cumulative function
    private static double zVect[] = new double[10000001];
    //Calculations are done for z in (0, zMax)
    private static double zMax = 10;
    //Number of sample points
    private static int N = 100000;
    //Max power of ten of numbers
    private static int maxPower;

    //***************Genetic part***********
    // Individuals are kept in the array
    // Individual[]. It is an array of strings.
    //Each individual encodes three numbers
    //corresponding to the 7 parameters of the
    //jFamily.
    // The number of individuals must be
    // less than limit.
    static int limit = 50000;
```

```
static double Fitness[];
static String Individual[], Individualc[];
//Actual number of individuals
static int nIndiv;
//Number of chars per parameter
static int nChars = 15;
static int Order[];
//static String b;
static int generation;
static int nGen;
static int ReportMin[], ReportMax[];
//Mutation rate per site
static double mutationRate;
static double error, oldError, newError;
static double deltaError;
static boolean test;
//number of terms under the root
static int nTermsRoot;
//number of terms in the second fraction
static int nTermsSecond;
static boolean inducedInit;
static long waitingTime = 1;

// Mandatory initialization
private static void initialize(long N) {
    for (int i = 0; i <= N; i++) {
        zVect[i] = 0.5;
    }
}

//*********Method 1: Trapezoidal rule*****************
//Returns the density function of the standardized
//normal distribution.
private static double zFunction(double z) {
    double f = 1 / Math.pow(2 * pi, 0.5) * Math.exp(-z * z / 2);
    return f;
}

//To use this method for another distribution,
//instead of  zFunction(z), write and make a call for your
//new distribution.
private static double f(double z) {
    double f = zFunction(z);
    return f;
}
```

```
    private static void trapRule() {
        initialize(N);
        //Trapezoidal rule for the cumulative function
        zHalfBodyTrapReuse(zMax, N);
    }

    //Returns 0.5 + the integral under the z-density function
    //in within 0 and z. Precision=1/N.
    //Trapezoidal rule adapted for reuse.
    private static void zHalfBodyTrapReuse(double zMax,
            long N) {
        /**
         * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciph
         */
        double Nr = N;
        double h = zMax / Nr;
        //System.out.println("h="+h);
        double sum = 0;
        for (int j = 1; j <= N; j++) {
            sum = sum + (h / 2) * (f(h * (j + 1)) + f(h * j));
            zVect[j] = sum + 0.5;
        }
    }

    //************** Method 2: algebraic fitting**********
    //Returns an   algebraic approximation
    //to the cumulative function of the
    //standardized normal distribution.
    private static double hcFunction(double z, h hf) {
        double p1 = 1;
        double p = p1 / hf.m;
        double hc = 0.5;
        double firstTerm = 1;
        for (int j = 0; j < nTermsRoot; j++) {
            firstTerm = firstTerm + hf.a[j] / (Math.pow(z, 2 * (j + 1)));
        }
        if (firstTerm >= 0) {
            hc = hc + 1 / (2 * Math.pow(firstTerm, p));
        } else {
            hc = 100;
        }
        double secondTerm = 0;
        for (int j = 0; j < nTermsSecond; j++) {
            secondTerm = secondTerm + hf.nc[j] / (Math.pow(z, 2 * j + 1))
                    + hf.c[j] * (Math.pow(z, 2 * j + 1));
        }
```

```
        secondTerm = p1 / secondTerm;
        hc = hc + secondTerm;
        return hc;
    }

    //To use this method for another approximation,
    //instead of  hFunction(z), write and make a call for your
    //new proposal.
    private static double fitting(double z, h hf) {
        double f = hcFunction(z, hf);
        return f;
    }

    //**************Comparison of two methods**************
    private static void title() {
        System.out.println(" Gen \t Error \t \t \t WT \t velocity");
    }

    //The maximal error of approximation is found
    private static double findMaxError(double zMax, int N, h hf) {
        zHalfBodyTrapReuse(zMax, N);
        double maxError = 0;
        double myError;
        double z ;
        double Nr = N;
        double h = zMax / Nr;
        double hz ;
        //System.out.println("h = " + h);
        for (int i = 1; i < N; i++) {
            z = i * h;
            hz = fitting(z, hf);
            myError = zVect[i] - hz;
            myError = Math.abs(myError);
            if (myError > maxError) {
                maxError = myError;
            }
            /*System.out.print("i = " + i + " zVect = " + zVect[i]);
System.out.println( " hz= " + hz + " error = " + error);*/
        }
        return maxError;
    }

    //*********************************
    //Genetic algorithm to optimize the
    //parameters of the hFamily.
    //We improve program B78, volume II, that finds the root
```

```
    //of a fifth degree polynomial.
    //*********************************
//*****************************************
//****************h CLASS *****************
//*****************************************
//This auxiliary class unites  fundamental
//parameters into an object.
//This encoding eases reuse of old code
    private static class h {
        //root

        int m;
        //number of terms under the root
        double a[] = new double[50];
        //Coefficients of terms with positive power,
        //second term
        double c[] = new double[50];
        //Coefficients of terms with negative power
        //second term
        double nc[] = new double[50];

        //Mandatory initialization
        //Mandatory initialization
        h() {
            m = 4;
            for (int j = 0; j < nTermsRoot; j++) {
                a[j] = 0;
            }
            for (int j = 0; j < nTermsSecond; j++) {
                c[j] = 0;
                nc[j] = 0;
            }
        }

        h(int mm, int nn,
                int ii, double aa[], double cc[], double ncc[]) {
            m = mm;
            /*
            for (int j = 0; j < nTermsRoot; j++) {
            a[j] = aa[j];
            }
             */
            System.arraycopy(aa, 0, a, 0, nTermsRoot);
            for (int j = 0; j < nTermsSecond; j++) {
                c[j] = cc[j];
                nc[j] = ncc[j];
```

```
            }
        }

        //Mandatory initialization
        h(h hf) {
            m = hf.m;
            /*
            for (int j = 0; j < nTermsRoot; j++) {
            a[j] = hf.a[j];
            }
             */
            System.arraycopy(hf.a, 0, a, 0, nTermsRoot);
            for (int j = 0; j < nTermsSecond; j++) {
                c[j] = hf.c[j];
                nc[j] = hf.c[j];
            }
        }

        //Printing method.
        static void print(h hf) {
            System.out.print(" m  = " + hf.m + ";");
            System.out.print(" a[] : ");
            for (int j = 0; j < nTermsRoot; j++) {
                System.out.print("  j = " + j + "   " + hf.a[j]);
            }
            System.out.print("; nc[] : ");
            for (int j = 0; j < nTermsSecond; j++) {
                System.out.print("  j = " + j + "   " + hf.nc[j]);
            }
            System.out.print("; c[] : ");
            for (int j = 0; j < nTermsSecond; j++) {
                System.out.print("  j = " + j + "   " + hf.c[j]);
            }
        }

    }//end of class h

//  Turn on of the random generator
    static Random r = new Random();

    //This method generates random chars
    // that represent digits
    private static char randomDigit() {
        return (char) (r.nextInt(10) + 48);
    }
```

```
//Generate a positive odd integer less than 9
private static char generateOddM() {
    int m = r.nextInt(5);
    m = (2 * m + 49);
    char mChar = (char) m;
    if (test) {
        System.out.println("i = " + mChar);
    }
    return mChar;
}

//Generate positive even integer less than 9
private static char generateEvenM() {
    int m = r.nextInt(4) + 1;
    m = (2 * m + 48);
    char mChar = (char) m;
    return mChar;
}

//Returns a string of digital numbers, a basic string
private static String generateSubString(int nChars) {
    String w = "";
    //An individual is assembled char by char
    for (int j = 1; j <= nChars; j++) {
        // We generate a random numeric char
        char c = randomDigit();
        String s = "" + c;
        //System.out.println( "char = " + c);
        // Char c is added to w
        w = w + s;
        //System.out.println( "w = " + w);
    }
    return w;
}

//A power  is generated
//Range runs form 0 to nChars
private static String addPower(int n) {
    Integer i = r.nextInt(n);
    String s = i.toString();
    if (s.length() == 1) {
        s = '0' + s;
    }
    return s;
}
```

```
//A string encoding for a sign is generated
private static String generateSign() {
    char s = '+';
    int spk = r.nextInt(2);
    if (spk == 0) {
        s = '-';
    }
    String t = "" + s;
    return t;
}

//A string encoding for a number is generated.
//a sign, a substring, a power with its sign.
private static String generateString(int nChars) {
    String sign = "" + generateSign();
    String ms = generateSubString(nChars);
    String signOfPower = "" + generateSign();
    String pk = addPower(maxPower);
    //Powers run contrary wise

    String k = sign + ms + 'P' + signOfPower + pk;
    if (test) {
        System.out.println("Sign  = " + sign);
        System.out.println("Main string  = " + ms);
        System.out.println("SignPower  = " + signOfPower);
        System.out.println("Power  = " + pk);
    }
    return k;
}

//Declarations and default initializations
// of other arrays.
private static void otherInit() {
    Order = new int[limit + 1];
    ReportMin = new int[limit + 1];
    ReportMax = new int[limit + 1];
    Fitness = new double[limit + 1];
    for (int i = 0; i < nIndiv; i++) {
        Order[i] = 0;
        ReportMin[i] = 0;
        ReportMax[i] = 0;
        Fitness = new double[limit + 1];
    }
}

private static String generateIndividual() {
```

```
        //m is generated
        char m = generateEvenM();
        String ind = "" + m;
        if (test) {
            System.out.println("m = " + m);
        }
        //the a's are  generated
        if (test) {
            System.out.println("\r The a's are generated");
        }
        for (int j = 0; j < nTermsRoot; j++) {
            String a = generateString(nChars);
            ind = ind + a;
            if (test) {
                System.out.println("j = " + j + " a = " + a);
            }
        }
        //the nc's are  generated
        if (test) {
            System.out.println("\r The nc's are generated");
        }
        for (int j = 0; j < nTermsSecond; j++) {
            String nc = generateString(nChars);
            ind = ind + nc;
            if (test) {
                System.out.println("j = " + j + " nc = " + nc);
            }
        }
        //the c's are  generated
        if (test) {
            System.out.println("\r The c's are generated");
        }
        for (int j = 0; j < nTermsSecond; j++) {
            String c = generateString(nChars);
            ind = ind + c;
            if (test) {
                System.out.println("j = " + j + " c = " + c);
            }
        }
        return ind;
    }

    private static String forcedInd() {
        //m is generated
        char m = '4';
        String ind = "" + m;
```

```
        //Coefficients are set to 1
        String a = "+100000000000000P+01";
        for (int i = 0; i < nTermsRoot + 2 * nTermsSecond; i++) {
            ind = ind + a;
        }
        return ind;
    }

    /* We generate nIndiv individuals (strings)
encoding for three parameters.
All parameters are positive.
Sequences are  random */
    private static void Initialization() {
        //Formal declaration of our array.
        Individual = new String[limit];
        if (test) {
            System.out.println("ORIGINAL POPULATION");
        }

        for (int i = 0; i < nIndiv; i++) {
            if (test) {
                System.out.println(" \r individual  = " + i);
            }
            Individual[i] = "";

            //Some individuals are given by the developer
            if (inducedInit & (i < 10)) {
                Individual[i] = forcedInd();
            } else {
                Individual[i] = generateIndividual();
            }
            if (test) {
                System.out.println("All at initialization = "
                        + Individual[i]);

                System.out.println("length  = "
                        + Individual[i].length());
                System.out.println();
            }
        }

        otherInit();
    }

    //A string encoding for a number is generated.
    //a sign, a substring, a power with its sign.
```

```
    private static double recoverNumber(String s) {
        //sign is recovered
        char sign = s.charAt(0);
        //sign is deleted
        s = s.substring(1);

        //main string is recovered
        String ms = s.substring(0, nChars);
        //main string is deleted
        s = s.substring(nChars);
        //'P' is deleted
        s = s.substring(1);
        //Sign of power is recovered
        char signOfPower = s.charAt(0);
        //Sign is deleted
        s = s.substring(1);
        //Power is recovered
        char c1 = s.charAt(0);
        char c2 = s.charAt(1);
        String pk = "" + c1 + c2;
        int p1 = Character.getNumericValue(c1);
        int p2 = Character.getNumericValue(c2);
        int power = p1 * 10 + p2;
        if (signOfPower == '-') {
            power = -power;
        }

        String k = sign + ms + 'P' + signOfPower + pk;

        if (test) {
            System.out.println("as substring = " + k);
        }
        double number = 0;
        for (int j = 0; j < nChars; j++) {
            char s3 = ms.charAt(j);
            if (s3 != '0') {
                int l = Character.getNumericValue(s3);
                int q = j + 1;
                double number1 = (int) l * Math.pow(10, power - q);
                number = number + number1;
            }
        }
        if (sign == '-') {
            number = -number;
        }
        if (test) {
```

```
        System.out.println("as number = " + number);
    }

    if (test) {
        System.out.println("Sign  = " + sign);
        System.out.println("Main string  = " + ms);
        System.out.println("SignOfPower  = " + signOfPower);
        System.out.println("Power as string = " + pk);
        System.out.println("Power as number = " + power);
        System.out.println("Final string = " + k);
        System.out.println("Final number = " + number);
    }
    return number;
}

// This method transforms a large string in  numbers,
//which are all packed in hf
private static h decoder(String s) {
    h hf = new h();
    if (test) {
        System.out.println("All    at   decoder  =  " + s);
        System.out.println(" length of s  = " + s.length());
    }
    //m is recovered
    char ms = s.charAt(0);
    int m = Character.getNumericValue(ms);
    if (test) {
        System.out.println("m = " + m);
    }
    //m is deleted
    s = s.substring(1);

    //the a's are  recovered
    if (test) {
        System.out.println("\r The a's are recovered");
    }
    for (int j = 0; j < nTermsRoot; j++) {
        if (test) {
            System.out.println("j = " + j);
        }
        hf.a[j] = recoverNumber(s);
        if (test) {
            System.out.println(" a = " + hf.a[j]);
        }
        //a is deleted
        s = s.substring(nChars + 5);
```

```
        }

        //the nc's are  recovered
        if (test) {
            System.out.println("\r The nc's are recovered");
        }
        for (int j = 0; j < nTermsSecond; j++) {
            if (test) {
                System.out.println("j = " + j);
            }
            hf.nc[j] = recoverNumber(s);
            if (test) {
                System.out.println(" nc = " + hf.nc[j]);
            }
            //a is deleted
            s = s.substring(nChars + 5);
        }

        //the c's are  recovered
        if (test) {
            System.out.println("\r The c's are recovered");
        }
        for (int j = 0; j < nTermsSecond; j++) {
            if (test) {
                System.out.println("j = " + j);
            }
            hf.c[j] = recoverNumber(s);
            if (test) {
                System.out.println(" c = " + hf.c[j]);
            }
            //a is deleted
            s = s.substring(nChars + 5);
        }
        return hf;
    }

    //Individuals are sorted by fitting
    private static void Sorting(int gen) {
        int Champ;
        for (int i = 0; i < nIndiv; i++) {
            if (test) {
                System.out.println("ind = " + i);
            }
            // Individual[i] is a string,
            // it is a genotype that encodes for hf;
            // hf is a set of parameters: it is the phenotype
```

```
            h hf = decoder(Individual[i]);
            double myError = findMaxError(zMax, N, hf);
            if (test) {
                System.out.println("i = " + i
                        + " maxError  = " + myError);
            }
            if (Math.abs(myError) < 0.000000000001) {
                System.out.println(" Gen = " + gen
                        + " Solution found");
                System.out.println(" Error = " + myError);
                h.print(hf);
            }
            Fitness[i] = 1 / (1 + myError * myError);
            if (test) {
                System.out.println();
            }
        }
        //Sorting
        for (int i = 0; i < nIndiv; i++) {
            Champ = 0;
            for (int j = 0; j < nIndiv; j++) {
                if (Fitness[j] >= Fitness[Champ]) {
                    Champ = j;
                }
            }
            //The array Order classifies individuals by  fitness
            // by equal or decreasing order.
            Order[i] = Champ;
            Fitness[Champ] = 0;
            if (test) {
                System.out.println(i + "th ind. is  No "
                        + Champ);
            }
        }
    }

    //waiting time reported
    private static void assessTime2() {
        String c = Individual[Order[0]];
        h hf = new h(decoder(c));
        error = findMaxError(zMax, N, hf);
        //Prints generation
        System.out.print(generation + "\t");
        System.out.print(error + "\t");
        newError = error;
        if (newError < oldError) {
```

```
            System.out.println(waitingTime);
            waitingTime = 1;
            oldError = newError;
        } else {
            waitingTime = waitingTime + 1;
            System.out.println("+");
        }

    }

    //optimization velocity reported
    //deltaError/waitingTime
    private static void assessTime() {
        String c = Individual[Order[0]];
        h hf = new h(decoder(c));
        error = findMaxError(zMax, N, hf);
        newError = error;
        if (newError < oldError) {
            double progress = oldError - newError;
            System.out.print(generation + "\t ");
            System.out.print(error + "\t");
            System.out.print(waitingTime + "\t");
            oldError = newError;
            double velocity = progress / waitingTime;
            System.out.println(velocity);
            waitingTime = 1;
        } else {
            waitingTime = waitingTime + 1;
        }
    }

    //Each individual of the top ten
    // produces 10  copies.
    private static void Reproduction(int gen) {
        if (test) {
            System.out.println("Reproduction");
        }
        if (test) {
            System.out.println("The best = " + Order[0]);
        }
        Individualc = new String[limit];
        int counter = 0;
        for (int top = 0; top < 10; top++) {
            for (int j = 0; j < 10; j++) {
                Individualc[counter] = Individual[Order[top]];
                counter = counter + 1;
```

```
        }
    }
    /*
    for (int j = 0; j < counter; j++) {
    Individual[j] = Individualc[j];
    }
     */
    System.arraycopy(Individualc, 0, Individual, 0, counter);
}

private static String mutate(int k, int place) {
    String s = Individual[k];
    //place = 18;
    if (test) {
        System.out.println(" indj         = " + s);
        System.out.println("placeMut = " + place);
    }
    //We go over the same steps of decoding
    //case m
    if (place == 0) {
        char c = generateEvenM();
        s = c + s.substring(1);
        if (test) {
            System.out.println(" m mutated    = " + s);
        }
    }

    //The a's, nc's and c's all have the same encoding
    //Case sign of number
    if (place % (nChars + 5) == 1) {
        String sign = generateSign();
        s = s.substring(0, place)
                + sign + s.substring(place + 1);
        if (test) {
            System.out.println(" SSSSSSSSSSSSSSSign = "
                    + s.charAt(place));
            System.out.println(" sign mutated    = " + s);
        }
    }

    //Case main string
    if (1 < (place % (nChars + 5))
            & (place % (nChars + 5) < nChars + 1)) {
        char c = randomDigit();
        s = s.substring(0, place)
                + c + s.substring(place + 1);
```

```
            if (test) {
                System.out.println("Basic mutated = " + s);
            }
        }

        //'P' is not a case for mutation
        //case sign of power
        if (place % (nChars + 5) == nChars + 3) {
            String sign = generateSign();
            s = s.substring(0, place)
                    + sign + s.substring(place + 1);
            if (test) {
                System.out.println(" Sign of power = "
                        + s.charAt(place));
                //System.out.println (" sign mutated    = " + s);
            }
        }
        //Case power:
        if (place % (nChars + 5) == nChars + 4) {
            String power = addPower(maxPower);
            s = s.substring(0, place)
                    + power + s.substring(place + 2);
            if (test) {
                System.out.println("Power mutated = " + s);
            }
        }
        if (place % (nChars + 5) == nChars + 5) {
            String power = addPower(maxPower);
            s = s.substring(0, place - 1)
                    + power + s.substring(place + 1);
            if (test) {
                System.out.println("Power mutated = " + s);
            }

        }
        return s;
    }

    //Individual j undergoes site by site mutation
    private static void mutateIndj(int j) {
        String s = Individual[j];

        int length = s.length();
        for (int place = 0; place < length; place++) {
            double q = r.nextDouble();
            if (q < mutationRate) {
```

```
            Individual[j] = mutate(j, place);
        }
    }
}

//We consider only one type of mutation:
//substitution of one char by another one.
//The first copy of the winner does not mutate.
private static void Mutation() {
    for (int j = 1; j < nIndiv; j++) {
        mutateIndj(j);
    }
    //Some individuals are generated ab initio.
    //We hope that they will help to a rapid escape
    //from local optims.
    int initial = 15;
    if (initial > 0) {
        for (int i = initial; i < nIndiv; i++) {
            if (test) {
                System.out.println("i = " + i);
            }
            Individual[i] = "";
            Individual[i] = generateIndividual();
        }
    }
}

//Two strings recombine and produce two offspring.
private static void Recombination() {
    int l = Individual[0].length();
    for (int j = 10; j < nIndiv; j++) {
        int m;
        int n;
        m = r.nextInt(nIndiv);
        n = r.nextInt(nIndiv);
        String a = Individual[m];
        String b = Individual[n];
        //Define place of recombination
        int placeRec = r.nextInt(l);
        //the zeroth individual is kept intact.
        if (m != 0) {
            Individual[m] = a.substring(0, placeRec)
                    + b.substring(placeRec);
        }
        if (n != 0) {
            Individual[n] = b.substring(0, placeRec)
```

```
                        + a.substring(placeRec);
            }
        }
    }

    //Mutation rate can increase
    private static void mutationRedef() {
        String c = Individual[0];
        h hf = decoder(c);
        double myError = findMaxError(zMax, N, hf);
        /* System.out.println("gen " + gen +
  " Error " + error);*/
 /* System.out.println("mutation rate = " +
    mutationRate);*/
        newError = myError;
        deltaError = Math.abs(newError - oldError);
        /* System.out.println("Error= " + error +
            " deltaError= " + deltaError); */
        oldError = newError;
        if (deltaError < 0.001) {
            mutationRate = 1.1 * mutationRate;
            if (mutationRate > 1) {
                mutationRate = 1;
            }

            /*  System.out.println("gen " + gen + "
    mutRedef");*/
        } else;
    }

    //Overall method
    private static void dynamics(int gen) {
        //Individuals are sorted by fitness
        Sorting(gen);
        //for(int i = 1; i<= 10;i++)
        //System.out.println(  Individual(Order(i))
        assessTime();
        //The top ten are preferentially reproduced
        Reproduction(gen);
        //The new population is subjected to mutation
        Mutation();
        Recombination();
        //every 100 generations, stagnation is revised
        if (gen % 100 == 0) {
            mutationRedef();
        }
```

```
}

//Decodes a string into a number
private static void decode() {
    System.out.println();
    System.out.println("Numeric values at decoding");
    for (int i = 0; i < nIndiv; i++) {
        System.out.println("i = " + i);
        decoder(Individual[i]);
    }
}

//test of encoding and decoding
private static void test() {
    nIndiv = 1;
    System.out.println("Strings at Initialization");
    Initialization();
    decode();
}

public static void main(String[] args) {
    nTermsRoot = 2;
    nTermsSecond = 2;
    inducedInit = true;
    nIndiv = 25;
    maxPower = 6;
    Initialization();
    System.out.println("nTermsRoot =  " + nTermsRoot);
    System.out.println("nTermsRoot =  " + nTermsSecond);
    nGen = 100000;
    oldError = 1000;
    //Mutation rate per site
    mutationRate = 0.3;
    test = false; //else true
    if (test) {
        test();
    } else {
        //test = true;
        //The whole cumulative function is estimated
        //by the trapezoidal rule
        trapRule();
        //The parameters in the hFamily are optimized
        //by a genetic algorithm.
        System.out.println("Running ");
        //Random else induced initialization
        title();
```

```
            for (int gen = 1; gen <= nGen; gen++) {
                generation = gen;
                dynamics(gen);
            }
        }
    }
}//End of Program E278 WaitingT
```

279, page 317. Replace the method *hcFunction(double z, h hf)* by the following one:

```
 //Returns an   algebraic approximation
  //to the cumulative function of the
  //standardized normal distribution.
  private static double hcFunction(double z, h hf)
  {
double p1 = 1;
double   p = p1/hf.m;
double  hc = 0.5;
double secondTerm = 0;
for(int j = 0; j < nTermsSecond; j++ )
   secondTerm = secondTerm + hf.nc[j]/(Math.pow(z,2*j+1))
                           +hf.c[j]*(Math.pow(z,2*j+1));
secondTerm = p1/secondTerm;
double firstTerm = 1;
for(int j = 0; j < nTermsRoot; j++ )
 {
   firstTerm = firstTerm + hf.a[j]/(Math.pow(z,2*(j+1)));
 }
if (firstTerm >= 0)
    hc = hc + (1+ secondTerm)/(2* Math.pow(firstTerm,p));
else hc = 100;

hc  = hc + secondTerm;
return hc;
  }
```

No improvement was detected by the Author.

280, page 317. Replace the method *hcFunction(double z, h hf)* by the next one. This architecture produced in one run and after 250 generations an error equal to 0.0026415875996782523. And this was all to it.

```
  //Returns an   algebraic approximation
  //to the cumulative function of the
  //standardized normal distribution.
  private static double hcFunction(double z, h hf)
  {
double p1 = 1;
double   p = p1/hf.m;
double  hc = 0.5;
double secondTerm = 0;
for(int j = 0; j < nTermsSecond; j++ )
   secondTerm = secondTerm + hf.nc[j]/(Math.pow(z,2*j+1))
                           +hf.c[j]*(Math.pow(z,2*j+1));
secondTerm = p1/secondTerm;
double firstTerm = 1;
for(int j = 0; j < nTermsRoot; j++ )
 {
   firstTerm = firstTerm + hf.a[j]/(Math.pow(z,2*(j+1)));
 }
if (firstTerm >= 0)
    hc = hc + (1)/(2* Math.pow(firstTerm,p)+ secondTerm);
else hc = 100;

hc  = hc + secondTerm;
return hc;
  }
```

281, page 317. Replace the method *hcFunction(double z, h hf)* by the following one. No differential performance was detected by the Author.

```
 //Returns an   algebraic approximation
  //to the cumulative function of the
  //standardized normal distribution.
  private static double hcFunction(double z, h hf)
  {
double p1 = 1;
double   p = p1/hf.m;
double  hc = 0.5;
double secondTerm = 0;
for(int j = 0; j < nTermsSecond; j++ )
   secondTerm = secondTerm + hf.nc[j]/(Math.pow(z,2*j+1))
                           +hf.c[j]*(Math.pow(z,2*j+1));
secondTerm = p1/secondTerm;
double firstTerm = 1;
for(int j = 0; j < nTermsRoot; j++ )
```

```
 {
   firstTerm = firstTerm + hf.a[j]/(Math.pow(z,2*(j+1)));
 }
if (firstTerm >= 0)
    hc = hc + (1)/(2* Math.pow(firstTerm+ secondTerm,p));
else hc = 100;

hc  = hc + secondTerm;
return hc;
  }
```

282, page 318. The best approximation of the cumulate normal distribution produced by the following program after 1000 generations was

$$F(z) \approx \frac{1}{1+e^{-(1.59718000723822*z+0.0700000001648929*z^3+1.2443249864931598E-12*z^5)}}$$

The maximal error in within z = 0 and z = 3 was 2.2173122641955256E-4, so this approximation matches 3 decimal figures. The code follows:

```
//Program E282 LogisticApproximation
/*
This  program studies an approximation to the
cumulative normal distribution
proposed by  Bowling et al.  :
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.429.6900\&rep=rep1\

 F(z) = 1/  (1 + e^-(1.5976*z + 0.07056*z^3 ))

This approximation is promised to match 3 decimal figures.
We use evolution to research what
the type  double  and  a fifth degree polynomial give of
themselves:

F(z) = 1 /  (1 + e^-(a*z + b*z^3 + c*z^5))


*/


package ejvol5v3p;

import java.util.Random;

public class LogisticApproximation {
```

```
private static double pi = 3.14159265358979323846;
//Values of the true cumulative function
private static double zVect[] = new double[10000001];
//Calculations are done for z in (0, zMax)
private static double zMax = 3;
//Number of sample points
private static int N = 10000;



//****************Genetic part***********
// Individuals are kept in the array
// Individual[]. It is an array of strings.
//Each individual encodes three numbers
//corresponding to the 3 parameters of the
//hFamily.
// The number of individuals must be
// less than limit.
static int limit = 50000;
static double Fitness[];
static String Individual[], Individualc[];
//Actual number of individuals
static int nIndiv;
//Number of chars per parameter
static int nChars = 15;
static int Order[];

static int generation;
static int nGen;
static int ReportMin[], ReportMax[];
static double mutationRate;
static double oldError, newError;
static double deltaError;
static boolean test, testEncoding;

// Mandatory initialization
private static void initialize(long N) {
    for (int i = 0; i <= N; i++) {
        zVect[i] = 0.5;
    }
}

//*********Method 1: Trapezoidal rule*****************
//Returns the density function of the standardized
//normal distribution.
private static double zFunction(double z) {
```

```
        double f = 1 / Math.pow(2 * pi, 0.5) * Math.exp(-z * z / 2);
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double z) {
        double f = zFunction(z);
        return f;
    }

    private static void trapRule() {
        initialize(N);
        title(N);
        //Trapezoidal rule for the cumulative function
        zHalfBodyTrapReuse(zMax, N);
    }

    //Returns 0.5 + the integral under the z-density function
    //in within 0 and z. Precision=1/N.
    //Trapezoidal rule adapted for reuse.
    private static void zHalfBodyTrapReuse(double zMax,
            long N) {
        /**
         * With N = 10^5, and for z=1 the algorithm achieves 11 correct ciph
         */
        double Nr = N;
        double h = zMax / Nr;
        //System.out.println("h="+h);
        double sum = 0;
        for (int j = 1; j <= N; j++) {
            sum = sum + (h / 2) * (f(h * (j + 1)) + f(h * j));
            zVect[j] = sum + 0.5;
        }
    }

    //************** Method 2: algebraic fitting**********

    //Returns an ad hoc algebraic approximation
    //to the cumulative function of the
    //standardized normal distribution.
    private static double hFunction(double a, double b,
            double c, double z) {
```

```
    double zP3 = z*z*z;
    double zP5 = z*z*z*z*z;
    //hZ = 1/  (1 + e^-(a*z  + b*z^3 + c* z^5));
    double hz =   1.0/  (1 + Math.exp( -(a*z  + b*zP3 + c* zP5)));
    return hz;
}

//To use this method for another approximation,
//instead of  hFunction(z), write and make a call for your
//new proposal.
private static double fitting(double z, h hf) {
    double a = hf.a;
    double b = hf.b;
    double c = hf.c;
    double f = hFunction(a,b,c,z);
    return f;
}

//**************Comparison of two methods**************
private static void title(long N) {
    System.out.println("cum = 0.5 + Area under "
            + " the standardized bell in within zero and z.");
    System.out.println("That area is calculated by "
            + "the Trapezoidal rule");
    System.out.println("with " + N + " divisions "
            + "in within 0 and 10.");
    System.out.println("A genetic algorithm calculates ");
    System.out.println("the optimal parameters of the h family");
    System.out.println("to best fit cum.");
}

//The maximal error of approximation is found
private static double findMaxError(double zMax, int N, h hf) {
    double maxError = 0;
    double error;
    double z;
    double Nr = N;
    double h = zMax / Nr;
    double hz;
    //System.out.println("h = " + h);
    for (int i = 0; i < N; i++) {
        z = i * h;
        hz = fitting(z, hf);
        error = zVect[i] - hz;
        error = Math.abs(error);
        /*System.out.print("z="+z + "  cum = " +  zVect[i]);
```

```
  System.out.print(" hz= "  +  hz);
  System.out.println(" Error = " + error);*/
            if (error > maxError) {
                maxError = error;
            }
        }
        return maxError;
    }

    //*********************************
    //Genetic algorithm to optimize the
    //parameters a,b,c, of the chosen family.
    //We improve program B78, volume II, that finds the root
    //of a fifth degree polynomial.
    //*********************************
//*****************************************
//****************h CLASS ******************
//*****************************************
//This auxiliary class unites  fundamental
//parameters into an object.
//This encoding eases reuse of old code
    private static class h {

        double a;
        double b;
        double c;

        //Mandatory initialization
        h(double l, double m, double n) {
            a = l;
            b = m;
            c = n;
        }

        //Mandatory initialization
        h(h hf) {
            a = hf.a;
            b = hf.b;
            c = hf.c;
        }

        //Printing method.
        static void print(h hf) {
            System.out.print(" a  = " + hf.a);
            System.out.print(" b  = " + hf.b);
            System.out.print(" c  = " + hf.c);
```

```
        }

    }//end of class h

//  Turn on of the random generator
    static Random r = new Random();

    //This method generates random chars
    // that represent digits
    private static char randomDigit() {
        return (char) (r.nextInt(10) + 48);
    }

    //m is positive even integer less than 9
    private static char generateM() {
        int m = r.nextInt(4) + 1;
        m = (2 * m + 48);
        char mChar = (char) m;
        return mChar;
    }

    //Returns a string of digital numbers, a basic string
    private static String generateSubString(int nChars) {
        String w = "";
        //An individual is assembled char by char
        for (int j = 1; j <= nChars; j++) {
            // We generate a random numeric char
            char c = randomDigit();
            String s = "" + c;
            //System.out.println( "char = " + c);
            // Char c is added to w
            w = w + s;
            //System.out.println( "w = " + w);
        }
        return w;
    }

    //A power  is generated
    //Range runs form 0 to nChars
    private static String addPower(int nChars) {
        Integer i = r.nextInt(nChars);
        String s = i.toString();
        if (s.length() == 1) {
            s = '0' + s;
        }
        return s;
```

```
}

//A string encoding for a sign is generated
private static String generateSign() {
    int spk = r.nextInt(2);
    char s = '+';
    if (spk == 0) {
        s = '-';
    }
    String t = "" + s;
    return t;
}

//A string encoding for a number is generated
private static String generateString(int nChars) {
    String signOfNumber = "" + generateSign();
    String k = generateSubString(nChars);
    if (test) {
        System.out.println("  String = " + k);
    }
    String pk = addPower(nChars);

    String s = "" + generateSign();

    k = k + 'P' + s + pk;
    if (test) {
        System.out.println("Power  = " + pk);
        System.out.println("SignPower  = " + s);
        System.out.println("As String   = " + k);
    }
    return k;
}

//Declarations and default initializations
// of other arrays.
private static void otherInit() {
    Order = new int[limit + 1];
    ReportMin = new int[limit + 1];
    ReportMax = new int[limit + 1];
    Fitness = new double[limit + 1];
    for (int i = 0; i < nIndiv; i++) {
        Order[i] = 0;
        ReportMin[i] = 0;
        ReportMax[i] = 0;
        Fitness = new double[limit + 1];
    }
```

```
    }

    private static String generateIndividual() {
        String aa = generateString(nChars);
        String bb = generateString(nChars);
        String cc = generateString(nChars);
        //c is generated
        if (test) {
            System.out.println("a = " + aa
                + " b = " + bb   + " c = " + cc);
        }
        String ind = "" + aa + bb + cc;
        return ind;
    }

    /* We generate nIndiv individuals (strings)
encoding for three parameters.
All parameters are positive.
Sequences are  random */
    private static void Initialization() {
        //Formal declaration of our array.
        Individual = new String[limit];
        if (test) {
            System.out.println("ORIGINAL POPULATION");
        }
        for (int i = 0; i < nIndiv; i++) {
            if (test) {
                System.out.println("i = " + i);
            }
            Individual[i] = "";
            Individual[i] = generateIndividual();


                System.out.println("Chro " + i + " " + Individual[i]);
                System.out.println();

            if (test) {

                System.out.println("Chro " + i + "   " + Individual[i]);
                System.out.println();
            }
        }
        otherInit();
    }

    //String s is decoded into a number
```

```
private static double recoverNumber(String s) {
    // We get the sign of power of k
    // which could be + else -.
    char sp = s.charAt(nChars + 1);
    if (test) {
        System.out.println("Sign of power = " + sp);
    }
    // We get the   power of k
    // which ranges from 00 to 99.
    char s1 = s.charAt(nChars + 2);
    int p1 = Character.getNumericValue(s1);
    s1 = s.charAt(nChars + 3);
    int p2 = Character.getNumericValue(s1);
    int power = p1 * 10 + p2;
    if (sp == '-') {
        power = -power;
    }
    if (test) {
        System.out.println("Power = " + power);
    }

    //*********** We get k
    String k = s.substring(0, nChars);
    if (test) {
        System.out.println("as substring = " + k);
    }
    double number = 0;
    for (int j = 0; j < nChars; j++) {
        char s3 = k.charAt(j);
        if (s3 != '0') {
            int l = Character.getNumericValue(s3);
            int q = j + 1;
            double number1 = (int) l * Math.pow(10, power - q);
            number = number + number1;
        }
    }
    if (test) {
        System.out.println("as number = " + number);
    }
    return number;
}

// This method decodes string s into three numbers, a,b,c
private static h decoder(String s) {
    if (test) {
        System.out.println(" Individual =   = " + s);
```

```
            System.out.println("length of s  = " + s.length());
        }

        /*System.out.println( "place nChars + 2 = "
                + s.charAt(nChars + 2));
System.out.println( "place nChars + 6 = "
                + s.charAt(2*nChars + 6));
         */

        //We get the string that encodes for a
        String aString = s.substring(0, nChars + 4);
        if (test) {
            System.out.println("aString = " + aString);
        }
        if (test) {
            System.out.println("Recovering a");
        }
        double aa = recoverNumber(aString);
        if (test) {
            System.out.println("a as number = " + aa);
        }

        //We get the string that encodes for b
        if (test) {
            System.out.println("Recovering b");
        }
        String bString = s.substring(nChars + 4, 2*nChars + 8);
        if (test) {
            System.out.println("bString = " + bString);
        }
        double bb = recoverNumber(bString);
        if (test) {
            System.out.println("b as number = " + bb);
        }

        //We get the string that encodes for b
        if (test) {
            System.out.println("Recovering c");
        }
        String cString = s.substring(2*nChars + 8);
        if (test) {
            System.out.println("cString = " + cString);
        }
        double cc = recoverNumber(cString);
        if (test) {
            System.out.println("c as number = " + cc);
```

```
        }
        /*
        System.out.println(" Individual =   = " + s);
        System.out.println("aString = " + aString);
        System.out.println("a as number = " + aa);
        System.out.println("bString = " + bString);
        System.out.println("b as number = " + bb);
        System.out.println("cString = " + cString);
        System.out.println("c as number = " + cc);
        */
        h hf = new h(aa, bb, cc);
        return hf;
    }

    private static void report() {
        h hf = decoder(Individual[Order[0]]);
        double error = findMaxError(zMax, N, hf);
        System.out.println("gen = " + generation);
        System.out.print("  Error = " + error);
        h.print(hf);
        System.out.println();
    }

    //Individuals are sorted by fitting
    private static void Sorting(int gen) {
        int Champ;
        for (int i = 0; i < nIndiv; i++) {
            if (test) {
                System.out.println("i = " + i);
            }
            // Individual[i] is a string,
            // it is a genotype that encodes for hf;
            // hf is a set of parameters: it is the phenotype
            h hf = decoder(Individual[i]);
            double error = findMaxError(zMax, N, hf);
            if (test) {
                System.out.println("i = "
                        + i + " maxError  = " + error);
            }
            if (Math.abs(error) < 0.000000000001) {
                System.out.println("Gen = " + gen
                        + " Solution found");
                System.out.println("Error = " + error);
            }
            Fitness[i] = 1 / (1 + error * error);
            if (test) {
```

```
            System.out.println();
        }
    }
    //Sorting
    for (int i = 0; i < nIndiv; i++) {
        Champ = 0;
        for (int j = 0; j < nIndiv; j++) {
            if (Fitness[j] >= Fitness[Champ]) {
                Champ = j;
            }
        }

        //The array Order classifies individuals by  fitness
        // by equal or decreasing order.
        Order[i] = Champ;
        Fitness[Champ] = 0;
        if (test) {
            System.out.println(i + "th ind. is  No "
                    + Champ);
        }
    }
}

//Each individual of the top ten
// produces 10  copies.
private static void Reproduction() {
    if (test) {
        System.out.println("Reproduction");
    }
    if (test) {
        System.out.println("The best = " + Order[0]);
    }
    Individualc = new String[limit];
    int counter = 0;
    for (int top = 0; top < 10; top++) {
        for (int j = 0; j < 10; j++) {
            Individualc[counter] = Individual[Order[top]];
            counter = counter + 1;
        }
    }
    /*
 for(int j = 0; j< counter; j++)
 Individual[j] = Individualc[j];
     */
    System.arraycopy(Individualc, 0, Individual, 0, counter);
```

```
    }

    //A mutation of ind j is done at place placeMut
    private static  String  mutateCoefficient(String s)
    {
        if (test) {
            System.out.println(" Coefficient         = " + s);
        }
        int placeMut = r.nextInt(nChars + 4);

        if (test) {
            System.out.println("  placeMut = " + placeMut);
        }

        char c;
        String sign;
        String power;



        //Case sign
        if (placeMut == nChars + 2) {
            sign = generateSign();
            s = s.substring(0, nChars + 2)
                    + sign + s.substring(nChars + 3);
            if (test) {
                System.out.println(" Sign mutated = " + s);
            }
        }


        //Case power:
        if ((placeMut == nChars + 3) | (placeMut == nChars + 4)) {
            power = addPower(nChars);
            s = s.substring(0, nChars + 3)
                    + power + s.substring(nChars + 4);
            if (test) {
                System.out.println("Power mutated = " + s);
            }
        }


        //Case basic string
        if ( (0 < placeMut) & (placeMut < nChars ))  {
            c = randomDigit();
            s = s.substring(0, placeMut)
```

```
                + c + s.substring(placeMut + 1);
        if (test) {
            System.out.println("Basic mutated = " + s);
        }
    }
    return s;
}



private static String mutateIndj(int j)
{
    String s = Individual[j];
    String output = "";
    int l = nChars + 4;
    for(int i = 0; i < 3; i++)
    {
        String w = s.substring(i*l, (i+1)*l);
        output = output + mutateCoefficient(w);
    }
    return output;
}

//We consider only one type of mutation:
//substitution of one char by another one.
//The first copy of the winner does not mutate.
private static void Mutation() {
    for (int j = 1; j < nIndiv; j++)
        Individual[j] = mutateIndj(j);


    //Some individuals are generated ab initio
    //We hope that they will help to escape
    //from local optima.
    int initial = nIndiv - 20;
    if (initial > 0) {
        for (int i = initial; i < nIndiv; i++) {
            if (test) {
                System.out.println("i = " + i);
            }
            Individual[i] = "";
            Individual[i] = generateIndividual();
        }
    }
}
```

```
    //Two strings recombine and produce two offspring.
    private static void Recombination() {
        for (int j = 50; j < nIndiv; j++) {
            //Define place of mutation
            int m = r.nextInt(nIndiv);
            int n = r.nextInt(nIndiv);
            String a = Individual[m];
            String bb = Individual[n];
            int placeRec = r.nextInt(2 * nChars + 9);
            Individual[m] = a.substring(0, placeRec)
                    + bb.substring(placeRec);
            Individual[n] = bb.substring(0, placeRec)
                    + a.substring(placeRec);
        }
    }

    //Mutation rate can increase
    private static void mutationRedef() {
        String c = Individual[0]; //champ reported
        h hf = decoder(c);
        double error = findMaxError(zMax, N, hf);
        /* System.out.println("gen " + gen +
 " Error " + error);*/
/* System.out.println("mutation rate = " +
    mutationRate);*/
        newError = error;
        deltaError = Math.abs(newError - oldError);
        /*System.out.println("Error= " + error +
" deltaError= " + deltaError); */
        oldError = newError;
        if (deltaError < 0.001) {
            mutationRate = 1.1 * mutationRate;
            if (mutationRate > 1) {
                mutationRate = 1;
            }

            /*  System.out.println("gen " + gen + "
    mutRedef");*/
        } else;
    }

    //Overall method
    private static void dynamics(int gen) {
        //Individuals are sorted by fitness
        Sorting(gen);
```

```
    //for(int i = 1; i<= 10;i++)
    //System.out.println(  Individual(Order(i))
    if (gen % 100 == 0) report();
    //The top ten are preferentially reproduced
    Reproduction();
    //The new population is subjected to mutation
    Mutation();
    Recombination();
    //every 100 generations, stagnation is revised
    if (gen % 100 == 0) {
        mutationRedef();
    }
}

//Decodes a string into a number
private static void decode() {
    System.out.println();
    System.out.println("Numeric values at decoding");
    for (int i = 0; i < nIndiv; i++) {
        System.out.println("i = " + i);
        System.out.println("All = " + Individual[i]);
        decoder(Individual[i]);
    }
}

//test of encoding and decoding
private static void test() {
    nIndiv = 1;
    System.out.println("Strings at Initialization");
    Initialization();
    decode();
    trapRule();
    System.out.println("\r Test ");
    h hf = new h(4, 0.4107102082519, 0.096088271997664);
    double error = findMaxError(zMax, N, hf);
    System.out.println("Max error at test= " + error);
}

public static void main(String[] args) {
    testEncoding = false; //else true
    if (testEncoding) {
        test();
    } else {
        test = false;
        //The whole cumulative function is estimated
        //by the trapezoidal rule
```

```
            trapRule();

            //The parameters in the hFamily are optimized
            //by a genetic algorithm.

            nIndiv = 500;
            Initialization();
            System.out.println("Running... ");
            nGen = 300000;
            oldError = 1000;
            mutationRate = 1;
            for (int gen = 1; gen <= nGen; gen++) {
                //System.out.println("Gen = " + gen);
                generation = gen;
                dynamics(gen);
            }
        }
    }
}//End of Program E282 LogisticApproximation
```

Problems of Chapter 15

286, page 320. Factorials over int

```
//Program E286 Factorial

package ejvol5v2p;

public class Factorial {

    private static int factorial(int n) {
        int fact = 1;
        for (int i = 1; i <= n; i++) {
            fact = fact * i;
        }
        return fact;
    }

    public static void main(String[] args) {
        int n = 300;
        for (int i = 1; i < n; i++) {
            System.out.println(i + "  " + factorial(i));
        }
```

```
    }
}//End of Program E286 Factorial
```

290, page 321. The output can be negative for factorials in the range from 20 to 30. This means that the factorial functions overgrow the capacity of the numerical system used by the type long of Java.

294, page 323. Double factorial, type long:

```
//Program E294 DoubleFactorial
//Calculates the double factorial function
//Implements a recursive definition.

package ejvol5v2p;

public class DoubleFactorial {

//Returns numb!!
    private static long doubleFactorial(long numb) {
        if (numb <= 0) {
            return 1;
        } else {
            return numb * doubleFactorial(numb - 2);
        }
    }

    public static void main(String[] args) {
        int n = 30;
        for (int i = -1; i < n; i++) {
            System.out.println(i + "  " + doubleFactorial(i));
        }
    }
}//End of Program E294 DoubleFactorial
```

295, page 323. Answer: Double factorial, BigNumbers

```
//Program E295 DoubleFBig
//Calculates the double factorial function.
//Big Numbers are used.
//Implements a recursive definition.
```

```
package ejvol5v3p;

import java.math.BigDecimal;


public class DoubleFBig {

    private static final BigDecimal ONE = BigDecimal.valueOf(1);
    private static final BigDecimal TWO = BigDecimal.valueOf(2);

    //Returns numb!!
    private static BigDecimal doubleFactorial(BigDecimal numb) {
        //output of compare:
        //+1 if numb is greater than ONE.
        //0 if they are equal
        //-1 if numb is less than ONE
        if (numb.compareTo(ONE) < 1) {
            return ONE;
        } else {
            return doubleFactorial(numb.subtract(TWO)).
                    multiply(numb);
        }
    }

    public static void main(String[] args) {
        int n = 70;
        for (int i = 1; i < n; i++) {
            System.out.println(i + "  "
                    + doubleFactorial(BigDecimal.valueOf(i)));
        }
        //Test for precision + overflow
        System.out.println(" Test: (n)!!/(n-2)!! = n");
        for (int i = 3; i < n; i++) {
            System.out.println(i + "  "
                    + doubleFactorial(BigDecimal.valueOf(i)).divide(
                            doubleFactorial(BigDecimal.valueOf(i - 2))));
        }
    }
}//End of Program E295 DoubleFBig
```

297, page 324. Gamma function, type long

```
//Program E297 GammaF
//Calculates the gamma function of
//integers 1,2,3,4,5...
//and halfIntegers 1/2, 3/2, 5/2, 7/2,...

package ejvol5v2p;

public class GammaF {

    private static final double PI = 3.14159265358979323846;

    //Returns numb!
    private static long factorial(long numb) {
        if (numb <= 1) {
            return 1;
        } else {
            return numb * factorial(numb - 1);
        }
    }


    //Returns numb!!
    private static long doubleFactorial(long numb) {
        if (numb <= 0) {
            return 1;
        } else {
            return numb * doubleFactorial(numb - 2);
        }
    }


    //Returns the Gamma function
    private static double Gamma(double x) {
        //Cuts the decimal part: 3.5 --> 3.
        int xRounded = (int) x;
        //System.out.println( x + "   " + xRounded);
        if (xRounded == x) {
            return factorial(xRounded - 1);
        } else {
            return Math.sqrt(PI) * doubleFactorial(2 * xRounded - 1)
                    / Math.pow(2, xRounded);
        }
    }

    public static void main(String[] args) {
        int n = 40;
```

```
        for (int i = -1; i < n; i++) {
            System.out.println(i + "  " + Gamma(i));
        }
        for (int i = 0; i < n; i++) {
            double r = i + 0.5;
            System.out.println(r + "  " + Gamma(r));
        }

    }

}//End of Program E297 GammaF
```

298, page 324. Gamma function with BigNumbers

```
//Program E298 GammaFBig
//Calculates the gamma function of
//integers 1,2,3,4,5...
//and halfIntegers 1/2, 3/2, 5/2, 7/2,...
//Uses big numbers.
//Pi is calculated to any desired accuracy:
//Slight and commented modification of the code
//presented by Chakraborty (2010)

/*
http://blog.taragana.com/index.php/archive/
calculate-pi-to-arbitrary-precision-sample-java-code/
 */
//Cited 15 Ap 2010.
//Original and present codes are free.

package ejvol5v2p;

import java.math.BigDecimal;


public class GammaFBig {

    /**
     * constants
     */
    private static final BigDecimal ZERO = BigDecimal.valueOf(0);
    private static final BigDecimal ONE = BigDecimal.valueOf(1);
    private static final BigDecimal TWO = BigDecimal.valueOf(2);
```

```
   private static final BigDecimal FOUR = BigDecimal.valueOf(4);
   private static BigDecimal sqrtPi;

   //Precision scale: number of digits after
   //the decimal point
   private static final int SCALE = 30;

   private static BigDecimal error;
   private static int iterations;
   private static boolean traceFlag;

   private static final int MAXNITERATIONS = 50;

   /**
    * rounding mode to use during pi computation 3.142 is rounded to 3.14 while
    * 3.14159 is to 3.1416.
    */
   private static final int ROUNDINGMODE = BigDecimal.ROUND_HALF_EVEN;

   /**
    * ************************************
    * Pi is calculated
* *********************************
    */
   /**
    * Compute the value of pi to the specified number of digits after the
    * decimal point. The value is computed using Machin's formula:
    *
    * pi/4 = 4*arctan(1/5) - arctan(1/239) pi = (4*arctan(1/5) -
    * arctan(1/239))*4 and a power series expansion of arctan(x) to sufficient
    * precision.
    * @param digits
    * @return
    */
   public static BigDecimal computePi(int digits) {
       //Precision of computations is greater than
       //demanded precision. With morePrecision = 10
       //corrections are done to the last digit only.

       //arctan(1/5)
       BigDecimal arctan1_5 = arctan(5, SCALE);
       //arctan(1/239)
       BigDecimal arctan1_239 = arctan(239, SCALE);

       //Arithmetic operators:
       //To add x: .add(x);
```

```
        //To multiply by x: .multiply(x)
        //To substract x: .substract(x)
        //To divide by x: .divide(x)
        //Operator can be chained and are executed
        //from left to right.
        // pi   = (4*arctan(1/5) - arctan(1/239))*4
        BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
                arctan1_239).multiply(FOUR);
        //only demanded precision is reported
        return pi.setScale(digits,
                BigDecimal.ROUND_HALF_UP);
    }

    /**
     * Compute the value, in radians, of the arctangent of the inverse of th
     * supplied integer to the specified number of digits after the decimal
     * point. The value is computed using the power series expansion for the
     * tangent:
     *
     * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 + (x^9)/9 ...
     * @param inverseX
     * @param SCALE
     * @return
     */

    //All results must have a scale of accurateness
    public static BigDecimal arctan(int inverseX,
            int SCALE) {
        BigDecimal result, numer, term;
        BigDecimal invX = BigDecimal.valueOf(inverseX);
        BigDecimal invX2
                = BigDecimal.valueOf(inverseX * inverseX);

        numer = BigDecimal.ONE.divide(invX,
                SCALE, ROUNDINGMODE);

        result = numer;
        int i = 1;
        //Loop to compute the series.
        //The loop ends when a term can be rounded to zero
        //given the scale of accurateness.
        do {
            numer
                    = numer.divide(invX2, SCALE, ROUNDINGMODE);
            int denom = 2 * i + 1;
            term
```

```
                = numer.divide(BigDecimal.valueOf(denom),
                        SCALE, ROUNDINGMODE);
        //The first term is term number 1.
        //Terms with even i are  subtracted
        //terms with odd i are added
        if ((i % 2) != 0) {
            result = result.subtract(term);
        } else {
            result = result.add(term);
        }
        i++;
    } //Can the term be rounded to zero?
    while (term.compareTo(BigDecimal.ZERO) != 0);
    return result;
}

//********Square root of a BigDecimal number************
//---------------------------
// Get initial approximation:
// the square root as double
//---------------------------
private static BigDecimal
        getInitialApproximation(double n) {
    double s = Math.sqrt(n);
    BigDecimal g = BigDecimal.valueOf(s);
    return g;
}

//------
// Trace
//------
private static void trace(String s) {
    if (traceFlag) {
        System.out.println(s);
    }
}
//----------------
// Get square root
//----------------

private static BigDecimal sqroot(BigDecimal n) {

    // Make sure n is a positive number
    if (n.compareTo(ZERO) <= 0) {
        throw new IllegalArgumentException();
    }
```

```
        //n is transformed to type double
        double s = n.doubleValue();
        BigDecimal initialGuess;
        if (s > 0) {
            initialGuess = getInitialApproximation(s);
        } else {
            initialGuess = ONE;
        }
        trace("Initial guess " + initialGuess.toString());
        BigDecimal lastGuess;
        BigDecimal guess = new BigDecimal(initialGuess.toString());

        // This is the algorithm:
        iterations = 0;
        boolean more = true;
        while (more) {
            lastGuess = guess;
            /* n/g */
            guess = n.divide(guess, SCALE,
                    BigDecimal.ROUND_HALF_UP);
            /* n/g + g*/
            guess = guess.add(lastGuess);
            /* (n/g + g)/2 */
            guess = guess.divide(TWO, SCALE,
                    BigDecimal.ROUND_HALF_UP);
            trace("Next guess " + guess.toString());
            error = n.subtract(guess.multiply(guess));
            if (++iterations >= MAXNITERATIONS) {
                more = false;
            } else if (lastGuess.equals(guess)) {
                more = error.abs().compareTo(ONE) >= 0;
            }
        }
        return guess;

    }

    //Returns numb!
    private static BigDecimal sqrtPi() {
        BigDecimal pi = computePi(SCALE);
        //System.out.println("pi = " + pi);
        BigDecimal x = sqroot(pi);
        //System.out.println("sqrt(2*pi) = " + x);
        return x;
    }
```

```
    //Returns numb!
    private static BigDecimal factorial(BigDecimal numb) {
        //output of compare:
        //+1 if numb is greater than ONE.
        //0 if they are equal
        //-1 if numb is less than ONE
        if (numb.compareTo(ONE) < 1) {
            return ONE;
        } else {
            return factorial(numb.subtract(ONE)).
                    multiply(numb);
        }
    }

//Returns numb!!
    private static BigDecimal doubleFactorial(BigDecimal numb) {
        //output of compare:
        //+1 if numb is greater than ONE.
        //0 if they are equal
        //-1 if numb is less than ONE
        if (numb.compareTo(ONE) < 1) {
            return ONE;
        } else {
            return doubleFactorial(numb.subtract(TWO)).
                    multiply(numb);
        }
    }

    //Returns x to power n
    private static BigDecimal Power(double x, int n) {
        BigDecimal xBD = BigDecimal.valueOf(x);
        if (n == 0) {
            return ONE;
        } else {
            return xBD.multiply(Power(x, n - 1));
        }
    }

    //Returns the Gamma function
    private static BigDecimal Gamma(double x) {
        //Cuts the decimal part: 3.5 --> 3.
        int n = (int) x;
        BigDecimal xRounded = BigDecimal.valueOf(n);
        //System.out.println( x + "   " + xRounded);
        //One over square root of Pi
        sqrtPi = sqrtPi();
```

```
        if (n == x) {
            return factorial(xRounded.subtract(ONE));
        } else {
            BigDecimal a = xRounded.multiply(TWO); //2*xRounded
            a = a.subtract(ONE); //a-1
            a = doubleFactorial(a);
            a = a.multiply(sqrtPi);
            BigDecimal p = Power(2, n);
            a = a.divide(p);
            return a;
        }
    }

    public static void main(String[] args) {

        System.out.println("Gamma of integer values");
        int n = 10;
        for (int i = -1; i < n; i++) {
            System.out.println(i + "  " + Gamma(i));
        }
        System.out.println("Gamma of half integer values");
        for (int i = 0; i < n; i++) {
            double r = i + 0.5;
            System.out.println(r + "  " + Gamma(r));
        }
        //Test: Gamma(r+1)/Gamma(r) = r;
        System.out.println("Test: Gamma(r+1)/Gamma(r) = r");
        for (int i = 3; i < 300; i++) {
            double r = i + 0.5;
            BigDecimal a = Gamma(r + 1);
            BigDecimal b = Gamma(r);
            System.out.println(r + "  " + a.divide(b));
        }
    }

}//End of Program E298 GammaFBig
```

300, page 324. Some values of the Beta function (corrected):

$\beta(1,1) = \frac{\Gamma(1)\Gamma(1)}{\Gamma(1+1)} = 1/1 = 1$
$\beta(1,2) = \beta(2,1) = \frac{\Gamma(2)\Gamma(1)}{\Gamma(2+1)} = 1/2 = 0.5$
$\beta(2,4) = \beta(4,2) = \frac{\Gamma(2)\Gamma(4)}{\Gamma(2+4)} = \frac{1\times 6}{120} = 0.05$

$\beta(20,20) = \beta(4,2) = \frac{\Gamma(20)\Gamma(20)}{\Gamma(40)} < (\frac{1}{2})^{20} < 9 \times 10^{-7}$
In general, when d.f. grow, the beta values tend to zero.

301, page 324. Answer. The beta function:

```
//Program E301 Beta
//Outputs the Beta function
//for inputs x and y.
//Two implementations:
//ordinary and integral representations.
//The integral representation is best for large inputs.

package ejvol5v2p;

public class Beta {

    private static final double PI = 3.14159265358979323846;
    private static double nu1, nu2;


    //Returns numb!
    private static long factorial(long numb) {
        if (numb <= 1) {
            return 1;
        } else {
            return numb * factorial(numb - 1);
        }
    }

    //Returns numb!!
    private static long doubleFactorial(long numb) {
        if (numb <= 0) {
            return 1;
        } else {
            return numb * doubleFactorial(numb - 2);
        }
    }

    //Returns the Gamma function
    private static double Gamma(double x) {
        //Cuts the decimal part: 3.5 --> 3.
        int xRounded = (int) x;
        //System.out.println( x + "    " + xRounded);
        if (xRounded == x) {
            return factorial(xRounded - 1);
```

```
        } else {
            return Math.sqrt(PI) * doubleFactorial(2 * xRounded - 1)
                    / Math.pow(2, xRounded);
        }
    }

    //Returns the Gamma function
    private static double beta1(double x, double y) {
        double b = Gamma(x) * Gamma(y) / Gamma(x + y);
        return b;
    }



    //Returns the Beta argument function in integral form
    //with nu1 and nu2 d.f.
    private static double f(double x) {

        double f = 2 * (Math.pow(x, 2 * nu2 - 1))
                * (Math.pow(1 - x * x, nu1 - 1));
        return f;
    }


    //Returns the integral under the function f
    //in within 0 and z. Precision=1/N.
    //Simpson's rule adapted for reuse.
    private static double HalfBodySimpson(double z, long N) {

        double h = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * h);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * h);
        }
        sum = sum + f((2 * m) * h);
        sum = (h / 3) * sum;
        return sum;
    }

    //Returns the Beta   function in integral form
    //with nu1 and nu2 d.f.
    private static double beta2(double nu1, double nu2) {
```

```
        double answer;
        int i = 3;
        int N = (int) Math.pow(10, i);
        double z = 1;
        answer = HalfBodySimpson(z, N);
        return answer;
    }

    public static void main(String[] args) {
        System.out.println("Ordinary and integral "
                + "representations of the beta function");
        for (int i = 1; i < 30; i++) {
            for (int j = 1; j < 30; j++) {
                nu1 = i;
                nu2 = j;
                System.out.println("i = " + i   + " j = " + j
                        + " beta = " + beta1(i, j) + "  " + +beta2(i, j));
            }
        }
    }
}//End of Program E301 Beta
```

302, page 324. Beta function calculated with the help of BigNumbers and by the integral representation. The integral method is simpler and results differ from those of BigNumbers beyond the tenth decimal cipher.

```
//Program E302 BetaBig
//Outputs the beta function
//for inputs x and y.
//The integral representation is compared
//with another that uses big numbers.
//Pi is calculated to any desired accuracy:
//Slight and commented modification of the code
//presented by Chakraborty (2010)

/*
http://blog.taragana.com/index.php/archive/
calculate-pi-to-arbitrary-precision-sample-java-code/
 */
//Cited 15 Ap 2010.
//Original and present codes are free.
//Assessment: the integral method is simpler and
//results differ from those of BigNumbers
//beyond 10 significant ciphers.
```

```
package ejvol5v2p;

import java.math.BigDecimal;


public class BetaBig {

    /**
     * constants
     */
    private static final BigDecimal ZERO = BigDecimal.valueOf(0);
    private static final BigDecimal ONE = BigDecimal.valueOf(1);
    private static final BigDecimal TWO = BigDecimal.valueOf(2);
    private static final BigDecimal FOUR = BigDecimal.valueOf(4);
    private static BigDecimal sqrtPi;

    //Precision SCALE: number of digits after
    //the decimal point
    private static final int SCALE = 100;

    private static BigDecimal error;
    private static int iterations;
    private static boolean traceFlag;

    private static final int MAXNITERATIONS = 50;

    /**
     * rounding mode to use during pi computation 3.142 is rounded to 3.14 w
     * 3.14159 is to 3.1416.
     */
    private static final int ROUNDINGMODE = BigDecimal.ROUND_HALF_EVEN;

    /**
     * ***********************************
     * Pi is calculated
 * ***********************************
     */
    /**
     * Compute the value of pi to the specified number of digits after the
     * decimal point. The value is computed using Machin's formula:
     *
     * pi/4 = 4*arctan(1/5) - arctan(1/239) pi = (4*arctan(1/5) -
     * arctan(1/239))*4 and a power series expansion of arctan(x) to suffici
     * precision.
     * @param digits
```

```
 * @return
 */
public static BigDecimal computePi(int digits) {
    //Precision of computations is greater than
    //demanded precision. With morePrecision = 10
    //corrections are done to the last digit only.

    //arctan(1/5)
    BigDecimal arctan1_5 = arctan(5, SCALE);
    //arctan(1/239)
    BigDecimal arctan1_239 = arctan(239, SCALE);

    //Arithmetic operators:
    //To add x: .add(x);
    //To multiply by x: .multiply(x)
    //To substract x: .substract(x)
    //To divide by x: .divide(x,SCALE, ROUNDINGMODE)
    //Operator can be chained and are executed
    //from left to right.
    // pi   = (4*arctan(1/5) - arctan(1/239))*4
    BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
            arctan1_239).multiply(FOUR);
    //only demanded precision is reported
    return pi.setScale(digits,
            BigDecimal.ROUND_HALF_UP);
}

/**
 * Compute the value, in radians, of the arctangent of the inverse of the
 * supplied integer to the specified number of digits after the decimal
 * point. The value is computed using the power series expansion for the arc
 * tangent:
 *
 * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 + (x^9)/9 ...
 * @param inverseX
 * @param SCALE
 * @return
 */

//All results must have a SCALE of accurateness
public static BigDecimal arctan(int inverseX,
        int SCALE) {
    BigDecimal result, numer, term;
    BigDecimal invX = BigDecimal.valueOf(inverseX);
    BigDecimal invX2
            = BigDecimal.valueOf(inverseX * inverseX);
```

```
        numer = BigDecimal.ONE.divide(invX,
                SCALE, ROUNDINGMODE);

        result = numer;
        int i = 1;
        //Loop to compute the series.
        //The loop ends when a term can be rounded to zero
        //given the SCALE of accurateness.
        do {
            numer
                    = numer.divide(invX2, SCALE, ROUNDINGMODE);
            int denom = 2 * i + 1;
            term
                    = numer.divide(BigDecimal.valueOf(denom),
                            SCALE, ROUNDINGMODE);
            //The first term is term number 1.
            //Terms with even i are  subtracted
            //terms with odd i are added
            if ((i % 2) != 0) {
                result = result.subtract(term);
            } else {
                result = result.add(term);
            }
            i++;
        } //Can the term be rounded to zero?
        while (term.compareTo(BigDecimal.ZERO) != 0);
        return result;
    }

    //********Square root of a BigDecimal number************
    //--------------------------
    // Get initial approximation:
    // the square root as double
    //--------------------------
    private static BigDecimal
            getInitialApproximation(double n) {
        double s = Math.sqrt(n);
        BigDecimal g = BigDecimal.valueOf(s);
        return g;
    }

    //------
    // Trace
    //------
    private static void trace(String s) {
```

```
    if (traceFlag) {
        System.out.println(s);
    }
}
//----------------
// Get square root
//----------------

private static BigDecimal sqroot(BigDecimal n) {

    // Make sure n is a positive number
    if (n.compareTo(ZERO) <= 0) {
        throw new IllegalArgumentException();
    }
    //n is transformed to type double
    double s = n.doubleValue();
    BigDecimal initialGuess;
    if (s > 0) {
        initialGuess = getInitialApproximation(s);
    } else {
        initialGuess = ONE;
    }
    trace("Initial guess " + initialGuess.toString());
    BigDecimal lastGuess;
    BigDecimal guess = new BigDecimal(initialGuess.toString());

    // This is the algorithm:
    iterations = 0;
    boolean more = true;
    while (more) {
        lastGuess = guess;
        /* n/g */
        guess = n.divide(guess, SCALE,
                BigDecimal.ROUND_HALF_UP);
        /* n/g + g*/
        guess = guess.add(lastGuess);
        /* (n/g + g)/2 */
        guess = guess.divide(TWO, SCALE,
                BigDecimal.ROUND_HALF_UP);
        trace("Next guess " + guess.toString());
        error = n.subtract(guess.multiply(guess));
        if (++iterations >= MAXNITERATIONS) {
            more = false;
        } else if (lastGuess.equals(guess)) {
            more = error.abs().compareTo(ONE) >= 0;
        }
```

```
        }
        return guess;

    }

    //Returns numb!
    private static BigDecimal sqrtPi() {
        BigDecimal pi = computePi(SCALE);
        //System.out.println("pi = " + pi);
        BigDecimal x = sqroot(pi);
        //System.out.println("sqrt(2*pi) = " + x);
        return x;
    }

    //Returns numb!
    private static BigDecimal OneOverSqrt2Pi() {
        BigDecimal pi = computePi(SCALE);
        //System.out.println("pi = " + pi);
        BigDecimal x = sqroot(pi.multiply(TWO));
        //System.out.println("sqrt(2*pi) = " + x);
        BigDecimal k = ONE.divide(x, SCALE, ROUNDINGMODE);
        //System.out.println("k = " + k);
        return k;
    }

    //Returns numb!
    private static BigDecimal factorial(BigDecimal numb) {
        //output of compare:
        //+1 if numb is greater than ONE.
        //0 if they are equal
        //-1 if numb is less than ONE
        if (numb.compareTo(ONE) < 1) {
            return ONE;
        } else {
            return factorial(numb.subtract(ONE)).
                    multiply(numb);
        }
    }

    //Returns numb!!
    private static BigDecimal doubleFactorial(BigDecimal numb) {
        //output of compare:
        //+1 if numb is greater than ONE.
        //0 if they are equal
        //-1 if numb is less than ONE
        if (numb.compareTo(ONE) < 1) {
```

```
        return ONE;
    } else {
        return doubleFactorial(numb.subtract(TWO)).
                multiply(numb);
    }
}

//Returns x to power n
private static BigDecimal Power(double x, int n) {
    BigDecimal xBD = BigDecimal.valueOf(x);
    if (n == 0) {
        return ONE;
    } else {
        return xBD.multiply(Power(x, n - 1));
    }
}

//Returns the Gamma function
private static BigDecimal Gamma(double x) {
    //Cuts the decimal part: 3.5 --> 3.
    int n = (int) x;
    BigDecimal xRounded = BigDecimal.valueOf(n);
    //System.out.println( x + "   " + xRounded);
    //One over square root of Pi
    sqrtPi = sqrtPi();

    if (n == x) {
        return factorial(xRounded.subtract(ONE));
    } else {
        BigDecimal a = xRounded.multiply(TWO); //2*xRounded
        a = a.subtract(ONE); //a-1
        a = doubleFactorial(a);
        a = a.multiply(sqrtPi);
        BigDecimal p = Power(2, n);
        a = a.divide(p);
        return a;
    }
}

//Returns the Gamma function.
//BigDecimals are used.
private static BigDecimal beta1(double x, double y) {

    BigDecimal a = Gamma(x);
    BigDecimal b = Gamma(y);
    BigDecimal c = Gamma(x + y);
```

```
        BigDecimal d = a.multiply(b);
        BigDecimal e = d.divide(c, SCALE, ROUNDINGMODE);
        return e.setScale(SCALE, BigDecimal.ROUND_HALF_UP);

    }

    private static double nu1, nu2;

    //Returns the beta argument function in integral form
    //with nu1 and nu2 d.f.
    private static double f(double x) {
        double f = Math.pow(x, nu1 - 1) * (Math.pow(1 - x, nu2 - 1));
        return f;
    }

    //Returns the integral under the function f
    //in within 0 and z. Precision=1/N.
    //Simpson's rule adapted for reuse.
    private static double HalfBodySimpson(double z, long N) {

        double h = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * h);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * h);
        }
        sum = sum + f((2 * m) * h);
        sum = (h / 3) * sum;
        return sum;
    }

    //Returns the beta   function in integral form
    //with nu1 and nu2 d.f.
    private static double beta2(double nu1, double nu2) {
        double answer;
        int i = 3;
        int N = (int) Math.pow(10, i);
        double z = 1;
        answer = HalfBodySimpson(z, N);
        return answer;
    }

    public static void main(String[] args) {
```

```
        for (int i = 3; i < 30; i++) {
            for (int j = 3; j < 30; j++) {
                nu1 = i;
                nu2 = j;
                BigDecimal a = beta1(i, j);

                System.out.println("i = " + i   + " j = " + j
                        + " beta = " + a + "   " + +beta2(i, j));
            }
        }
    }

}//End of Program E302 BetaBig
```

304, page 325. p-values for a binomial distribution.

```
//Program E304 PValBi
//Outputs the p-value of
//the  binomial distribution
//for  n coins (low values),
//probability of head p,
//and input k.

package ejvol5v2p;

public class PValBi {

    private static final double COEFFICIENTSBINOMIAL[]
            = new double[200];
    private static final double TERMSBINOMIAL[]
            = new double[200];

    private static int n;

    private static double p;

    private static int k;

    //Prints data in vector
    private static void printVector(double Vect[], int n) {
        for (int i = 0; i <= n; i++) {
            System.out.println(Vect[i]);
        }
```

```
    }

    //Sums data in vector with real numbers
    private static double sumVector(double Vect[], int n) {
        double sum = 0;
        for (int i = 0; i <= n; i++) {
            sum = sum + Vect[i];
        }
        return sum;
    }

    //Returns numb!
    private static long factorial(long numb) {
        if (numb <= 1) {
            return 1;
        } else {
            return numb * factorial(numb - 1);
        }
    }

    //The  distribution  of the binomial distribution
    // for  n coins
    //probability of head p,
    //event k
    private static void binomialDistribution(int n,
            double p) {
        long biCoeff;
        for (int r = 0; r < n + 1; r++) {
            biCoeff = factorial(n)
                    / (factorial(r) * factorial(n - r));
            COEFFICIENTSBINOMIAL[r] = biCoeff;
        }
        System.out.println("Coefficients ");
        printVector(COEFFICIENTSBINOMIAL, n);
        System.out.println("Sum = "
                + sumVector(COEFFICIENTSBINOMIAL, n));

        for (int r = 0; r < n + 1; r++) {

            TERMSBINOMIAL[r] = COEFFICIENTSBINOMIAL[r]
                    * Math.pow(p, r) * Math.pow(1 - p, n - r);
        }
        System.out.println("Probabilities for 0,1,2...");
        System.out.println("Paste these data to Excel, "
                + " OpenOffice or Gnumeric and \nget a bell.");
        printVector(TERMSBINOMIAL, n);
```

```
    sumVector(TERMSBINOMIAL, n);
    System.out.println("Sum = "
            + sumVector(TERMSBINOMIAL, n));
}

//The  p-value of the binomial distribution
// for  n repeats
//probability of head p,
//event k
private static double pValuesBinomial(int n,
        double p, int k) {
    double mean = n * p;
    double sum = 0;
    if (k >= mean) {
        for (int r = k+1; r <= n; r++) {
            sum = sum + TERMSBINOMIAL[r];
        }
    } else {
        for (int r = 0; r < k; r++) {
            sum = sum + TERMSBINOMIAL[r];
        }
    }
    return sum;
}

public static void main(String[] args) {
    n = 8;
    p = 0.5;
    System.out.println("Binimial distribution");
    System.out.println("n = " + n + " p = " + p);

    binomialDistribution(n, p);
    k = 2;
    System.out.println("\nEvent = " + k);
    double pVal = pValuesBinomial(n, p, k);
    System.out.println("One tail p-value of " + k
            + " = " + pVal);
    System.out.println("Two tails p-value of " + k
            + " = " + 2 * pVal);
    k = 6;
    System.out.println("\nEvent = " + k);
    pVal = pValuesBinomial(n, p, k);
    System.out.println("One tail p-value of " + k
            + " = " + pVal);
    System.out.println("Two tails p-value of " + k
            + " = " + 2 * pVal);
```

```
    }

}//End of Program E304 PValBi
```

305, page 325. p-values for a binomial distribution. Big numbers are used.

```
//Program E305 PValBiBig
//Outputs the p-value of
//the  binomial distribution
//for  n coins,
//probability of head p,
//and input k.
//Big numbers are used.

package ejvol5v2p;

import java.math.BigDecimal;

public class PValBiBig {

    private static final BigDecimal COEFFICIENTSBINOMIAL[]
            = new BigDecimal[501];
    private static final BigDecimal TERMSBINOMIAL[]
            = new BigDecimal[501];

    private static int n;

    private static double p;

    private static int k;

    //Precision SCALE: number of digits after
    //the decimal point
    private static final int SCALE = 100;

    /**
     * rounding mode to use during pi computation 3.142 is rounded to 3.14 w
     * 3.14159 is to 3.1416.
     */
    private static final int ROUNDINGMODE = BigDecimal.ROUND_HALF_EVEN;
```

```
    private static final BigDecimal ONE = BigDecimal.valueOf(1);

    private static final BigDecimal TWO = BigDecimal.valueOf(2);

    private static final boolean PRINT = false;

    //Prints data in vector
    private static void printVector(BigDecimal Vect[], int n) {
        for (int i = 0; i <= n; i++) {
            System.out.println(Vect[i]);
        }
    }

    //Sums data in vector with real numbers
    private static BigDecimal sumVector(BigDecimal Vect[], int n) {
        BigDecimal sum = BigDecimal.valueOf(0);
        for (int i = 0; i <= n; i++) {
            sum = sum.add(Vect[i]);
        }
        return sum;
    }

//Returns numb!
    private static BigDecimal factorial(BigDecimal numb) {
        //output of compare:
        //+1 if numb is greater than ONE.
        //0 if they are equal
        //-1 if numb is less than ONE
        if (numb.compareTo(ONE) < 1) {
            return ONE;
        } else {
            return factorial(numb.subtract(ONE)).
                    multiply(numb);
        }
    }

//Returns x to power n
    private static BigDecimal Power(double x, int n) {
        BigDecimal xBD = BigDecimal.valueOf(x);
        if (n == 0) {
            return ONE;
        } else {
            return xBD.multiply(Power(x, n - 1));
        }
    }
```

```
//The  distribution  of the binomial distribution
// for  n coins
//probability of head p,
//event k
private static void binomialDistribution(int n,
        double p) {
    BigDecimal biCoeff;

    for (int r = 0; r < n + 1; r++) {
        BigDecimal N = BigDecimal.valueOf(n);
        BigDecimal R = BigDecimal.valueOf(r);
        biCoeff = factorial(N).divide(
                factorial(R).multiply(factorial(N.subtract(R))),
                SCALE, ROUNDINGMODE);
        COEFFICIENTSBINOMIAL[r] = biCoeff;
    }
    for (int r = 0; r < n + 1; r++) {

        TERMSBINOMIAL[r] = (COEFFICIENTSBINOMIAL[r].multiply(
                Power(p, r))).multiply(Power(1 - p, n - r));

    }
    sumVector(TERMSBINOMIAL, n);
    if (PRINT)
    {
        System.out.println("Coefficients ");
        printVector(COEFFICIENTSBINOMIAL, n);
        System.out.println("Sum = "
               + sumVector(COEFFICIENTSBINOMIAL, n));


        System.out.println("Probabilities for 0,1,2...");
        System.out.println("Paste these data to Excel, "
                + " OpenOffice or Gnumeric and \nget a bell.");
        printVector(TERMSBINOMIAL, n);

        System.out.println("Sum = "
            + sumVector(TERMSBINOMIAL, n));
    }
}

//The  p-value of the binomial distribution
// for  n repeats
//probability of head p,
//event k
private static BigDecimal pValuesBinomial(int n,
```

```
            double p, int k) {
        double mean = n * p;
        BigDecimal sum = BigDecimal.valueOf(0);
        if (k >= mean) {
            for (int r = k+1; r < n; r++) {
                sum = sum.add(TERMSBINOMIAL[r]);
            }
        } else {
            for (int r = 0; r < k; r++) {
                sum = sum.add(TERMSBINOMIAL[r]);
            }
        }
        return sum;
    }

    public static void main(String[] args) {
        n = 500;
        p = 0.5;
        System.out.println("Binimial distribution");
        System.out.println("n = " + n + " p = " + p);

        binomialDistribution(n, p);
        k = 275;
        System.out.println("\nEvent = " + k);
        BigDecimal pVal = pValuesBinomial(n, p, k);
        System.out.println("One tail p-value of " + k
                + " = " + pVal);
        System.out.println("Two tails p-value of " + k
                + " = " + pVal.multiply(TWO));
    }

}//End of Program E305 PValBiBig
```

306, page 325. Changes are correlated all along the whole code. One might bold to say that correlated changes imply an extension or maybe a change of function. Correlated changes are difficult to produce by random mutation: in the case of human developers, one commits a lot of bugs before a new functional unit could be produced and in nature one expects a lot of imperfections that are expected to leave their tracks during many, many generations. Because imperfections are absent from the fossil record, it is obvious for every Java developer that the Evolutionary Theory is false. Nevertheless, this appreciation is not a dogma in our community, rather it is fuel to work hard in order to produce material that could be acceptable for those that love to go beyond any subjective feeling. This is an extremely daunting enterprise: get prepared to

work for decades.

307, page 325. Type double functions well for n less than 175 (using 64 bits). The Ramanujan approximation is implemented as follows. It produces results that are exact up to four decimal ciphers and for n as large as 1000. The approximation is better the larger is n.

```
//Program E307 PValBiRa
//Outputs the p-value of
//the  binomial distribution
//for  n coins (low values),
//probability of head p,
//and input k.
//We use the Ramanujan approximation for the factorial.

package ejvol5v2p;

public class PValBiRa {

    private static final double COEFFICIENTSBINOMIAL[]
            = new double[5001];
    private static final double LCOEFFICIENTSBINOMIAL[]
            = new double[5001];
    private static final double TERMSBINOMIAL[]
            = new double[5001];
    private static final double LTERMSBINOMIAL[]
            = new double[5001];

    private static int n;

    private static double p;

    private static int k;

    private static final double PI = 3.14159265358979323846;

    private static final boolean PRINT = false;

    //Prints data in vector
    private static void printVector(double Vect[], int n) {
        for (int i = 0; i <= n; i++) {
            System.out.println(Vect[i]);
        }
    }
```

```
//Sums data in vector with real numbers
private static double sumVector(double Vect[], int n) {
    double sum = 0;
    for (int i = 0; i <= n; i++) {
        sum = sum + Vect[i];
    }
    return sum;
}

//Returns numb!
private static double logFactorial(double n) {
    if (n == 0) {
        return 0;
    } else {
        double l = n * Math.log(n) - n
                + Math.log(n * (1 + 4 * n * (1 + 2 * n)))
                    / 6 + Math.log(PI) / 2;
        return l;
    }
}

//The  distribution  of the binomial distribution
// for  n coins
//probability of head p,
//event k
private static void binomialDistribution(int n,
        double p) {
    double logBiCoeff;
    for (int r = 0; r <= n; r++) {
        logBiCoeff = logFactorial(n)
                - logFactorial(r) - logFactorial(n - r);
        if(PRINT) System.out.println(logBiCoeff);
        LCOEFFICIENTSBINOMIAL[r] = logBiCoeff;
        COEFFICIENTSBINOMIAL[r] = Math.exp(logBiCoeff);
    }


    for (int r = 0; r < n + 1; r++) {

        LTERMSBINOMIAL[r] = LCOEFFICIENTSBINOMIAL[r]
                + r * Math.log(p) + (n - r) * Math.log(1 - p);
        TERMSBINOMIAL[r] = Math.exp(LTERMSBINOMIAL[r]);
    }
    sumVector(TERMSBINOMIAL, n);
    if(PRINT)
    {
```

```
            System.out.println("Coefficients ");
            printVector(COEFFICIENTSBINOMIAL, n);
            System.out.println("Sum = "
                    + sumVector(COEFFICIENTSBINOMIAL, n));
            System.out.println("Probabilities for 0,1,2...");
            System.out.println("Paste these data to Excel, "
                    + " OpenOffice or Gnumeric and \nget a bell.");
            printVector(TERMSBINOMIAL, n);

            System.out.println("Sum = "
                    + sumVector(TERMSBINOMIAL, n));
        }
    }

    //The  p-value of the binomial distribution
    // for  n repeats
    //probability of head p,
    //event k
    private static double pValuesBinomial(int n,
            double p, int k) {
        double mean = n * p;
        double sum = 0;
        if (k >= mean) {
            for (int r = k+1; r < n; r++) {
                sum = sum + TERMSBINOMIAL[r];
            }
        } else {
            for (int r = 0; r < k; r++) {
                sum = sum + TERMSBINOMIAL[r];
            }
        }
        return sum;
    }


    public static void main(String[] args) {
        n = 1000;
        p = 0.5;
        System.out.println("Binimial distribution");
        System.out.println("n = " + n + " p = " + p);

        binomialDistribution(n, p);
        k = 2;
        System.out.println("\nEvent = " + k);
        double pVal = pValuesBinomial(n, p, k);
        System.out.println("One tail p-value of " + k
```

```
                + " = " + pVal);
        System.out.println("Two tails p-value of " + k
                + " = " + 2 * pVal);
        k = n-k;
        System.out.println("\nEvent = " + k);
        pVal = pValuesBinomial(n, p, k);
        System.out.println("One tail p-value of " + k
                + " = " + pVal);
        System.out.println("Two tails p-value of " + k
                + " = " + 2 * pVal);
    }

}//End of Program E307 PValBiRa
```

310, page 326. p-value for a normal distribution.

```
//Program E310 PValNormal
//Reports the p value for a normal distribution.
//Combines Simpson's rule
//with a change of scale.

package ejvol5v2p;

public class PValNormal {

    private static final double PI = 3.14159265358979323846;
    private static double z;
    private static double mean;
    private static double deviation;
    private static double event;

    //****************Style: reuse*************
    //Returns the transformed density function of the standardized
    //normal distribution.
    private static double zFunction(double t) {
        z = t / (1 - t * t);
        double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2)
                * (1 + t * t) / ((1 - t * t) * (1 - t * t));
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
```

```
    private static double f(double z) {
        double f = zFunction(z);
        return f;
    }

    //Returns the integral under the z-density function
    //in within 0 and z. Precision=1/N.
    //Simpson's rule adapted for reuse.
    private static double zHalfBodySimpson(double z, long N) {

        double h = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * h);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * h);
        }
        sum = sum + f((2 * m) * h);
        sum = (h / 3) * sum;
        return sum;
    }

    //One tail p-value is reported for the Z distribution
    //and normalized event z > 0
    private static double work(double z) {
        z = (Math.sqrt(1 + 4 * z * z) - 1) / (2 * z);
        int N;
        double answer;
        int i = 3;
        N = (int) Math.pow(10, i);
        answer = 0.5 - zHalfBodySimpson(z, N);
        return answer;
    }

    public static void main(String[] args) {
        mean = 5;
        deviation = 0.7;
        event = 6.4;
        System.out.println("Normal distribution with mean = "
                + mean + " deviation = " + deviation
                + "\n Event = " + event);
        double zz = (event - mean) / deviation;
        System.out.println("z = " + zz);
        double pVal = 0;
```

```
        if (zz == 0) {
            pVal = 0.5;
        }
        //Upper tail
        if (zz > 0) {
            pVal = work(zz);
        }
        //Lower tail
        if (zz < 0) {
            pVal = work(-zz);
        }
        System.out.println("One tail p-value for "
                + event + " is " + pVal);
        System.out.println("Two tails p-value for "
                + event + " is " + 2 * pVal);
    }
}//End of Program E310 PValNormal
```

312, page 326. p-values for a t distribution.

```
//Program E312 PValT
//Reports the p-value for a t-test
//Combines Simpson's rule
//with a change of scale.

package ejvol5v2p;

public class PValT {

    private static final double PI = 3.14159265358979323846;
    private static double mean;
    private static int sampleSize;
    private static double sampleDeviation;
    private static double event;
    private static double nu;

    //Returns numb!
    private static double factorial(long numb) {
        if (numb <= 1) {
            return 1;
        } else {
            return numb * factorial(numb - 1);
```

```
        }
    }

//Returns the double factorial
    private static double doubleFact(double x) {
        if (x <= 0) {
            return 1;
        } else {
            return x * doubleFact(x - 2);
        }
    }

//Returns the Gamma function
    private static double Gamma(double x) {
        int xRounded = (int) x;
        //System.out.println( x + "   " + xRounded);
        if (xRounded == x) {
            return factorial(xRounded - 1);
        } else {
            return Math.sqrt(PI) * doubleFact(2 * xRounded - 1)
                    / Math.pow(2, xRounded);
        }
    }

    //****************Style: reuse*************
//Returns the transformed density function of the
//t distribution with nu d.f.
//nu is a globally defined variable.
    private static double tFunction(double t) {
        double z = t / (1 - t * t);
        double f = (Gamma((nu + 1) / 2)
                / (Math.sqrt(PI * nu) * Gamma(nu / 2)))
                * Math.pow(1 + z * z / nu, -(nu + 1) / 2) * (1 + t * t) / (
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double z) {
        double f = tFunction(z);
        return f;
    }

    //Returns the integral under the t-density function
    //in within 0 and t. Precision=1/N.
```

```
//Simpson's rule adapted for reuse.
private static double HalfBodySimpson(double z, long N) {

    double h = z / (2 * N);
    long m = N;
    double sum = f(0);
    for (int j = 1; j <= m; j++) {
        sum = sum + 4 * f((2 * j - 1) * h);
    }
    for (int j = 1; j < m; j++) {
        sum = sum + 2 * f((2 * j) * h);
    }
    sum = sum + f((2 * m) * h);
    sum = (h / 3) * sum;
    return sum;
}

//One tail p-value is reported for the t distribution
//and tValue > 0
private static double work(double z) {
    double limSup = (Math.sqrt(1 + 4 * z * z) - 1) / (2 * z);
    int N;
    double answer;
    int i = 3;
    N = (int) Math.pow(10, i);
    answer = 0.5 - HalfBodySimpson(limSup, N);
    return answer;
}

public static void main(String[] args) {
    mean = 5;
    sampleSize = 80;
    sampleDeviation = 0.7;
    event = 6.4;
    nu = sampleSize - 1;
    System.out.println("Normal distribution with mean = "
            + mean + ". \nSample size = "
            + sampleSize + "\nSample deviation = "
            + sampleDeviation
            + "\nEvent = " + event
            + "\nDegrees of freedom  " + nu);
    double tValue = (event - mean) / sampleDeviation;

    System.out.println("t of the event = " + tValue);
    double pVal = 0;
    nu = sampleSize - 1;
```

```
        if (tValue == 0) {
            pVal = 0.5;
        }
        //Upper tail
        if (tValue > 0) {
            pVal = work(tValue);
        }
        //Lower tail
        if (tValue < 0) {
            pVal = work(-tValue);
        }
        System.out.println("One tail p-value for t-value "
                + tValue + " is \n" + pVal);
        System.out.println("Two tails p-value for t-value "
                + tValue + " is \n" + 2 * pVal);
    }
}//End of Program E312 PValT
```

315, page 327. p-values for the χ^2 distribution. Results differ from those of Gnumeric in the 9-th decimal place.

```
//Program E315 PValChi
//Reports the p-value for a chi-2-test
//Combines Simpson's rule
//with a change of scale.

package ejvol5v2p;

public class PValChi {

    private static final double PI = 3.14159265358979323846;
    private static double sampleSize;
    private static double sampleVar;
    private static double sigmaPow2;
    private static double nu;

    //Returns numb!
    private static double factorial(long numb) {
        if (numb <= 1) {
            return 1;
        } else {
            return numb * factorial(numb - 1);
        }
```

```
}

//Returns the double factorial
private static double doubleFact(double x) {
    if (x <= 0) {
        return 1;
    } else {
        return x * doubleFact(x - 2);
    }
}

//Returns the Gamma function
private static double Gamma(double x) {
    int xRounded = (int) x;
    //System.out.println( x + "   " + xRounded);
    if (xRounded == x) {
        return factorial(xRounded - 1);
    } else {
        return Math.sqrt(PI) * doubleFact(2 * xRounded - 1)
                / Math.pow(2, xRounded);
    }
}

//****************Style: reuse*************
//Returns the transformed density function of the
//chi-square distribution with nu d.f.
//nu is a globally defined variable.
private static double chiFunction(double t) {
    double z = t / (1 - t * t);
    double f = (1 / (Math.pow(2, nu / 2) * Gamma(nu / 2)))
            * Math.exp(-z / 2) * Math.pow(z, (nu - 2) / 2)
            * (1 + t * t) / ((1 - t * t) * (1 - t * t));
    return f;
}

//To use this method for another distribution,
//instead of  zFunction(z), write and make a call for your
//new distribution.
private static double f(double z) {
    double f = chiFunction(z);
    return f;
}

//Returns the integral under the chi-density function
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
```

```
    private static double HalfBodySimpson(double z, long N) {

        double h = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * h);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * h);
        }
        sum = sum + f((2 * m) * h);
        sum = (h / 3) * sum;
        return sum;
    }

    //One tail p-value is reported for the chi distribution
    //and chiValue > 0
    private static double work(double z) {
        double limSup = (Math.sqrt(1 + 4 * z * z) - 1) / (2 * z);
        int N;
        double answer;
        int i = 3;
        N = (int) Math.pow(10, i);
        answer = 1 - HalfBodySimpson(limSup, N);
        return answer;
    }

    public static void main(String[] args) {
        /*
sampleSize = 80;
        sampleDeviation = 0.7;
sigma  = 6.4;
nu= sampleSize-1;
System.out.println("Sample size = "
                    + sampleSize + "\nSample deviation = "
                    + sampleDeviation
        + "\nsigma = " + sigma
        + "\nDegrees of freedom  " + nu);
double chiValue = nu * sampleDeviation*sampleDeviation/
                  (sigma*sigma);
         */

        sampleSize = 9;
        sampleVar = 4;
        sigmaPow2 = 5;
```

```
        nu = sampleSize - 1;
        System.out.println("Sample size = "
                + sampleSize + "\nSample variance = "
                + sampleVar
                + "\nPop Var = " + sigmaPow2
                + "\nDegrees of freedom  " + nu);
        double chiValue = nu * sampleVar / sigmaPow2;

        System.out.println("chi-value of the event = " + chiValue);
        double pVal = work(chiValue);
        //Lower tail

        System.out.println("One tail p-value for chi-value "
                + chiValue + " and " + nu
                + " df is \n" + pVal);
        System.out.println("Two tails p-value for chi-value "
                + chiValue + " and " + nu
                + " df is \n" + 2 * pVal);
        if (2 * pVal > 1) {
            System.out.println("chi-value is lower than the median.");
        }

        chiValue = 20.8;
        nu = 8;
        if (chiValue > 0) {
            pVal = work(chiValue);
        }
        //Lower tail
        if (chiValue < 0) {
            pVal = work(-chiValue);
        }
        System.out.println("\nOne tail p-value for chi-value "
                + chiValue + " is \n" + pVal);
        System.out.println("Two tails p-value for chi-value "
                + chiValue + " is \n" + 2 * pVal);

    }
}//End of Program E315 PValChi
```

318, page 327. The next code use previous programs to calculate p-values for an F distribution.

```
//Program E318 PValF
//Reports the p-value for a F-test
```

```
//Uses the integral representation
//of the beta function plus Simpson's rule

package ejvol5v2p;

public class PValF {

    private static double nu1, nu2;

    //Returns the beta   function  with   d.f. =
    //a(num) and b(den)
    private static double g(double x, double a, double b) {
        double g = Math.pow(x, a - 1)
                * (Math.pow(1 - x, b - 1));
        return g;
    }

    //Returns the integral under the function f
    //in within 0 and u. Precision=1/N.
    //Simpson's rule adapted for reuse.
    //a and b are degrees of freedom
    private static double HalfBodySimpsong(double u,
            double a, double b, long N) {
        double h = u / (2 * N);
        long m = N;
        double sum = g(0, a, b);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * g((2 * j - 1) * h, a, b);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * g((2 * j) * h, a, b);
        }
        sum = sum + g((2 * m) * h, a, b);
        sum = (h / 3) * sum;
        return sum;
    }

    //Returns the  beta  function in integral form
    //with a and b d.f.
    private static double beta(double a, double b) {
        double answer;
        int i = 3;
        int N = (int) Math.pow(10, i);
        double u = 1;
        answer = HalfBodySimpsong(u, a, b, N);
        return answer;
```

```
}

//****************F density function*************
//Returns the transformed density function of the
//F distribution with nu1 and nu2 d.f.
private static double FFunction(double t) {
    double z = t / (1 - t * t);
    double f = Math.pow(z, nu1 / 2 - 1)
            * Math.pow(nu2 + nu1 * z, -(nu1 + nu2) / 2)
            * ((1 + t * t) / ((1 - t * t) * (1 - t * t)));
    //System.out.println(f);
    return f;
}

//To use this method for another distribution,
//instead of  FFunction(z), write and make a call for your
//new distribution.
private static double f(double z) {
    double f = FFunction(z);
    return f;
}

//Returns the integral under the F-density function
//in within 0 and z. Precision=1/N.
//Simpson's rule adapted for reuse.
private static double HalfBodySimpson(double z, long N) {
    double h = z / (2 * N);
    long m = N;
    double sum = f(0);
    for (int j = 1; j <= m; j++) {
        sum = sum + 4 * f((2 * j - 1) * h);
    }
    for (int j = 1; j < m; j++) {
        sum = sum + 2 * f((2 * j) * h);
    }
    sum = sum + f((2 * m) * h);
    sum = (h / 3) * sum;
    double a = Math.pow(nu1, nu1 / 2);
    double b = Math.pow(nu2, nu2 / 2);
    double c = beta(nu1 / 2, nu2 / 2);
    double k = a * b / c;
    sum = k * sum;
    System.out.println("a = " + a
            + "\nb = " + b
            + "\nc = " + c
            + "\nk = " + k
```

```
                + " sum after k = " + sum);
        return sum;
    }

    //One tail p-value is reported for the F distribution
    //and FValue > 0
    private static double work(double z) {
        double limSup = (Math.sqrt(1 + 4 * z * z) - 1) / (2 * z);
        int N;
        double answer;
        int i = 3;
        N = (int) Math.pow(10, i);
        //The p-value is the integral from FValue to infinite.
        answer = 1 - HalfBodySimpson(limSup, N);
        return answer;
    }

    public static void main(String[] args) {
        double R = 1;
        double devNum = 8;
        double devDen = 7;
        int sampleSizeNum = 8;
        int sampleSizeDen = 7;
        nu1 = sampleSizeNum - 1;
        nu2 = sampleSizeDen - 1;
        double sampleVarNum = devNum * devNum;
        double sampleVarDen = devDen * devDen;

        double RExp = sampleVarNum / (sampleVarDen);
        if (RExp < 1) {
            RExp = 1 / RExp;
        }
        double FValue = RExp / R;
        System.out.println("F test:  ");
        System.out.println(
                "\nSample size numerator   = "
                + sampleSizeNum
                + "\nSample size denominator   = "
                + sampleSizeDen
                + "\nSample variance Num  = "
                + sampleVarNum
                + "\nSample variance Den  = "
                + sampleVarDen
                + "\nRExp  = " + RExp
                + "\nR = " + R
                + "\nF-value exp =   " + FValue);
```

```
        double pVal = work(FValue);
        System.out.println("One tail p-value for Fvalue "
                + FValue + " and "
                + "\ndf " + nu1 + ", "
                + nu2
                + " is \n" + pVal);
        System.out.println("Two tails p-value for Fvalue "
                + FValue + " and df" + nu1 + ", "
                + nu2
                + "   is \n" + 2 * pVal);
        if (2 * pVal > 1) {
            System.out.println("F-value is lower than the median");
        }

    }
}//End of  Program E318 PValF
```

320, page 328. The incomplete beta function is used to calculate p-values for an F-test.

```
//Program E320 PValFPro
//Reports the p-value for an F-test
//The code uses the incomplete beta function
//plus Simpson's rule to calculate integrals.

package ejvol5v2p;

public class PValFPro {

    private static double nu1, nu2;

    //Returns the beta   function
    //with df a and b
    private static double g(double x, double a, double b) {
        double g = Math.pow(x, a - 1)
                * (Math.pow(1 - x, b - 1));
        return g;
    }

    //Returns the integral under the function g
    //in within 0 and u. Precision=1/N.
    //Simpson's rule adapted for reuse.
    //a and b are degrees of freedom.
```

```
private static double HalfBodySimpson(double u,
        double a, double b, long N) {
    double h = u / (2 * N);
    long m = N;
    double sum = g(0, a, b);
    for (int j = 1; j <= m; j++) {
        sum = sum + 4 * g((2 * j - 1) * h, a, b);
    }
    for (int j = 1; j < m; j++) {
        sum = sum + 2 * g((2 * j) * h, a, b);
    }
    sum = sum + g((2 * m) * h, a, b);
    sum = (h / 3) * sum;
    return sum;
}

//Returns the incomplete beta  function in integral form
//with a and b d.f.
private static double incompleteBeta(double u,
        double a, double b) {
    double answer;
    int i = 3;
    int N = (int) Math.pow(10, i);
    answer = HalfBodySimpson(u, a, b, N);
    return answer;
}

//Returns the regularized incomplete beta function
//in integral form with a and b d.f.
private static double RIBeta(double u,
        double a, double b) {

    double answer, answer1, answer2;
    answer1 = incompleteBeta(u, a, b);
    answer2 = incompleteBeta(1, a, b);
//System.out.println( "beta(a,b) = " + answer2);
    answer = answer1 / answer2;
    return answer;
}

public static void main(String[] args) {
    double R = 1;
    double devNum = 8;
    double devDen = 7;
    int sampleSizeNum = 8;
    int sampleSizeDen = 7;
```

```
        nu1 = sampleSizeNum - 1;
        nu2 = sampleSizeDen - 1;
        double sampleVarNum = devNum * devNum;
        double sampleVarDen = devDen * devDen;

        double RExp = sampleVarNum / (sampleVarDen);
        if (RExp < 1) {
            RExp = 1 / RExp;
        }
        double FValue = RExp / R;
        System.out.println("F test:  ");
        double u = nu1 * FValue / (nu2 + nu1 * FValue);
        double a = RIBeta(u, nu1 / 2, nu2 / 2);
        double b = RIBeta(1 - u, nu2 / 2, nu1 / 2);
        double test = a + b;
        System.out.println("test for precission, "
                + "must output 1 exactly: \n" + test);
        double Error = test - 1;
        System.out.println("Error = " + Error);
        double pVal = 1 - a;
        System.out.println(
                "\nSample size numerator   = "
                + sampleSizeNum
                + "\nSample size denominator   = "
                + sampleSizeDen
                + "\nSample variance Num  = "
                + sampleVarNum
                + "\nSample variance Den  = "
                + sampleVarDen
                + "\nRExp  = " + RExp
                + "\nR = " + R
                + "\nF-value exp =   " + FValue);

        System.out.println("One tail p-value for Fvalue "
                + FValue + " \nand df " + nu1 + ", "
                + nu2
                + " is \n" + pVal);
        System.out.println("Two tails p-value for Fvalue "
                + FValue + " \nand df" + nu1 + ", "
                + nu2
                + "   is \n" + 2 * pVal);
        if (2 * pVal > 1) {
            System.out.println("F-value is lower than the median");
        }
    }
}//End of Program E320 PValFPro
```

324, page 328. p-values for a Pareto distribution. The mathematical solution is immediate:

If $V > m$, the p-value for one tail associated to V is

$$p = \int_V^\infty f(x)dx = \int_V^\infty \frac{rm^r}{x^{r+1}}dx = rm^r \int_V^\infty \frac{1}{x^{r+1}}dx = rm^r \int_V^\infty x^{-r-1}dx$$

$$= rm^r(-x^{-r}/r)|_V^\infty = -\frac{m^r}{x^r}|_V^\infty = 0 + \frac{m^r}{V^r} = (\frac{m}{V})^r$$

328, page 329. Critical values for the binomial distribution.

```
//Program E328 CritBin
//Outputs critical values of
//the  binomial distribution
//for  n coins,
//probability of head p,
//significance level alpha.

package ejvol5v2p;

public class CritBin {

    private static final double COEFFICIENTSBINOMIAL[]
            = new double[20];
    private static final double TERMSBINOMIAL[]
            = new double[20];

    //Prints data in vector
    private static void printVector(double Vect[], int n) {
        for (int i = 0; i <= n; i++) {
            System.out.println(Vect[i]);
        }
    }

    //Sums data in vector with real numbers
    private static double sumVector(double Vect[], int n) {
        double sum = 0;
        for (int i = 0; i <= n; i++) {
            sum = sum + Vect[i];
        }
        return sum;
    }
```

```
    //Returns numb!
    private static long factorial(long numb) {
        if (numb <= 1) {
            return 1;
        } else {
            return numb * factorial(numb - 1);
        }
    }

    //The  binomial distribution
    // for  n repeats
    //probability of head p,
    //significance level alpha
    private static void binomialDistribution(int n,
            double p) {
        long biCoeff ;
        for (int r = 0; r < n + 1; r++) {
            biCoeff = factorial(n)
                    / (factorial(r) * factorial(n - r));
            COEFFICIENTSBINOMIAL[r] = biCoeff;
        }
        System.out.println("Coefficients ");
        printVector(COEFFICIENTSBINOMIAL, n);
        System.out.println("Sum = "
                + sumVector(COEFFICIENTSBINOMIAL, n));

        for (int r = 0; r < n + 1; r++) {

            TERMSBINOMIAL[r] = COEFFICIENTSBINOMIAL[r]
                    * Math.pow(p, r) * Math.pow(1 - p, n - r);
        }
        System.out.println("Probabilities for 0,1,2...");
        System.out.println("Paste this vector to Excel"
                + " and make a graphic");
        printVector(TERMSBINOMIAL, n);
        sumVector(TERMSBINOMIAL, n);
        System.out.println("Sum = "
                + sumVector(TERMSBINOMIAL, n));
    }

//Critical value for significance level alpha, one tail
    private static void critical1(int n,
            double p,
            double alpha
    ) {
```

```
        int numbTails = 1;
        System.out.println("\nLower tail");
        double cumulative = 0;
        double target = alpha;
        int i = -1;
        while (cumulative < target) {
            i = i + 1;
            cumulative = cumulative + TERMSBINOMIAL[i];
            System.out.println(i + "  cumulative = " + cumulative);
        }
        System.out.println("To cumulate in the lower tail "
                + target
                + ", \nwe need the first " + i + " Terms.");
        System.out.println("The critical value for alpha =  "
                + alpha
                + ", numb of tails = " + numbTails
                + "\nin the lower tail is "
                + i + ". All terms below it are extreme.\n");
        System.out.println("Upper tail");
        i = n + 1;
        cumulative = 0;
        while (cumulative < target) {
            i = i - 1;
            cumulative = cumulative + TERMSBINOMIAL[i];
            System.out.println(i + "  cumulative = " + cumulative);
        }

        System.out.println("To cumulate in the upper tail "
                + target
                + ", \nwe need term  " + i + " and above.");
        System.out.println("The critical value for alpha =  "
                + alpha
                + ", numb of tails = " + numbTails
                + "\nin the upper tail is "
                + i + ". All terms above it are extreme.\n");
    }

//Critical value for significance level alpha, two tails
    private static void critical2(int n,
            double p,
            double alpha) {

        critical1(n, p, alpha / 2);
    }

    public static void main(String[] args) {
```

```
        int n = 8;//number of coins
        double p = 0.5; //probability of head
        double alpha = 0.05;//number of repeats
        int numbTails = 1;
        System.out.println("Binomial distribution: \n"
                + " n = " + n + ", p = " + p + ", "
                + "significance level = alpha = " + alpha);
        binomialDistribution(n, p);
        System.out.println("\nCRITICAL VALUES");

        System.out.println("\nONE TAIL, alpha = " + alpha);
        System.out.println("\nTerms and cumulative values");
        critical1(n, p, alpha);
        System.out.println("\nTWO TAILS, alpha = " + alpha);
        System.out.println("\nTerms and cumulative values");
        critical2(n, p, alpha);
    }

}//End of Program E328 CritBin
```

339, page 343. Critical values for a standardized normal distribution (professional implementation).

```
//Program E339 CritNormal4
//Reports   critical values
//for a standardized normal distribution
//and a given significance level.
//Combines Simpson's rule
//with a change of scale.
//An initial approximation is improved
//by analytical hunting.

package ejvol5v2p;

public class CritNormal4 {

    private static final double PI = 3.14159265358979323846;
    private static double z;
    //Significance level
    private static double alpha;
    private static double target;
    private static double greatSum;
    private static double Error;
    private static double uCritic;
```

```
private static double zCritic;
private static double jCritic;
//Right border or the critical interval
private static double rBorder;
private static double h;
private static int N;

//****************Style: reuse*************
//Returns the transformed density function of
//the standardized normal distribution.
private static double zFunction(double t) {
    z = t / (1 - t * t);
    double f = 1 / Math.pow(2 * PI, 0.5) * Math.exp(-z * z / 2)
            * (1 + t * t) / ((1 - t * t) * (1 - t * t));
    return f;
}

//To use this method for another distribution,
//instead of  zFunction(z), write and make a call for your
//new distribution.
private static double f(double x) {
    double f = zFunction(x);
    return f;
}

//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
private static double upperU(double u,
        long N,
        double target) {
    h = u / N;
    double r = h / 6;
    double sum = 0.5;
    jCritic = 0;
    for (int j = 0; sum < target; j++) {
        sum = sum
                + r * (f(j * h) + 4 * f((j + 0.5) * h) + f((j + 1) * h))
        jCritic = j + 1;
    }
    jCritic = jCritic - 1;
    greatSum = sum
            - r * (f(jCritic * h) + 4 * f((jCritic + 0.5) * h)
            + f((jCritic + 1) * h));
```

```
    double uAprox = (jCritic) * h;
    return uAprox;
}

//Returns the transformed value z = u/(1-u*u)
private static double z(double u) {
    double zz = u / (1 - u * u);
    return zz;
}

//Reports the upper z-critical for
//Significance level = alpha.
private static double zCrit(double alpha) {
    double u = 1;
    int i = 3;
    N = (int) Math.pow(10, i);
    uCritic = upperU(u, N, alpha);
    Error = target - greatSum;
    double zzCritic = z(uCritic);
    return zzCritic;
}

//*****Analytical hunting*******
//Returns the density function of the Z distribution
private static double ff(double z) {
    double f = Math.exp(-z * z / 2) / Math.pow(2 * PI, 0.5);
    return f;
}

// Returns F(b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
private static double F(double b) {
    double a = zCritic;
    double hh = (b - a) / 6;

    double Fb = hh * (ff(a) + 4 * ff((a + b) / 2)
            + ff(b)) - Error;
    return Fb;
}

//Returns the  ratio  F(gi)/ ff(gi)
private static double ratio(double gi) {
    double ratio = F(gi) / ff(gi);
    return ratio;
}
```

```
    //Critical value for the upper tail.
    //A professional  approximation is found by
    //analytical hunting
    private static double hunting(double zCritic) {
        double gSubi = rBorder;
        double gSubiPlus = 0;
        int iterations = 0;
        int maxIterations = 30;
        boolean more = true;
        while (more) {
            gSubiPlus = gSubi - ratio(gSubi);
            iterations = iterations + 1;
            if (iterations >= maxIterations) {
                more = false;
            }
            if (gSubiPlus == gSubi) {
                more = false;
            }
            gSubi = gSubiPlus;
        }
        return gSubiPlus;
    }


    //Critical value for significance level alpha, two tails
    private static double critical(
            double alpha) {
        target = 1 - alpha;
        zCritic = zCrit(target);
        double ur = (jCritic + 1) * h;
        rBorder = z(ur);
        zCritic = hunting(zCritic);
        return zCritic;
    }

    public static void main(String[] args) {
        alpha = 0.05;
        System.out.println("Critical values for one tail "
                + "\nof the standardized normal "
                + "distribution.  "
                + "\nAlpha = Significance level = "
                + alpha + ".");

        System.out.println("z-critic  = +/-"
                + critical(alpha));
        System.out.println("\nCritical values for two tails "
```

```
                + "\nof the standardized normal "
                + "distribution."
                + "\nAlpha = Significance level = "
                + alpha + ".");
        double alphaHalf = alpha / 2;

        System.out.println("z-critic  = +/-"
                + critical(alphaHalf));

    }
}//End of Program E339 CritNormal4
```

340, page 343. Critical values for the t-distribution (professional implementation).

```
//Program E340 CritT
//Reports   critical values
//for a t-distribution
//and a given significance level.
//Combines Simpson's rule
//with a change of scale.
//An initial approximation is improved
//by analytical hunting.

package ejvol5v2p;


public class CritT {

    private static final double PI = 3.14159265358979323846;
    //Significance level
    private static double alpha;
    private static double target;
    private static double greatSum;
    private static double Error;
    private static double uCritic;
    private static double vCritic;
    private static double jCritic;
    //Right border or the critical interval
    private static double rBorder;
    private static double h;
    private static int N;
    private static double nu;
```

```
    //****************Style: reuse*************
    //Returns numb!
    private static double factorial(long numb) {
        if (numb <= 1) {
            return 1;
        } else {
            return numb * factorial(numb - 1);
        }
    }

//Returns the double factorial
    private static double doubleFact(double x) {
        if (x <= 0) {
            return 1;
        } else {
            return x * doubleFact(x - 2);
        }
    }

//Returns the Gamma function
    private static double Gamma(double x) {
        int xRounded = (int) x;
        //System.out.println( x + "   " + xRounded);
        if (xRounded == x) {
            return factorial(xRounded - 1);
        } else {
            return Math.sqrt(PI) * doubleFact(2 * xRounded - 1)
                    / Math.pow(2, xRounded);
        }
    }

    //****************Style: reuse*************
//Returns the transformed density function of the
//t distribution with nu d.f.
//nu is a globally defined variable.
    private static double tFunction(double t) {
        double z = t / (1 - t * t);
        double f = (Gamma((nu + 1) / 2)
                / (Math.sqrt(PI * nu) * Gamma(nu / 2)))
                * Math.pow(1 + z * z / nu, -(nu + 1) / 2)
                * (1 + t * t) / ((1 - t * t) * (1 - t * t));
        return f;
    }

    //To use this method for another distribution,
```

```
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double z) {
        double f = tFunction(z);
        return f;
    }

    //Reports the transformed upper u-critical for
    //the given distribution.
    //Significance level = alpha.
    //Precision=1/N.
    //Simpson's rule adapted for reuse.
    private static double upperU(double u,
            long N,
            double target) {
        h = u / N;
        double r = h / 6;
        double sum = 0.5;
        jCritic = 0;
        for (int j = 0; sum < target; j++) {
            sum = sum
                    + r * (f(j * h) + 4 * f((j + 0.5) * h) + f((j + 1) * h));
            jCritic = j + 1;
        }
        jCritic = jCritic - 1;
        greatSum = sum
                - r * (f(jCritic * h) + 4 * f((jCritic + 0.5) * h)
                + f((jCritic + 1) * h));

        double uAprox = (jCritic) * h;
        return uAprox;
    }

//Returns the transformed value z = u/(1-u*u)
    private static double z(double u) {
        double z = u / (1 - u * u);
        return z;
    }

    //Reports the upper z-critical for
    //Significance level = alpha.
    private static double vCrit(double target) {
        double u = 1;
        int i = 3;
        N = (int) Math.pow(10, i);
        uCritic = upperU(u, N, target);
```

```
        Error = target - greatSum;
        double zCritic = z(uCritic);
        return zCritic;
    }

    //*****Analytical hunting*******
    //Returns the density function of the t-distribution
    private static double ff(double z) {
        double f = (Gamma((nu + 1) / 2)
                / (Math.sqrt(PI * nu) * Gamma(nu / 2)))
                * Math.pow(1 + z * z / nu, -(nu + 1) / 2);
        return f;
    }

    //F(b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
    private static double F(double b) {
        double a = vCritic;
        double hh = (b - a) / 6;

        double Fb = hh * (ff(a) + 4 * ff((a + b) / 2)
                + ff(b)) - Error;
        return Fb;
    }

    //Returns the  ratio  F(gi)/ ff(gi)
    private static double ratio(double gi) {
        double ratio = F(gi) / ff(gi);
        return ratio;
    }

    //Critical value for the upper tail.
    //A professional  approximation is found by
    //analytical hunting
    private static double hunting() {
        double gSubi = rBorder;
        double gSubiPlus = 0;
        int iterations = 0;
        int maxIterations = 30;
        boolean more = true;
        while (more) {
            gSubiPlus = gSubi - ratio(gSubi);
            iterations = iterations + 1;
            if (iterations >= maxIterations) {
                more = false;
            }
            if (gSubiPlus == gSubi) {
```

```
                more = false;
            }
            gSubi = gSubiPlus;
            /*System.out.println("iteration = " + iterations +
              "New crit-value = " + gSubi);*/
        }
        return gSubiPlus;
    }

//Critical value for significance level alpha, two tails
    private static double critical(
            double alpha) {
        target = 1 - alpha;
        vCritic = vCrit(target);
        //System.out.println("First approx = " + vCritic);
        double ur = (jCritic + 1) * h;
        rBorder = z(ur);
        vCritic = hunting();
        //System.out.println("Hunting approx = " + vCritic);
        return vCritic;
    }

    //************Test********
    //The cumulative function evaluated at z
    private static double cum(double z, long N) {
        double hh = z / (2 * N);
        long m = N;
        double sum = f(0);
        for (int j = 1; j <= m; j++) {
            sum = sum + 4 * f((2 * j - 1) * hh);
        }
        for (int j = 1; j < m; j++) {
            sum = sum + 2 * f((2 * j) * hh);
        }
        sum = sum + f((2 * m) * hh);
        sum = (hh / 3) * sum;
        return sum + 0.5;
    }

    //The cumulative function evaluated at
    //vCritic must be equal to the target = 1-alpha.
    private static void test(double z) {
        System.out.println(
                "\nArea under the t-bell in within -infty and " + z);
        z = (Math.sqrt(1 + 4 * z * z) - 1) / (2 * z);
        int NN;
```

```
        double answer;
        NN = (int) Math.pow(10, 3);
        answer = cum(z, NN);
        System.out.println(answer);
        System.out.println("Expected = " + target);
        double error = target - answer;
        System.out.println("Forth and back Error = " + error);
    }

    public static void main(String[] args) {
        alpha = 0.05;
        nu = 7;
        System.out.println("Critical values for one tail "
                + "\nof the t-distribution with  "
                + nu + " degrees of freedom"
                + "\nAlpha = Significance level = "
                + alpha + ".");

        System.out.println("critical value  = +/-"
                + critical(alpha));
        System.out.println("\nCritical values for two tails "
                + "\nof the t-distribution with  "
                + nu + " degrees of freedom"
                + "\nAlpha = Significance level = "
                + alpha + ".");
        double alphaHalf = alpha / 2;

        System.out.println("Critical value  = +/-"
                + critical(alphaHalf));
        System.out.println("\nTEST");
        //Test to measure the accurateness of the approximation
        test(vCritic);
    }
}//End of Program E340 CritT
```

341, page 343. Critical values for the chi-2-distribution (professional implementation).

```
package ejvol5v2p;


//Program E341 CritChi2
//Reports   critical values
//for a chi-square-distribution
```

```
//and a given significance level.
//Combines Simpson's rule
//with a change of scale.
//An initial approximation is improved
//by analytical hunting.


public class CritChi2 {

    private static final double PI = 3.14159265358979323846;
    //Significance level
    private static double alpha;
    private static double target;
    private static double greatSum;
    private static double Error;
    private static double uCritic;
    private static double vCritic;
    private static double jCritic;
    //Right border or the critical interval
    private static double rBorder;
    private static double h;
    private static int N;
    private static double nu;

    //****************Style: reuse*************
    //Returns numb!
    private static double factorial(long numb) {
        if (numb <= 1) {
            return 1;
        } else {
            return numb * factorial(numb - 1);
        }
    }

    //Returns the double factorial
    private static double doubleFact(double x) {
        if (x <= 0) {
            return 1;
        } else {
            return x * doubleFact(x - 2);
        }
    }

    //Returns the Gamma function
    private static double Gamma(double x) {
        int xRounded = (int) x;
```

```
        //System.out.println( x + "   " + xRounded);
        if (xRounded == x) {
            return factorial(xRounded - 1);
        } else {
            return Math.sqrt(PI) * doubleFact(2 * xRounded - 1)
                    / Math.pow(2, xRounded);
        }
    }

    //****************Style: reuse*************
    //Returns the transformed density function of the
    //chi-square distribution with nu d.f.
    //nu is a globally defined variable.
    private static double chiFunction(double t) {
        double z = t / (1 - t * t);
        double f = (1 / (Math.pow(2, nu / 2) * Gamma(nu / 2)))
                * Math.exp(-z / 2) * Math.pow(z, (nu - 2) / 2)
                * (1 + t * t) / ((1 - t * t) * (1 - t * t));
        return f;
    }

    //To use this method for another distribution,
    //instead of  zFunction(z), write and make a call for your
    //new distribution.
    private static double f(double z) {
        double f = chiFunction(z);
        return f;
    }

    //Reports the transformed upper u-critical for
    //the given distribution.
    //Significance level = alpha.
    //Precision=1/N.
    //Simpson's rule adapted for reuse.
    private static double upperU(double u,
            long N,
            double target) {
        h = u / N;
        double r = h / 6;
        double sum = 0;
        jCritic = 0;
        for (int j = 0; sum < target; j++) {
            sum = sum
                    + r * (f(j * h) + 4 * f((j + 0.5) * h) + f((j + 1) * h))
            jCritic = j + 1;
        }
```

```
    jCritic = jCritic - 1;
    greatSum = sum
            - r * (f(jCritic * h) + 4 * f((jCritic + 0.5) * h)
            + f((jCritic + 1) * h));

    double uAprox = (jCritic) * h;
    return uAprox;
}

//Returns the transformed value z = u/(1-u*u)
private static double z(double u) {
    double z = u / (1 - u * u);
    return z;
}

//Reports the upper z-critical for
//Significance level = alpha.
private static double vCrit(double alpha) {
    double u = 1;
    int i = 4;
    N = (int) Math.pow(10, i);
    uCritic = upperU(u, N, target);
    Error = target - greatSum;
    double zCritic = z(uCritic);
    return zCritic;
}

//*****Analytical hunting*******
//Returns the density function of the chi2-distribution
private static double ff(double z) {
    double f = (1 / (Math.pow(2, nu / 2) * Gamma(nu / 2)))
            * Math.exp(-z / 2) * Math.pow(z, (nu - 2) / 2);
    return f;
}

//F(b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
private static double F(double b) {
    double a = vCritic;
    double hh = (b - a) / 6;

    double Fb = hh * (ff(a) + 4 * ff((a + b) / 2)
            + ff(b)) - Error;
    return Fb;
}

//Returns the  ratio  F(gi)/ ff(gi)
```

```
private static double ratio(double gi) {
    double ratio = F(gi) / ff(gi);
    return ratio;
}

//Critical value for the upper tail.
//A professional  approximation is found by
//analytical hunting
private static double hunting() {
    double gSubi = rBorder;
    double gSubiPlus = 0;
    int iterations = 0;
    int maxIterations = 30;
    boolean more = true;
    while (more) {
        gSubiPlus = gSubi - ratio(gSubi);
        iterations = iterations + 1;
        if (iterations >= maxIterations) {
            more = false;
        }
        if (gSubiPlus == gSubi) {
            more = false;
        }
        gSubi = gSubiPlus;
        /*System.out.println("iteration = " + iterations +
          "New crit-value = " + gSubi);*/
    }
    return gSubiPlus;
}

//Critical value for significance level alpha, two tails
private static double critical(
        double alpha) {
    target = 1 - alpha;
    vCritic = vCrit(target);
    //System.out.println("First approx = " + vCritic);
    double ur = (jCritic + 1) * h;
    rBorder = z(ur);
    vCritic = hunting();
    //System.out.println("Hunting approx = " + vCritic);
    return vCritic;
}

//************Test********
//The cumulative function evaluated at z
private static double cum(double z, long N) {
```

```
    double hh = z / (2 * N);
    long m = N;
    double sum = f(0);
    for (int j = 1; j <= m; j++) {
        sum = sum + 4 * f((2 * j - 1) * hh);
    }
    for (int j = 1; j < m; j++) {
        sum = sum + 2 * f((2 * j) * hh);
    }
    sum = sum + f((2 * m) * hh);
    sum = (hh / 3) * sum;
    return sum;
}

//The cumulative function evaluated at
//vCritic must be equal to the target = 1-alpha.
private static void test(double z) {
    System.out.println(
            "\nArea under the chi2 density function "
            + "in within 0 and \n" + z);
    z = (Math.sqrt(1 + 4 * z * z) - 1) / (2 * z);
    int NN;
    double answer;
    NN = (int) Math.pow(10, 4);
    answer = cum(z, NN);
    System.out.println(answer);
    System.out.println("Expected = " + target);
    double error = target - answer;
    System.out.println("Forth and back Error = " + error);
}

public static void main(String[] args) {
    alpha = 0.05;
    nu = 7;
    System.out.println("Critical values for one tail "
            + "\nof the chi-square- "
            + "distribution.  "
            + "\nAlpha = Significance level = "
            + alpha + ".");

    System.out.println("Lower critical value  = "
            + critical(1 - alpha));
    System.out.println("Upper critical value  = "
            + critical(alpha));
    System.out.println("\nCritical values for two tails "
            + "\nof the chi-square "
```

```
                + "distribution."
                + "\nAlpha = Significance level = "
                + alpha + ".");
        double alphaHalf = alpha / 2;

        System.out.println("Lower critical value  = "
                + critical(1 - alphaHalf));
        System.out.println("Upper critical value  = "
                + critical(alphaHalf));
        System.out.println("\nTEST");
        //Test to measure the accurateness of the approximation
        test(vCritic);
    }
}//End of Program E341 CritChi2
```

342, page 343. Critical values for the F-distribution (professional implementation).

```
//Program E342 CritF
//Reports   critical values
//for the F-distribution
//and a given significance level.
//Combines Simpson's rule
//with a change of scale.
//An initial approximation is improved
//by analytical hunting.

package ejvol5v2p;

public class CritF {


    //Significance level
    private static double alpha;
    private static double target;
    private static double greatSum;
    private static double Error;
    private static double uCritic;
    private static double vCritic;
    private static double jCritic;
    //Right border or the critical interval
    private static double rBorder;
    private static double h;
```

```
private static int N;
private static double nu1, nu2;
private static double k;

//****************Style: reuse*************
//Returns the beta   function  with   d.f. =
//a(num) and b(den)
private static double g(double x, double a, double b) {
    double g = Math.pow(x, a - 1)
            * (Math.pow(1 - x, b - 1));
    return g;
}

//Returns the integral under the function f
//in within 0 and u. Precision=1/N.
//Simpson's rule adapted for reuse.
//a and b are degrees of freedom
private static double HalfBodySimpsong(double u,
        double a, double b, long N) {
    double hh = u / (2 * N);
    long m = N;
    double sum = g(0, a, b);
    for (int j = 1; j <= m; j++) {
        sum = sum + 4 * g((2 * j - 1) * hh, a, b);
    }
    for (int j = 1; j < m; j++) {
        sum = sum + 2 * g((2 * j) * hh, a, b);
    }
    sum = sum + g((2 * m) * hh, a, b);
    sum = (hh / 3) * sum;
    return sum;
}

//Returns the  beta  function in integral form
//with a and b d.f.
private static double beta(double a, double b) {
    double answer;
    int i = 3;
    int NN = (int) Math.pow(10, i);
    double u = 1;
    answer = HalfBodySimpsong(u, a, b, NN);
    return answer;
}

//****************F density function*************
//Returns the transformed density function of the
```

```
//F distribution with nu1 and nu2 d.f.
private static double FFunction(double t) {
    double z = t / (1 - t * t);
    double f = k * Math.pow(z, nu1 / 2 - 1)
            * Math.pow(nu2 + nu1 * z, -(nu1 + nu2) / 2)
            * ((1 + t * t) / ((1 - t * t) * (1 - t * t)));
    //System.out.println(f);
    return f;
}

//To use this method for another distribution,
//instead of  FFunction(z), write and make a call for your
//new distribution.
private static double f(double z) {
    double f = FFunction(z);
    return f;
}

//Reports the transformed upper u-critical for
//the given distribution.
//Significance level = alpha.
//Precision=1/N.
//Simpson's rule adapted for reuse.
private static double upperU(double u,
        long N,
        double target) {

    double a = Math.pow(nu1, nu1 / 2);
    double b = Math.pow(nu2, nu2 / 2);
    double c = beta(nu1 / 2, nu2 / 2);
    k = a * b / c;
    h = u / N;
    double r = h / 6;
    double sum = 0;
    jCritic = 0;
    for (int j = 0; sum < target; j++) {
        sum = sum
                + r * (f(j * h) + 4 * f((j + 0.5) * h) + f((j + 1) * h))
        jCritic = j + 1;
    }
    jCritic = jCritic - 1;
    greatSum = sum
            - r * (f(jCritic * h) + 4 * f((jCritic + 0.5) * h)
            + f((jCritic + 1) * h));

    double uAprox = (jCritic) * h;
```

```
    return uAprox;
}

//Returns the transformed value z = u/(1-u*u)
private static double z(double u) {
    double z = u / (1 - u * u);
    return z;
}

//Reports the upper z-critical for
//Significance level = alpha.
private static double vCrit(double alpha) {
    double u = 1;
    int i = 4;
    N = (int) Math.pow(10, i);
    uCritic = upperU(u, N, target);
    Error = target - greatSum;
    double zCritic = z(uCritic);
    return zCritic;
}

//*****Analytical hunting*******
//Returns the density function of the t-distribution
private static double ff(double z) {
    double f = k * Math.pow(z, nu1 / 2 - 1)
            * Math.pow(nu2 + nu1 * z, -(nu1 + nu2) / 2);
    return f;
}

//F(b) = h(f(a)  + 4 * f((a+b)/2)  + f(b))-Error;
private static double F(double b) {
    double a = vCritic;
    double hh = (b - a) / 6;
    double Fb = hh * (ff(a) + 4 * ff((a + b) / 2)
            + ff(b)) - Error;
    return Fb;
}

//Returns the  ratio  F(gi)/ ff(gi)
private static double ratio(double gi) {
    double ratio = F(gi) / ff(gi);
    return ratio;
}

//Critical value for the upper tail.
//A professional  approximation is found by
```

```
//analytical hunting
private static double hunting() {
    double gSubi = rBorder;
    double gSubiPlus = 0;
    int iterations = 0;
    int maxIterations = 30;
    boolean more = true;
    while (more) {
        gSubiPlus = gSubi - ratio(gSubi);
        iterations = iterations + 1;
        if (iterations >= maxIterations) {
            more = false;
        }
        if (gSubiPlus == gSubi) {
            more = false;
        }
        gSubi = gSubiPlus;
        /*System.out.println("iteration = " + iterations +
          "New crit-value = " + gSubi);*/
    }
    return gSubiPlus;
}

//Critical value for significance level alpha, two tails
private static double critical(
        double alpha) {
    target = 1 - alpha;
    vCritic = vCrit(target);
    //System.out.println("First approx = " + vCritic);
    double ur = (jCritic + 1) * h;
    rBorder = z(ur);
    vCritic = hunting();
    //System.out.println("Hunting approx = " + vCritic);
    return vCritic;
}

//************Test********
//The cumulative function evaluated at z
private static double cum(double z, long N) {
    double hh = z / (2 * N);
    long m = N;
    double sum = f(0);
    for (int j = 1; j <= m; j++) {
        sum = sum + 4 * f((2 * j - 1) * hh);
    }
    for (int j = 1; j < m; j++) {
```

```
        sum = sum + 2 * f((2 * j) * hh);
    }
    sum = sum + f((2 * m) * hh);
    sum = (hh / 3) * sum;
    return sum;
}

//The cumulative function evaluated at
//vCritic must be equal to the target = 1-alpha.
private static void test(double z) {
    System.out.println(
            "\nArea under the F- density function "
            + "in within 0 and \n" + z);
    z = (Math.sqrt(1 + 4 * z * z) - 1) / (2 * z);
    int NN;
    double answer;
    NN = (int) Math.pow(10, 4);
    answer = cum(z, NN);
    System.out.println(answer);
    System.out.println("Expected = " + target);
    double error = target - answer;
    System.out.println("Forth and back Error = " + error);
}

public static void main(String[] args) {
    alpha = 0.05;
    nu1 = 7;
    nu2 = 6;

    System.out.println("Critical values for one tail "
            + "\nof the F- "
            + "distribution with dfs   "
            + nu1 + " and " + nu2 + ", "
            + "\nAlpha = Significance level = "
            + alpha + ".");

    System.out.println("Lower critical value  = "
            + critical(1 - alpha));
    System.out.println("Upper critical value  = "
            + critical(alpha));
    System.out.println("\nCritical values for two tails "
            + "\nof the chi-square "
            + "distribution."
            + "\nAlpha = Significance level = "
            + alpha + ".");
    double alphaHalf = alpha / 2;
```

```
        System.out.println("Lower critical value  = "
                + critical(1 - alphaHalf));
        System.out.println("Upper critical value  = "
                + critical(alphaHalf));
        System.out.println("\nTEST");
        //Test to measure the accurateness of the approximation
        test(vCritic);
    }
}//End of Program E342 CritF
```

343, page 343. Critical values for a Pareto distribution.
The probability density of a Parteto distribution is

$$f(x) = \begin{cases} 0, & \text{if } x < m, \\ \frac{rm^r}{x^{r+1}}, & \text{if } x \geq m, \end{cases}$$

Let $V > m$ be the lower tail critical value associated to significance level α. It fulfills

$$\alpha = \int_m^V f(x)dx = \int_m^V \frac{rm^r}{x^{r+1}}dx = rm^r \int_m^V \frac{1}{x^{r+1}}dx = rm^r \int_m^V x^{-r-1}dx$$

$$= rm^r(-x^{-r}/r)|_m^V = -\frac{m^r}{x^r}|_m^V = -\frac{m^r}{V^r} + \frac{m^r}{m^r} = 1 - \frac{m^r}{V^r}$$

Therefore

$$\alpha = 1 - \frac{m^r}{V^r}$$

$$\frac{m^r}{V^r} = 1 - \alpha$$

$$\frac{m^r}{1-\alpha} = V^r$$

$$V = \frac{m}{(1-\alpha)^{1/r}}$$

Let $V > m$ be the upper tail critical value associated to significance level α. It fulfills

$$\alpha = \int_V^\infty f(x)dx = \int_V^\infty \frac{rm^r}{x^{r+1}}dx = rm^r \int_V^\infty \frac{1}{x^{r+1}}dx = rm^r \int_V^\infty x^{-r-1}dx$$

$$= rm^r(-x^{-r}/r)|_V^\infty = -\frac{m^r}{x^r}|_V^\infty = 0 + \frac{m^r}{V^r} = (\frac{m}{V})^r$$

Therefore

$\alpha = (\frac{m}{V})^r$

$\alpha^{1/r} = \frac{m}{V}$

$V = \frac{m}{\alpha^{1/r}}$

345, page 350. Too much noise but humble result.

Problems of Chapter 16

349, page 353. Since the toolbox project is very difficult and very interesting, we have provided a constructive answer in `Vol VI StatTrek` in the section of the scientific method in our site `http://www.evoljava.com`. An answer with just the last version of the toolbox can be found in the program that accompanies our course of `Basic Statistics version 18`. Warning: you have the right to become a professional and to develop a very fine and modern toolbox. In that case, learn `Functional Java` that was introduced by Java 8 in 2015. To that aim, we have produced our own version in `Vol 16, Functional Java, easy vs. complex problems` that can be downloaded from our website.

Problems of Chapter 17

403, page 388. When specialized perfection is looked for, the evolutionary pattern is always the same: the evolutionary trend is fast and cheap at the beginning, but then it slows down and finally stops. The character of being a bug is determined here by a fault against the evolutionary trend dictated by the selective criterion, which in the case of race horses is to be faster than everybody. So, we have a bug each time that appears an offspring that is slower than their parents and this is the rule when the evolutionary trend approaches stabilization.

We have therefore abundant bugs in the evolutionary process that is awaken by artificial selection, but where are the left tracks corresponding to an evolution towards perfection? In general they do not appear in the fossil record: the differences between perfection, good quality and mediocrity are many times a matter of coordination of small details and not of great changes. Is this a falsification of our law of abundant bugs and tracks of an evolution towards perfection? No:

Evolution comprises two levels (that might be intertwined). The first is the design of the evolutionary environment with the combinatorial basis and the rules for reproduction, mutation, recombination and selection. And the second, is evolution properly in which the elements of the combinatorial basis are blindly recombined to fit better and better solutions to the fitness function. Experience shows that

- The design of the evolutionary environment is an extremely difficult task because one can easily end with too poor solutions to targeted problems.
- Once one has an evolutionary environment, the ensued evolution has no magic, rather it obeys the ordinary laws of computation, i.e., if a task is feasible, it takes time. So there are easy problems that can be resolved very easily in some few blind computations. But there are complex problems whose solutions demand huge amount of computations. In this last case, one can see that local optima are persistently visited and so they are sure candidates to appear in any process of random sampling. These are the tracks we speak about, thanks to which we can trace an evolution towards high complexity and perfection. Local optima is what defines horses that are valued in 500, 1000, 10000, 1000000 or million of dollars. But their fossils will be almost indistinguishable for most people.

Bibliography

[1] Adleman L (1994)
Molecular computation of solution to combinatorial problems. *Science* 266: 1021–1024(Nov 11).
https://www.usc.edu/dept/molecular-science/papers/fp-sci94.pdf
Verified 31/I/2017 369

[2] Ahituv N, Zhu Y , Visel A, Holt A, Afzal V, Pennacchio L, Rubin E (2007) Deletion of Ultraconserved Elements Yields Viable Mice.
PLoS Biol. 2007 September; 5(9): e234. doi: 10.1371/journal.pbio.0050234.
http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1964772/?tool=pmcentrez
Verified 31/I/2017 371

[3] Akhmechet S (2006) Functional Programming For The Rest of Us.
http://www.defmacro.org/
Verified 31/I/2017 162

[4] Armstrong L (1968) What A Wonderful World. Lyrics.
http://www.lyricsfreak.com/l/louis+armstrong/what+a+wonderful+world_20085347.html
http://www.songfacts.com/detail.php?id=499
Verified 31/I/2017 379

[5] Ashliman D (2009) Human Sacrifice in Legends and Myths
http://www.pitt.edu/~dash/sacrifice.html#contents
Verified 31/I/2017 362

[6] Ayala et al (2008) Science, Evolution and Creationism.
http://www.nap.edu/catalog.php?record_id=11876
Verified 31/I/2017 379

[7] BBC News (2001) Richard Dawkins: The foibles of faith, In Depth: Newsmakers, Friday, 12 October, 2001, 15:25 GMT 16:25 UK
http://news.bbc.co.uk/2/hi/in_depth/uk/2000/newsmakers/1595744.stm#top
Verified 31/I/2017 379

[8] biologicalphysics.iop.org (2013) Biological Molecules 6: An introduction to molecular biology and evolution.
http://biologicalphysics.iop.org/cws/article/lectures/47042
Verified 31/I/2017 366

[9] Black A, Nierstrasz O, Ducasse S, Pollet D (2009) Squeak by Example,
http://squeakbyexample.org/SBE.pdf
Verified 31/I/2017 387

[10] S.R. Bowling; M.T. Khasawneh; S. Kaewkuekool; B.R. Cho (2009) A logistic approximation to the cumulative normal distribution,
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.429.6900&rep=re
Verified 31/I/2017 318

[11] Casio (2017)
http://keisan.casio.com/
Verified 31/I/2017 325

[12] Burden R, D Faires (1985) Numerical Analysis. PWS Boston. 188, 214

[13] Chakraborty A (2007) Calculate Pi To Arbitrary Precision (Sample Java Code).
http://blog.taragana.com/index.php/archive/calculate-pi-to-arbitrary-pre
Verified 31/I/2017 210

[14] Choi I and S Hou (2007) Global extent of horizontal gene transfer. PNAS 2007 104:4489-4494;
http://www.pnas.org/content/104/11/4489
Verified 31/I/2017 354

[15] D'Ambros M, Lanza M (2006) Software Bugs and Evolution: A Visual Approach to Uncover Their Relationship. In Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering), pp. 227 - 236, IEEE Computer Society, 2006.
http://www.inf.usi.ch/faculty/lanza/publications.html
Verified 31/I/2017 382

[16] Darnell J, Lodish H, Baltimore D (1986) *Molecular cell biology.* Scientific American Books. 166, 369, 372

[17] Darwin C (1859) *The origin of species.* Murray, Londres.
http://darwin-online.org.uk/
Verified 31/I/2017 366

[18] Dawkins R (1986) The Blind Watchmaker. W.W.Norton & Company, NY, London.
http://terebess.hu/keletkultinfo/The_Blind_Watchmaker.pdf
Verified 1/II/2017 231

[19] Dawkins R (1976). The Selfish Gene. New York City: Oxford University Press.
https://www.pdf-archive.com/2014/07/21/the-selfish-gene
Verified 1/II/2017 379

[20] Dembski WA (2000). "Intelligent Design Coming Clean". Access Research Network.
http://www.arn.org/docs/dembski/wd_idcomingclean.htm
Verified 31/I/2017 381

[21] Dettman J, Sirjusingh C, Kohn L, Anderson J (2007) Incipient speciation by divergent adaptation and antagonistic epistasis in yeast. *Nature*, 445: 82–85.
http://www.nature.com/nature/journal/v447/n7144/abs/nature05856.html
Verified 1/I/2017 368

[22] Developer (2011)
http://www.developer.com/java.
Verified 31/I/2017

[23] Dexter M(2011)
http://sourceforge.net/projects/eclipsetutorial/
Verified 31/I/2017

[24] dextercowley (2015) Eclipse and Java Video Tutorials. Video tutorials for learning Java OOP programming with Eclipse.
http://sourceforge.net/projects/eclipsetutorial/
Verified 31/I/2017 4

[25] Ebbinghaus H, Flum J, Thomas W (1984), *Mathematical logic*, Springer, N.Y. 359

[26] Easa Networks: Anthropology of Religion
http://www.easaonline.org/networks/religion/index.shtml
Verified 31/I/2017 362

[27] Eckel B (2003) Thinking in Java. Pearson Education, Inc. Upper Saddle River, NJ 07458
http://www.computer-books.us/java_11.php
Verified 31/I/2017 387

[28] EvolJava (2017) Java for the study of Evolution.
http://www.evoljava.com/
Verified 31/I/2017 4

[29] Editor of Nature (2008) Editorial: Spread the word, *Nature* 451, 108.
http://www.nature.com/nature/journal/v451/n7175/full/451108b.html
Verified 31/I/2017 379

[30] Elzanowski A, Ostell J (2008). "The Genetic Codes". National Center for Biotechnology Information (NCBI).
http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi?mode=c
Verified 31/I/2017 364

[31] Epstein R (2004) Genetic Engineering and Its Dangers
http://online.sfsu.edu/~rone/GEessays/gedanger.htm
Verified 31/I/2017 367, 385

[32] FAO (2011) Aquatic Species Distribution Map Viewer
http://www.fao.org/figis/geoserver/factsheets/species.html
Verified 31/I/2017 367

[33] Felleisen M, Findler R, Flatt M, Krishnamurthi S (2001)
How to Design Programs. An Introduction to Computing and Programming. The MIT Press Cambridge,Massachusetts, London, England.
http://www.htdp.org/2003-09-26/
Verified 31/I/2017 387

[34] Fitch A (2011) Biogeography
http://www.nyu.edu/projects/fitch/courses/evolution/html/biogeography.ht
Verified 31/I/2017 367

[35] Foote B, Yoder J (1995) Evolution, architecture and metamorphosis
http://www.laputan.org/metamorphosis/metamorphosis.html
Verified 31/I/2017 163, 386

[36] FossilMuseum (2011) Linnaean Taxonomy
http://www.fossilmuseum.net/Tree_of_Life/Linnaean-Taxonomy.htm
Verified 31/I/2017 366

[37] Fredkin E (2001) Digital philosophy
http://52.7.130.124/wp-content/uploads/2015/07/intro-to-DP.pdf
Verified 31/I/2017 377

[38] Freeland S, Hurst L (1998). "The genetic code is one in a million". J. Mol. Evol. 47 (3): 238–48.
http://www.webpages.uidaho.edu/~stevel/565/literature/The%20genetic%20co
Verified 31/I/2017 371

[39] Frey C (2005) Taylor Polynomials–A Visual Approach to Approximations
http://mathdemos.org/mathdemos/TaylorPolynomials/
Verified 1/II/2017 202

[40] Gilleland M (2010) Big Square Roots
http://www.sixhat.net/square-root-big-numbers-java-20101110.html
Verified 31/I/2017

[41] Glanzberg M (2009) Truth. The Stanford Encyclopedia of Philosophy. Editor Zalta E.
http://plato.stanford.edu/archives/spr2009/entries/truth/
Verified 31/I/2017 359

[42] Gnumeric (2017)
http://www.gnumeric.org/
Verified 31/I/2017 191

[43] Gomez D (2006) Programando con Smalltalk
http://stephane.ducasse.free.fr/FreeBooks.html
Verified 31/I/2017 387

[44] Habgood J(2008) The creation of Creationism. Today's brand of Protestant extremism should worry theologians as well as scientists. The Times Literary Supplement, July 23, 2008
http://www.the-tls.co.uk/articles/private/the-creation-of-creationism/
Verified 31/I/2017 380

[45] Harper (2011) Practical Foundations for Programming. Springer.
https://www.cs.cmu.edu/~rwh/pfpl/2nded.pdf
Verified 31/I/2017 162

[46] Higuchi T., Iwata M., and Liu W., (Editors) , Proceedings of The First International Conference on Evolvable Systems: From Biology to Hardware (ICES96). Lecture Notes in Computer Science, Vol. 1259, Springer-Verlag, Heidelberg, 1997. 161

[47] Holland J (2011) Genetic Algorithms. Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand
http://www2.econ.iastate.edu/tesfatsi/holland.gaintro.htm
Verified 31/I/2017 369

[48] Howard D (2010) Einsteinś Philosophy of Science. The Stanford Encyclopedia of Philosophy. Ed. Zalta E.
http://plato.stanford.edu/archives/sum2010/entries/einstein-philscience/
Verified 31/I/2017 376

[49] Horst S (2011) The Computational Theory of Mind. The Stanford Encyclopedia of Philosophy. Ed. Zalta E.
http://plato.stanford.edu/archives/spr2011/entries/computational-mind/
Verified 31/I/2017 360

[50] Ings S (2008) An eye for the eye. A celebration of one of evolution's crowning glories. Nature 456, 304-309 (2008) — doi:10.1038/456304a. Free article:
http://www.nature.com/news/2008/081119/full/456304a.html
Verified 31/I/2017 370

[51] Jain R, Rivera M, Lake J (1999) Horizontal gene transfer among genomes: The complexity hypothesis Proc. Natl. Acad. Sci. USA, Vol. 96, pp. 3801–3806, March 1999
http://www.pnas.org/content/96/7/3801/F5.expansion.html
Verified 31/I/2017

[52] Jones S (1987) The Implementation of Functional Programming Languages. Prentice Hall, 1987.
http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1
Verified 31/I/2017 162

[53] Jovanovic R (2005) Machin's formula
http://milan.milanovic.org/math/english/pi/machin.html
Verified 31/I/2017 209

[54] Kaiser D (2005) Physics and Feynman's Diagrams. American scientist, vol 93: 156-165.
http://www.americanscientist.org/issues/feature/2005/2/physics-and-feynm
Verified 31/I/2017 376

[55] Kane G, Malcolm J. Perry, Anna N. Zytkow (2000) The Beginning of the End of the Anthropic Principle
http://arxiv.org/abs/astro-ph/0001197
Verified 31/I/2017 374, 376

[56] Kauffman S (1995) At home in the universe. Oxford University Press, NY. 375

[57] Kenneth J. Loceya and Jay T. Lennona (2015) Scaling laws predict global microbial diversity
http://www.pnas.org/content/113/21/5970.abstract
Verified 31/I/2017 361

[58] Krešimir Kumeri (2011) Feynman Diagrams for Beginners
https://arxiv.org/pdf/1602.04182.pdf
Verified 31/I/2017 376

[59] Koelman J (2010) Big Bang, Big Bewilderment
http://www.science20.com/hammock_physicist/big_bang_big_bewilderment
Verified 31/I/2017 361

[60] Klimas E, Skublics S and Thomas D. (1996) Smalltalk With Style. Prentice Hall
http://stephane.ducasse.free.fr/FreeBooks/WithStyle/SmalltalkWithStyle.p
Verified 31/I/2017 387

[61] Koza J, (1980, 1996) Genetic programming. On the Programming of Computers by Means of Natural Selection. A Bradfort Book. The MIT Press. First printing: 1980; Fifth Printing: 1996.
http://posgrado.itlp.edu.mx/mcastro/CE/AG/MIT%20-%20John%20R%20Koza%20-%
Verified 1/II/2017 161, 369

[62] Koza J, (2007) General information about Genetic Programming.
http://www.genetic-programming.org/
Verified 31/I/2017 161, 369

[63] Kuhn T (1969) The Structure of Scientific Revolutions, 2nd. ed., Chicago: Univ. of Chicago.
http://sciencepolicy.colorado.edu/students/envs_5110/structure_of_scientific_revo
Verified 1/II/2017 376

[64] Lanza M (2008) Software evolution.
http://www.slideshare.net/michele.lanza/soft-evo2
Verified 31/I/2017 369

[65] Linde A (1994) The Self-Reproducing Inflationary Universe. Scientific American 271: 48-55
http://www.stanford.edu/%7Ealinde/1032226.pdf
Verified 31/I/2017 375, 376

[66] Leadlay P (2011) Biosynthesis of polyketides antibiotics.
http://www.nature.com/nrmicro/journal/v3/n12/full/nrmicro1287.html
Verified 31/I/2017 167

[67] Marques de Cantu María José (1990) Probabilidad y Estadística para Ciencias Quimico-Biológicas. McGraw-Hill. Mexico. 120

[68] Meffert K, Rotstan N (2011) JGAP: Java Genetic Algorithms Package.
http://jgap.sourceforge.net/
Verified 31/I/2017 369

[69] Mens T(2008) Introduction and Roadmap: History and Challenges of Software Evolution. In: Software Evolution. Mens T, Demeyer S (Eds.), Springer, pp 1–14. 161, 385

[70] Meta-religion (2011) Popol Vuh
http://www.meta-religion.com/World_Religions/Ancient_religions/Central_america/popol_vuh.htm
Verified 31/I/2017 380

[71] Mozilla Org (2011) Bug number 640000
https://bugzilla.mozilla.org/show_bug.cgi?id=640000
Verified 31/I/2017 369

[72] Myers D (2010) Godel's Incompleteness Theorem
http://www.math.hawaii.edu/~dale/godel/godel.html
Verified 31/I/2017. Address must be copied from the latex source. 359

[73] Nasa (2011) Planet Quest
http://planetquest.jpl.nasa.gov/
Verified 31/I/2017 375

[74] NetBeans Developing Applications with NetBeans IDE. (2014)
http://docs.oracle.com/cd/E50453_01/doc.80/e50452/whatsnew.htm
Verified 31/I/2017 4

[75] NetBeans IDE Java Quick Start Tutorial (2015)
https://netbeans.org/kb/docs/java/quickstart.html
Verified 31/I/2017 4

[76] Oracle (2015)
http://www.oracle.com/index.html
Verified 31/I/2017 3

[77] Oracle (2011a) Free Java download
http://java.com/en/download/index.jsp.
Verified 31/I/2017

[78] Oracle (2011b) Java Developer Tutorials and Training.
http://www.oracle.com/technetwork/java/index-jsp-135888.html.
Cited 7 Marz 2011.
Verified 31/I/2017

[79] Oracle (2011c) Class BigInteger
http://download.oracle.com/javase/1.5.0/docs/api/java/math/BigInteger.ht
Verified 31/I/2017 207

[80] Oracle (2017) Class BigInteger
https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html
Verified 31/I/2017 207

[81] Puthoff H (1990?) Quantum Vacuum Fluctuations: A New Rosetta Stone of Physics?
http://www.padrak.com/ine/QUANTFLUX.html
Verified 31/I/2017 361

[82] R Project (2011),
http://www.r-project.org/
Verified 31/I/2017 155

[83] Rafelski J, B Müller (2006) The structured vacuum. Thinking about nothing.
http://www.physics.arizona.edu/~rafelski/Books/StructVacuumE.pdf

Verified 31/I/2017 361

[84] Reznick D (2011) The Guppy project. Science behind the project.
http://cnas.ucr.edu/guppy/scientists/reznick.html

Verified 31/I/2017 368

[85] Riel A (1996) Object Oriented Design Heuristics. Addison-Wesley Professional.
http://c2.com/cgi/wiki?ObjectOrientedDesignHeuristics
Verified 31/I/2017 387

[86] Rodríguez J (2007) On peacebuilding and individual psychology. *Antípoda* 5, 209–237. In Spanish: Sobre la construcción de la paz y la sicología individual.
http://antipoda.uniandes.edu.co/indexar.php?c=Revista+No+05
Verified 31/I/2017 360

[87] ROOT (2011) , http://root.cern.ch/drupal/
Verified 31/I/2017 155

[88] Rothamsted Research (2001) A Beginners' Guide to Molecular Biology
http://www.rothamsted.ac.uk/notebook/gene.htm
Verified 31/I/2017 2

[89] Salzberg S, White O, Peterson J, Eisen J (2001) Microbial Genes in the Human Genome: Lateral Transfer or Gene Loss? Science 292, 903-1906.
http://www.cbcb.umd.edu/~salzberg/docs/ScienceLateralTransfer.pdf
Verified 31/I/2017 364

[90] Schmidhuber J (2000) Algorithmic Theories of Everything
http://arxiv.org/abs/quant-ph/0011122
Verified 31/I/2017 376

[91] Selinger P (2007) Lecture Notes on the Lambda Calculus, Department of Mathematics and Statistics, University of Ottawa, pp. 16
http://www.mathstat.dal.ca/~selinger/papers/lambdanotes.pdf
Verified 31/I/2017. Address must be copied from the latex source. 351

[92] Sklar L (2009) Philosophy of Statistical Mechanics, The Stanford Encyclopedia of Philosophy (Summer 2009 Edition), Edward N. Zalta (ed.),
http://plato.stanford.edu/archives/sum2009/entries/statphys-statmech/.
Verified 31/I/2017 374

[93] Stenger V (2011) The Anthropic Principle.
http://www.talkreason.org/articles/anthro_philo.pdf
Verified 31/I/2017 374

[94] Sweeney T (2008) The Next Mainstream Programming Language: A Game Developer's Perspective . Epic Games.
http://dl.acm.org/citation.cfm?doid=1111037.1111061
Verified 31/I/2017 372

[95] Visual-arts-cork (2011) ENCYCLOPEDIA OF ART. Altamira Cave Paintings
http://www.visual-arts-cork.com/prehistoric/altamira-cave-paintings.htm
Verified 31/I/2017 358

[96] Jose Wudka (1998) What is the "scientific method"?
http://physics.ucr.edu/~wudka/Physics7/Notes_www/node6.html
Verified 31/I/2017 358

[97] Alexander Zien (2000?) A Primer on Molecular Biology - The MIT Press
https://mitpress.mit.edu/sites/default/files/titles/content/978026219509
Verified 31/I/2017 364